Algorithms for Reasoning about Information Flow and Knowledge in Distributed Systems

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

Niklas Metzger

Saarbrücken, 2025

Day of Colloquium	February 23, 2026
Dean of Faculty	Prof. Dr. Roland Speicher
Chair of the Committee	Prof. Dr. Thorsten Herfet
Examination Board	Prof. Bernd Finkbeiner, Ph.D.
	Prof. Yoram Moses, Ph.D.
	Prof. Dr. David Basin
Academic Assistant	Dr. N. Ege Saraç

UNIVERSITÄT
DES
SAARLANDES

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus anderen Quellen oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet. Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder ähnlicher Form in einem Verfahren zur Erlangung eines akademischen Grades vorgelegt.

Declaration of original authorship

I hereby declare that this dissertation is my own original work except where otherwise indicated. All data or concepts drawn directly or indirectly from other sources have been correctly acknowledged. This dissertation has not been submitted in its present or similar form to any other academic institution either in Germany or abroad for the award of any other degree.

Saarbrücken, Oktober 2025

gez. / signed
Niklas Metzger

Abstract

This thesis shows how reasoning about information flow and knowledge, two core hyperproperties in the security and verification of distributed systems, can guide the design of correct-by-construction synthesis algorithms. Traditionally, these notions appear only as system requirements: security constrains information flow, and agents act based on what they know. By reinterpreting them as principles for algorithm design, we develop fundamentally new compositional synthesis methods. For distributed synthesis, whose goal is to automatically build component implementations that jointly satisfy a specification, we introduce explicit assumptions about the necessary information flow between components. These assumptions enable a decomposition into subproblems whose local solutions compose into a globally correct system. For controller synthesis, where the synthesized controller has to correctly govern a plant, we introduce prophecies: declarative statements of what a controller must know about a plant's implementation. Prophecies allow us to construct a universal controller that is correct for all plants and can later be specialized to a concrete plant. Finally, we take first steps toward handling common knowledge, the shared agreement among agents, by lifting existing temporal hyperlogics to express second-order hyperproperties. We also present a monitoring algorithm that checks, during execution, whether a distributed system satisfies a second-order hyperproperty.

Zusammenfassung

Diese Arbeit zeigt, wie die Analyse von Informationsfluss und Wissen, zwei zentrale Hypereigenschaften in der Sicherheit und Verifikation verteilter Systeme, die Entwicklung von korrekt-durch-Konstruktion Synthesealgorithmen positiv beeinflussen kann. Traditionell treten diese Begriffe nur als Systemanforderungen auf: Sicherheit beschränkt den Informationsfluss, und Agenten handeln basierend auf dem, was sie wissen. Indem wir sie als Entwurfsprinzipien für das Algorithmendesign neu interpretieren, entwickeln wir grundlegend neue Methoden für die kompositionelle Synthese. Für die verteilte Synthese, deren Ziel es ist, automatisch mehrere Implementierungen zu erzeugen, die gemeinsam eine Spezifikation erfüllen, führen wir explizite Annahmen über den notwendigen Informationsfluss zwischen Komponenten ein. Diese Annahmen ermöglichen eine Zerlegung in Teilprobleme, deren lokale Lösungen zu einem global korrekten System zusammengesetzt werden können. Für die Controllersynthese, bei der der synthetisierte Controller eine Strecke korrekt steuern muss, führen wir Prophezeiungen ein: deklarative Aussagen darüber, was ein Controller über die Implementierung einer Strecke wissen muss. Prophezeiungen erlauben es uns, einen universellen Controller zu konstruieren, der für jede Strecke korrekt ist und später auf eine konkrete Strecke spezialisiert werden kann. Abschließend unternehmen wir erste Schritte zur Behandlung von Allgemeinwissen, dem geteilten Wissen der Agenten, indem wir bestehende temporale Hyperlogiken so erweitern, dass sie Hypereigenschaften zweiter Stufe ausdrücken können. Zudem präsentieren wir einen Monitoring-Algorithmus, der zur Laufzeit überprüft, ob ein verteiltes System eine Hypereigenschaft zweiter Stufe erfüllt.

Acknowledgements

I am deeply grateful to Bernd Finkbeiner for giving me the opportunity to join his group. Bernd, thank you for the countless discussions, for always making time when needed, and for opening the doors to the research community for me. You shaped not only how I conduct research but also how I approach problems in general. Your rational and strategic way of reasoning and your ability to always keep the big picture in mind are truly inspiring.

The Reactive Systems Group is a role model research group, and I am most fortunate to have been part of it. I fondly remember the many discussions in the old and the new offices, exceptional specialty coffee, deadline pushes, table tennis, eye-rolling for yet another bad joke of mine, and all the, sometimes even productive, deep talks. I want to thank all my wonderful colleagues for this fantastic time: Alfredo, Arthur, Chris, Ege, Florian, Frederik, Frederik, Hadar, Hazem, Iona, Jan, Jana, Jesko, Julian, Malte, Matthias, Max, Mishel, Nico, Noemi, Norine, Oscar, Raven, and Vladimir. The atmosphere you create is one-of-a-kind, and I am rooting for every single one of you.

I was fortunate to have the best office mates imaginable. I once stormed into Norine's office at Saarland University unannounced, yet she welcomed me without hesitation. Thank you for proofreading almost everything of importance, sharing the enthusiasm for our special guests Joschi and Emma, and the constant supply of snacks. I also want to thank Frederik for letting me share the same side of the room in the new office and for being the most kind-hearted person in the world. Thank you to Hadar, Iona, and Mishel for making 1.09 such a wonderful office.

One of the most valuable lessons from my PhD is that research is best enjoyed through collaboration. I want to sincerely thank all my co-authors for the rewarding collaborations and the fun we had while working together: Anne, Bernd, Chris, Dennis, Felix, Frederik, Hadar, Jesko, Norine, Julian, Julià̀n, Julia, Raven, Satya, Tamara, Tom, and Yoram. I also want to thank Alfredo, Jan, Julian, Norine, Raven, and Satya for proofreading and providing me with valuable feedback, and Bernd, Yoram, and David for taking the time to review this thesis.

Over the course of 10 years living in Saarbrücken, I met many wonderful people, most of whom I met during my undergraduate studies at Saarland University. I want to thank all my friends for the great times we shared at university, going out, climbing, hiking, skiing, watching soccer, and for all the joy along the way.

The greatest support comes from my family, and I am grateful for all the love and encouragement they have given me. Thank you to my parents, my brother and his wonderful family, my cousins, my aunts and uncles, and my grandma. Our family has a truly special bond, and I am thankful for all the great moments we have shared, for always being there for each other, and for the unconditional love.

Finally, and most importantly, I would like to thank my wife, Saskia, for her unwavering support, her patience with late-night and weekend work, and for being my safe space. Sharing all the special moments with you makes them infinitely more valuable, and I cannot wait to begin our next chapter together: “Where you invest your love, you invest your life.” – Mumford & Sons.

Contents

Chapter 1

Introduction

Modern computer systems have evolved far beyond the era of single processes on isolated machines. Today, most practical applications are distributed across multiple threads within a single host machine and often span numerous locations worldwide. Notable and highly security-critical examples include financial systems, cloud computing, and social media networks. These systems do not function in isolation but actively *react* to environmental conditions and *interact* among their components, in addition to performing local computations. This so-called *distributed computing* has transformed computer systems by enhancing user availability and enabling high-performance, time-critical computations. However, the distribution of interdependent components introduces significant correctness challenges, such as data corruption, race conditions, and deadlocks, as well as security vulnerabilities, including unauthorized access through impersonation or man-in-the-middle attacks. If such faulty behavior remains undetected in large-scale distributed systems, it can result in severe privacy breaches and substantial financial losses.

A crucial approach to mitigating failures is the use of *formal methods*. Given a formal specification that a system must adhere to, such as a temporal property, formal methods provide verifiable proof that the system either satisfies or violates the specification. For temporal properties, algorithms that formally *verify* specifications against one-component systems have been successfully applied in industry for years, including SAT solvers [36], model checkers [58], and SMT solvers [16]. However, ensuring the correctness of distributed systems through formal methods poses significant challenges. Not only must each component be locally correct, but the combined behavior of all components must also be verified, which exponentially increases the number of scenarios to be analyzed. A widely used type of models for such systems is distributed reactive systems, or short *distributed systems*. Reactive systems operate continuously,

producing outputs in response to observed environment inputs, and are designed to run indefinitely without termination. Typical applications of reactive systems include hardware circuits, control systems where a controller interacts with a provided plant, and communication protocols. For distributed systems, the models are sets of reactive systems accompanied by a topological structure that defines the observations and communications between the individual components. During execution, the components build up knowledge about the environment inputs through their own observations and the information shared by other components. These components, then, act upon a simple but effective principle: Whenever a component makes a decision, it must know the fact that justifies this decision.

Reasoning about knowledge [81] of components in a distributed system is one of the foundational aspects of distributed computing. Each component typically has only a partial view of the system's events, and ambiguity can arise: For example, if one component learns a hidden secret from others, the rest of the agents cannot know that information without implicit communication, so-called information flow. Checking whether a process knows a fact, then checks whether, based on its local observations, an agent can infer that this fact holds. This observation has initiated the development of formal logics [81, 129] and methods for verifying knowledge properties in distributed systems [176, 210, 200, 116, 210], and even the use of knowledge for correct-by-construction algorithms for distributed system development [79]. The first step in computing an agent's knowledge is determining which situations are indistinguishable from the current execution. All indistinguishable situations are grouped, and a component is said to know a fact if that fact holds in every situation within the group. Knowledge, therefore, needs a more involved correctness analysis than traditional functional properties: It is a *hyperproperty* [47]. Hyperproperties, as introduced by Clarkson and Schneider [60], define correctness on the system level, analyzing all traces at once, and not on the execution level, where each execution is analyzed separately. They allow for the direct comparison of two or more traces, e.g., checking if two or more traces are indistinguishable for an agent, and to reason about the existence of such traces in a system. Hyperproperties, and formal logics and methods for hyperproperties, have been extensively studied in security [60] and formal methods [59, 105, 94, 53, 95] research, and are necessary to verify information security in reactive systems.

A central hyperproperty in the field of security is *information flow*. Information-flow security analyzes how sensitive data propagates through a system. Absence of information flow is usually specified as noninterference [122], which requires that an attacker cannot deduce secret information solely by observing public inputs and

outputs. Violations of noninterference have led to high-profile vulnerabilities, such as those exploited by the Meltdown [164] and Spectre [148] attacks on Intel processors. Since its introduction as a formal concept in 2010 [60], the specification [151] and verification [124, 25, 193] of information-flow security properties have been actively developed and remain a major topic in security research, as it is crucial to mitigate security leaks.

This thesis presents a novel approach to incorporating knowledge and information-flow reasoning into correct-by-construction algorithms for distributed systems. We argue that these concepts should not only be treated as properties to be verified, but can also serve as foundational principles for developing new algorithmic solutions for automatic system development, i.e., synthesis. We demonstrate the practical benefits and enhanced effectiveness by designing *compositional* synthesis algorithms that are guided by knowledge and information flow. Our techniques provide new insights into the compositionality of distributed systems and show that the keys to successful decomposition are *information flow* and *knowledge*, not system behavior.

Our first contribution focuses on the necessary information flow between components in a distributed system. We invent new algorithms for synthesizing distributed systems compositionally, where the components assume the reception of certain information during execution. In monolithic (one-component) synthesis, a specification is automatically translated into an implementation that is guaranteed to satisfy the specification against every possible environment behavior, making the task of system development superfluous [55]. *Distributed (multi-component) synthesis* [184] extends reactive synthesis by automatically constructing correct implementations for a specification of a distributed system. Distributed synthesis is particularly intriguing compared to standard reactive synthesis because it generates a set of implementations, each operating with only a partial view of the overall system state. Our synthesis approaches are motivated by the following observation: *compositional synthesis* has the potential to make distributed synthesis scalable [196, 155, 87, 68, 102]. In compositional synthesis, a distributed synthesis problem is divided into local subproblems, each solved independently. After synthesizing the local implementations, these subsystems are composed to form the overall solution. When successful, this approach can mitigate the state-explosion problem. We introduce new techniques for decomposing distributed systems by decomposing their specifications. A key challenge in compositional synthesis is to identify the assumptions that one process p must make about another process q, and vice versa. Local synthesis for p is only feasible once such assumptions are established; with these in place, each process can be synthesized under the guarantee that the

other will satisfy its respective assumption. Previous work on compositional synthesis has primarily focused on inferring *behavioral* assumptions about a subsystem's environment [196, 54, 37, 156, 153, 88, 103]. However, behavioral assumptions are often *too restrictive* to encompass all possible correct implementations of a component, requiring an iterative process of refining assumptions [54]. We introduce a new class of assumptions that capture *information flow*, a hyperproperty, between components, rather than specific behavioral constraints. Based on these assumptions, we develop two compositional synthesis algorithms, one for liveness and one for safety specifications, outperforming existing approaches.

Furthermore, we propose a compositional synthesis method for a special case of distributed systems, namely, controller synthesis. In controller synthesis, the subcomponents are one controller, whose implementation must be synthesized, and one implementation of a plant that is already given. The controller and plant must be correct against all possible environment behaviors. The standard technique to synthesize controllers uses a model of a plant [152, 209] and composes the plant with an automaton for the temporal specification before utilizing a synthesis algorithm. This results in an ω-regular game that can be solved using standard algorithms from reactive synthesis [2, 142, 143]. This method faces significant scalability challenges: Incorporating *all possible* plant behaviors into the plant model to ensure a correct-by-design solution leads to an enormous state space in the plant automaton, making the game-solving process computationally infeasible. We present a new synthesis approach synthesizing a *universal controller*. This generalized version of a controller is correct for every plant for which a solution exists and abstracts away the plant's explicit state space. The abstraction, again, is knowledge guided: For every action that the controller could take, we compute the knowledge that the controller must obtain about the plant and compile it in a so-called *prophecy*. Whenever the prophecy holds, the associated controller action is a correct move. In a second step, we adapt the universal controller to the specific plant model. This new approach improves the scalability, adaptability, and explainability of controller synthesis.

In a last step, we explore properties beyond standard knowledge and information-flow reasoning by introducing a new formal logic for hyperproperties that can reason about complex properties like common knowledge. Common knowledge is an intriguing property in multi-agent systems and specifies a system-wide knowledge of a fact φ. Assume that everyone knows that φ holds, meaning that each individual agent knows the fact. Common knowledge then specifies that *everyone knows that everyone knows that ... φ holds*, adding a possibly infinite chain of reasoning on top of the knowledge

property. This way of reasoning occurs in many complex properties, e.g., asynchronous hyperproperties and Markzukiewicz traces, that are *second-order* hyperproperties. In second-order hyperproperties, an explicit quantification over sets of traces is necessary in the specification. We introduce the first logic to reason about a large class of second-order hyperproperties: An extension of HyperLTL that can quantify explicitly over sets of traces, called Hyper^{2}LTL. For Hyper^{2}LTL, we present a monitoring algorithm to verify if a system satisfies or violates a given Hyper^{2}LTL formula while reasoning over all previously observed executions.

In summary, we make the following contributions:

- We introduce a new class of assumptions for compositional distributed synthesis that are based on information flow. We formalize their construction from LTL specifications, and develop compositional synthesis algorithms for both liveness and safety properties. Our approach enables more scalable synthesis by leveraging hyperproperty-based assumptions instead of restrictive behavioral ones.
- We propose a novel compositional synthesis method for controller synthesis based on knowledge. This method abstracts from the explicit plant state space, annotates controller outputs with knowledge-based conditions, so-called prophecies, and builds an explicit controller based on prophecy evaluations. This approach improves scalability and adaptability compared to traditional approaches.
- We introduce Hyper^{2}LTL, the first temporal logic with quantification over trace sets, enabling the specification and monitoring of complex second-order hyperproperties such as common knowledge.
- We present a monitoring algorithm for a class of second-order hyperproperties and demonstrate its effectiveness on practical benchmarks.

The following sections introduce the underlying concepts of this thesis, including distributed systems, hyperproperties, distributed synthesis, and monitoring of temporal properties. We continue by giving a detailed overview of the contribution of this thesis.

1.1 Distributed Systems and Temporal Properties

Distributed reactive systems are models of multi-component systems that operate independently and communicate with each other. Examples of distributed systems include multi-agent systems [75], network protocols [198], and the execution of smart contracts on blockchains. The usual way of modelling distributed systems is in terms

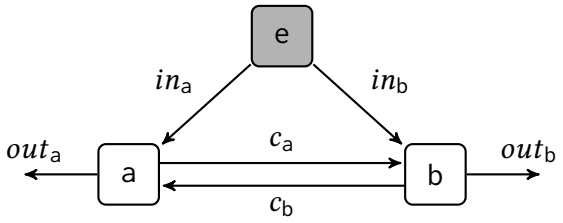


Figure 1.1 An architecture for the two-component bit transmission protocol [81] with two processes a and b, the environment e, inputs in_a and in_b, outputs out_a and out_b communication variables c_a and c_b.

of so-called reactive system [56], where each system component has a designated finite-state model.

Distributed Reactive Systems. Reactive systems [56, 132] continuously react to observations from an unrestricted environment. The execution of a reactive system builds an *infinite trace* on which all environment outputs and system outputs are grouped together. In a *distributed* system, a set of reactive systems is combined, each observing different environment outputs and communicating via specific outputs with each other. The topological structure of a distributed system is specified by an *architecture*.

Example 1. A simple example of a distributed system is the *bit transmission protocol* [81], a 2-component system where input bits in_a are transferred from component a to component b. The architecture of this example is shown in Figure 1.1. The processes a and b receive the inputs in_a and in_b from the environemnt, respectively. The can communicate via c_a and c_b and output on the variables out_a and out_b, respectively.

Trace Properties. The correctness of distributed systems is commonly specified via *temporal properties* [187], which are properties that specify the infinite behavior of a system. Temporal properties traditionally use modalities over time to express properties like *something happens infinitely often* or *at some point in time, something happens*. Trace properties are temporal properties that specify the correctness of a system by reasoning over all its executions individually. The most prominent logic for specifying temporal properties is *linear-time temporal logic* (LTL) [183], which reasons about all executions of a system in isolation and in discrete time steps.

Example 2. In the bit transmission protocol, the specification for process b requires that its output out_b always matches the value of process a's input in_a, resembling the intended behavior that the input bit of a is transmitted to process b. This can be

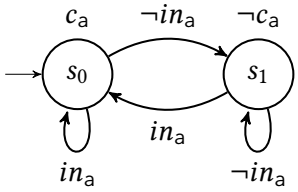
expressed in LTL as

$$\varphi_b = \square(in_{\mathsf{a}} \leftrightarrow \bigcirc out_{\mathsf{b}}),$$

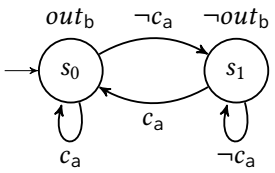
stating that at every time step ($\square$), the value of in_{a} is equivalent to out_{b} in the following step ($\bigcirc$). There are no constraints on a's behavior, so $\varphi_a = \mathit{true}$. Figure 1.2 illustrates an implementation, more specifically a Moore machine [180], for both processes that together satisfy φ_b. State labels indicate process outputs, while edge labels represent inputs. Since b controls out_{b} but cannot directly observe in_{a} (which is only visible to a), it must react to information it does not directly receive. To enable this, a transmits the value via the communication channel c_{a}, which b can observe. In this specific implementation, process a reacts to in_{a} with c_{a} and to $\neg in_{\mathsf{a}}$ with $\neg c_{\mathsf{a}}$ and process b reacts to c_{a} with out_{b} and to $\neg c_{\mathsf{a}}$ with $\neg out_{\mathsf{b}}$.

Temporal properties have long been applied in industry [205], starting with hardware verification [111] and ranging to large-scale verification algorithms with symbolic methods [50, 57]. The latest advances lift temporal logic verification to software [13], where temporal specification languages like TLA+ [159] are successfully used by, for example, Amazon [181] and Microsoft [39] to find errors in system implementations before deployment.

Hyperproperties. Recent security breaches, most famously Meltdown [165] and Spectre [147], raised awareness for a conceptually different class of properties, so-called *hyperproperties*. The authors present powerful *side-channel* attacks, where multiple executions of a system are compared to infer a secret. It is impossible to verify the absence of such security leaks with trace properties as they only reason about single executions. Clarkson and Schneider introduced the notion of temporal hyperproperties [60], predating Spectre and Meltdown, where correctness is stated over sets of traces via existential and universal quantification over traces in the system. Hyperproperties cover a wide range of properties, including information-flow [60, 192, 60], robustness [194], fairness [64], and causality [62, 61], and knowledge [81]. The research area of temporal hyperproperties has evolved drastically since the introduction of HyperLTL[59], the first temporal logic for hyperproperties that extends LTL with trace quantification. On the logical side, temporal hyperproperties have been developed for various system models [74, 10, 182] and explored regarding expressiveness of the logics [63, 48], where special focus has been put on decidability [41]. On the algorithmical side, verification and model-checking [105, 31, 92, 73, 33], monitoring [53, 42, 95, 34], and synthesis [93, 40, 65, 9] techniques have been actively developed and span a wide range of applications.



(a) Process a



(b) Process b

Figure 1.2 An implementation of a bit transmission distributed systems with 2 processes. Process a receives an input bit from the environment on in_a and transfers its value to process b via the communication variable c_a. Process b reacts to c_a by setting out_b equivalent to c_a.

Example 3. We formalize an information-flow hyperproperty that is satisfied by the distributed system in Figure 1.2. It states that process a forwards the information on in_a to process b, specified as a HyperLTL formula.

$$\psi = \forall\pi\forall\pi'.\,\square\left(\left(in_a^{\pi} \leftrightarrow in_a^{\pi'}\right) \rightarrow \left(c_a^{\pi} \leftrightarrow c_a^{\pi'}\right)\right)$$

The formula universally quantifies over a pair of traces π and π' and, for any chosen pair, states that whenever the variable in_a differs between π and π', shown by the annotation of variable in_a with the trace variables π and π', respectively. If this is the case, then the variable c_a must also be different. The implementation in Figure 1.2 satisfies this hyperproperty since process a copies the exact value of in_a to c_a.

Knowledge. Epistemology, the symbolic reasoning about knowledge, dates back to Lewis and Langford [162] in 1959 and has since had a significant impact on the verification of distributed systems [136, 137, 116, 210]. We refer to [81] for a detailed overview of knowledge reasoning. In terms of temporal properties, knowledge was introduced as an extension of LTL, called LTL_K [129], where the additional operator K_A reasons about the knowledge of agent A based on their observations: If, starting from a trace t, a fact holds on every trace that the agent cannot distinguish from t, then the agent knows this fact. Knowledge is a hyperproperty and can be specified in HyperLTL, as we show in the following example.[1]

Example 4. We state that process b knows at every point in time the value in_a in the previous time step, formulated in LTL extended with knowledge modalities [81]. The

[1]While expressing knowledge in HyperLTL is possible for some variants of knowledge formalization, this is not always the case, as shown in [46].

knowledge operator K is annotated with b, denoting that we reason about the knowledge of process b.

$$\varphi = \square\Big(K_{\mathsf{b}}\big(\ominus in_{\mathsf{a}}\big) \vee K_{\mathsf{b}}\big(\ominus \neg in_{\mathsf{a}}\big)\Big)$$

At every point in time, process b can either infer that in_{a} was true at the time step ($\ominus$), or was false at the previous time step. This property can also be stated as the HyperLTL formula with temporal operators about the past

$$\psi = \forall\pi\forall\pi'.\square\Big(\boxminus\big(c_{\mathsf{a}}^{\pi} \leftrightarrow c_{\mathsf{a}}^{\pi'}\big) \rightarrow \ominus\big(in_{\mathsf{a}}^{\pi} \leftrightarrow in_{\mathsf{a}}^{\pi'}\big)\Big)$$

that specifes that on all trace pairs where process b had the same observations up to some time point ($\boxminus$), the value of in_{a} is equivalent in the previous step on both traces.[2]

Knowledge operators were also shown to be useful as conditionals in knowledge-based programs [79] with applications in verification [12, 146, 160] and synthesis [206, 175, 80]. Temporal logics with knowledge can also be used to specify security properties regarding information-flow [14] and noninterference [131] where the absence of knowledge is a witness for satisfying the security property.

1.2 Reactive Synthesis

An intriguing problem in formal methods is reactive synthesis, which dates back to Alonzo Church in 1962 [55] and is, therefore, often referred to as *Church's problem*. In synthesis, the goal is to automatically construct a correct implementation solely from the provided specification, e.g., formulated in LTL, which fascinatingly makes the task of manual implementation superfluous. For specifications in LTL, this problem is decidable, it is 2EXPTIME-complete [186], and a multitude of automata-based [84] and game-based [38] algorithms have been proposed and implemented [141, 179, 191], fostering an active research area that is driven by the yearly SYNTCOMP competition [141].

Distributed Synthesis. In modern reactive systems, distribution is the norm: Implementations typically consist of multiple processes, each with its own set of inputs, which may or may not be shared with other processes. Distributed synthesis extends Church's problem to this setting, aiming to automatically construct implementations composed of several interacting processes. However, Pnueli and Rosner showed that

[2] The standard HyperLTL definition does not contain past operators. For some verification techniques, e.g., monitoring, extending the logic with past operators is straightforward [34].

distributed synthesis is undecidable even for simple architectures with just two independent processes [185]. Despite this negative result, the search for practical algorithms remains ongoing with substantial theoretical progress [171, 184, 154, 89, 168, 167]. Recent progress, particularly in *compositional synthesis* [196, 54, 37, 156, 153, 88, 103], has led to promising new techniques that seek to avoid the inherent complexity of distributed synthesis.

Example 5. The bit transmission implementation shown in Figure 1.2 is a possible result of a distributed synthesis algorithm that constructs two implementations that comply with the architecture in Figure 1.1 and to the LTL specification $\varphi = \square(in_{\mathsf{a}} \leftrightarrow \bigcirc out_{\mathsf{b}})$. Note that there are infinitely many implementations that can be synthesized for φ: Process a can, at every point in time, either copy the value of in_{a} to c_{a}, or copy the negation of the value of in_{a} to c_{a}. A possible *decomposition* of φ is the following:

$$\varphi_{\mathsf{a}} = \square(in_{\mathsf{a}} \leftrightarrow c_{\mathsf{a}}) \qquad\qquad \varphi_{\mathsf{b}} = \square(c_{\mathsf{a}} \leftrightarrow out_{\mathsf{b}})$$

The individual synthesis tasks for φ_{a} and φ_{b} as monolithic (single process) synthesis tasks build a solution for the distributed synthesis task, exactly the one in Figure 1.2. Note that the output of a in time step n is part of the input of b in step $n + 1$, and vice versa.

Controller Synthesis. Supervisory control [190], like reactive synthesis, seeks the automatic construction of systems that are guaranteed to meet their formal specifications.[3] While reactive synthesis typically starts from a logical specification, supervisory control is based on an open system model called the *plant*. In practice, both perspectives are often combined: For instance, a robot's behavioral objectives may be given as a temporal logic formula, while its environment, including obstacles, humans, and physical interactions, is described by a plant model. This has led to methods that either automatically abstract physical constraints into a plant automaton [202, 26, 212] or directly model the plant as an event-based automaton [152, 209]. The plant model is then composed with the automaton that represents the logical specification. This results in an ω-regular game that can be solved using established reactive synthesis algorithms [2, 142, 143].

Example 6. Assume the same specification of Example 5 and a controller synthesis problem where process b in Figure 1.2b is provided as a model of the plant. The controller synthesis task is to construct an implementation for the process a that, together with

[3] See [78, 197] for a detailed comparison of the two approaches.

process b, satisfies the specification. Then, in contrast to the distributed synthesis problem in Example 5, there is only one correct controller that can be synthesized, namely the one shown in Figure 1.2a.

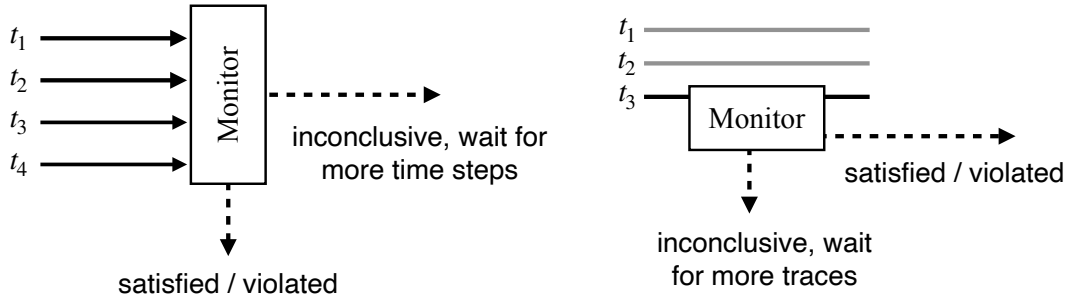
Not only must a controller be synthesized for the formula, but it must also be suitable for the provided plant model. Controller synthesis is a special case of distributed synthesis and is 2EXPTIME-complete since, provided with the implementation of the plant, it can be encoded as a monolithic synthesis problem.

1.3 Monitoring

Monitoring is a widely employed technique to check, during execution, whether systems satisfy specified properties. A monitor, running alongside the deployed system, observes its behavior and evaluates it against a given property. This approach is particularly valuable for systems that cannot be fully verified before deployment and may therefore contain errors. In runtime monitoring, correctness properties define the set of acceptable individual executions of a system. These properties can be specified using a variety of formalisms, including language-based approaches such as extended regular expressions [199], query languages [173], and rule-based systems [17].

Trace Property Monitoring. We consider the monitoring problem of temporal properties. Bauer et al. [22] introduced a foundational framework for monitoring LTL properties that is based on a three-valued semantics: A property can be determined as already satisfied, already violated, or inconclusive, indicating that further observations are needed to reach a verdict. The properties for which finding of a verdict is possible are called monitorable.

Example 7. The specification of the bit transmission protocol is monitorable. Assume a trace that is correct until time step n and in_{a} is *true* at n. A monitor can never conclude that the execution satisfies the property: Every new time step raises a new obligation to react correctly to in_{a}. Now we add $\neg out_{\mathsf{b}}$ to the trace at position $n+1$. This trace now violates the specification at time point $n+1$, and for no further execution, the trace can satisfy the formula. Therefore, the specification is monitorable with a definite verdict in case of a violation.



(a) The parallel model.

(b) The sequential model.

Figure 1.3 The two models for monitors that verify hyperproperties during execution of a system. In Figure 1.3b, finite traces are observed sequentially, one after another. For every new trace, the monitor evaluates if the property holds. In the parallel model in Figure 1.3a, n traces are observed in parallel, and the monitor evaluates the traces against the formula for every new time step.

Hyperproperty Monitoring. Monitoring hyperproperties [95] extends monitoring of trace properties to complex hyperproperties, such as information security, robustness, and knowledge. A fundamental question in hyperproperty monitoring is how to present the input, which now consists of multiple execution traces, to the monitor. Fundamentally, the monitor receives either all traces simultaneously or incrementally, one at a time. The two input models we consider are the *parallel*, with simultaneous traces, and the *sequential model*, with incremental traces [95]. A visualization can be found in Figure 1.3. In the parallel model, the number of traces is fixed, and the monitor observes the values of the next time step for all (possibly infinite) traces at every computation step. In the sequential model, *finite* traces are provided one after another, and the monitor computes verdicts for a potentially unbounded number of observed traces.

Example 8. We return to the distinguishability hyperproperty presented before. Recall the following formula

$$\psi = \forall\pi\forall\pi'.\,\square\left(\left(in_{\mathsf{a}}^{\pi} \leftrightarrow in_{\mathsf{a}}^{\pi'}\right) \rightarrow \left(c_{\mathsf{a}}^{\pi} \leftrightarrow c_{\mathsf{a}}^{\pi'}\right)\right)$$

where ψ states that whenever on two traces the value of in_{a} is different, then also the values of c_{a} are different. This property is monitorable; whenever it is violated, it will remain violated for every future trace or extension. Consider the following two finite

traces, which we monitor in the sequential setting.

$$t_1 = \{in_a, c_a\}, \{in_a, c_a\}, \{in_a, c_a\}, \{in_a, c_a\}$$
$$t_2 = \{\neg in_a, \neg c_a\}, \{in_a, c_a\}, \{\neg in_a, \neg c_a\}, \{\neg in_a, \neg c_a\}$$

A monitor observing both traces will not yet come to a verdict and output *inconclusive*: For these two traces, the formula is satisfied, but there might be executions of the system that are not yet observed and violate the formula. One such execution is the following:

$$t_3 = \{\neg in_a, \neg c_a\}, \{\neg in_a, c_a\}, \{\neg in_a, \neg c_a\}, \{\neg in_a, \neg c_a\}$$

Even though the values of in_a on t_1 and t_3 are different in the second time step, the values of c_a are equivalent. This is a violation of ψ and will remain a violation no matter what traces are observed in future steps.

1.4 Contributions

This thesis presents novel algorithms for the synthesis and monitoring of distributed systems, guided by the principles of knowledge reasoning. We develop new compositional synthesis techniques based on information flow and prophecies, and introduce a monitoring approach capable of verifying second-order hyperproperties, such as common knowledge.

1.4.1 Distributed Synthesis with Information Flow Assumptions

The most promising technique for distributed synthesis is compositional synthesis, where the overall synthesis task is decomposed into multiple sub-tasks that, combined, solve the distributed synthesis problem. Compositional synthesis critically depends on finding assumptions that a process p must make about another process q, and vice versa. The local synthesis of p is impossible without such assumptions, but once the assumptions are known, one can build each individual process while given the guarantee that the implementation of the other process will satisfy the assumption. Earlier research on deriving assumptions for compositional synthesis has focused on inferring *behavioral* assumptions on the environment of a subsystem [54, 37, 169]. However, behavioral assumptions are *too strong* to capture all possible correct implementations of a component.

Information Flow Assumptions. We present a novel class of assumptions that formalize the *information flow* between components rather than specific behavioral constraints. Information flow is a hyperproperty that states that a component will receive some information presented on the environment variables. We formalize information flow in a distributed system by computing *distinguishable* traces, trace pairs on which the goal formula dictates an observable difference; if they could not be distinguished, the specification could not be satisfied. This technique is adapted from knowledge reasoning, where a so-called agent *knows* a fact if and only if it holds on all traces that are indistinguishable for this agent. We turn this reasoning into *assumptions* for the agents during compositional synthesis. Based on the assumption that on two traces that *must be* distinguishable for an agent, the agent *will be* able to distinguish them, it will provide a correct implementation. Correctness, however, is trickier than for trace property assumptions: The information-flow assumption only specifies that two traces will be distinguishable, not *how* they will be distinguishable; this will be chosen by the implementation of the agent that observes the difference. As a result, it is not yet possible to construct an executable implementation for the process in question, since such an implementation would need to correctly decode the information provided by its partner process. However, determining how to decode this information is only feasible once the encoding method used by the sending process is known.

Example 9. Recall the information-flow assumption that we stated for process b and the LTL goal formula $\varphi = \square(in_{\mathsf{a}} \leftrightarrow \bigcirc out_{\mathsf{b}})$.

$$\psi = \forall\pi\forall\pi'.\square\left(\left(in_{\mathsf{a}}^{\pi} \nleftrightarrow in_{\mathsf{a}}^{\pi'}\right) \rightarrow \left(c_{\mathsf{a}}^{\pi} \nleftrightarrow c_{\mathsf{a}}^{\pi'}\right)\right)$$

It relates all trace pairs that must be distinguishable by process a, namely those where the values of in_{a} are different at least once. The assumption that this difference will be observable is on the right-hand side of the formula; whenever in_{a} is different, then the b-observable variables c_{a} must also be different. There are infinitely many implementations of a that satisfy this hyperproperty: It can choose to copy the value of in_{a} to c_{a} or its dual value in every time step. A correct implementation of b that assumes the information-flow assumption has to provide correct reactions to every possible encoding of information by process a.

Compositional Synthesis with Information Flow Assumptions. To address this challenge, we synthesize a prototype implementation for the process that is correct for *any* sender implementation, provided the sender fulfills the information-flow assumption. This prototype, referred to as a *hyper implementation*, differs from a concrete

implementation in that it has access to the original (unencoded) information. With this access, a hyper implementation can always produce the correct output to satisfy the specification. In the final implementation, the original information is no longer directly available and must be reconstructed from the communication received. Nevertheless, the information-flow assumption ensures that such reconstruction is always possible, making direct access to the original information unnecessary. Our compositional algorithm operates as follows: We introduce *necessary information-flow assumptions* for distributed systems with two processes and present a method for automatically deriving these assumptions from process specifications given in linear-time temporal logic (LTL). We strengthen information-flow assumptions to the notion of *time-bounded* information-flow assumptions in the case of liveness LTL goals, which characterize information that must be received in finite time, and *prefix* information-flow assumptions for safety LTL formulas. For both approaches, we present synthesis procedures that compute hyper implementations and the respective composition and decomposition techniques. Our findings are backed by experimental evaluations and a prototype implementation that show the effectiveness of this conceptually new compositional synthesis approach.

1.4.2 Controller Synthesis with Prophecies

We introduce a compositional synthesis approach for controller synthesis, a special (and computationally easier) case of distributed systems. In this setting, the system consists of a controller, whose implementation must be synthesized, and a plant, whose behavior is already specified. The combined system must satisfy the specification for all possible environment behaviors. Traditionally, controller synthesis composes a plant automaton [202, 26, 212] or a plant model [152, 209] with an automaton for the temporal specification, resulting in an ω-regular game that can be solved using established reactive synthesis algorithms [2, 142, 143]. However, this approach often suffers from scalability issues: Exploring the full structure of the plant makes synthesis computationally challenging. Our new approach circumvents this by synthesizing a *universal controller*, which is correct for any plant for which a solution exists. In a subsequent step, the universal controller is specialized to the given plant model. This is achieved by annotating the controller's output decisions with conditions, so-called *prophecies* on the plant's behavior, about the required *knowledge* of the plant to perform the next action.

Prophecies for Controllers. The major observation is that the exploration of the plant's state space can be avoided by solving the controller synthesis problem *abstractly*, i.e., solely based on the specification. This is accomplished by introducing assumptions about the plant's branching structure, which we refer to as *prophecies*. To explain the concept of prophecies, we take a detour via algorithms for reactive synthesis. Reactive synthesis creates an implementation that satisfies some (LTL) specifications. The key task of any synthesis algorithm is to provide sufficient memory for an implementation, based on the given specification, so that the formula's correctness can be guaranteed. For example, obligations that depend on the future or past of a system's branching structure must be stored so that they can later be satisfied. The amount of memory needed to guarantee correctness defines the maximal state space of the resulting implementation. For controller synthesis, this task is accompanied by a second problem: The memory of the plant must be aded to the controller's memory. The structure of the plant additionally *restricts* the behavior of an implementation. We use *prophecies*, abstract conditions over the plant's future branching structure, to avoid the full exploration of the state space of the plant.

Example 10. Consider the controller c, the plant p, and the external environment e. Each of these processes has a Boolean output variable: o_{c}, o_{p}, and o_{e}, respectively. At every time step, each process selects the value of its output variable based on the history of all previous outputs of every component. The temporal specification is given by the following LTL formula:

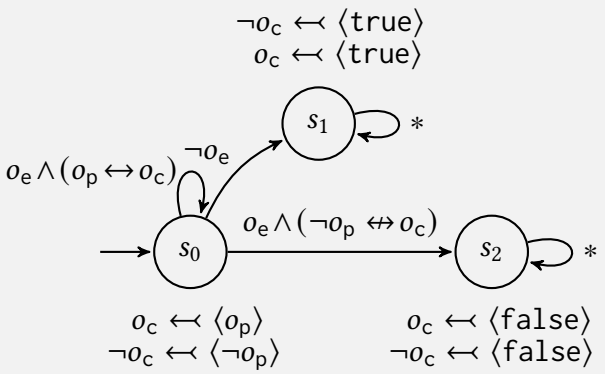
$$\varphi \coloneqq (o_{\mathsf{c}} \leftrightarrow o_{\mathsf{p}})\, \mathcal{W} (\bigcirc \neg o_{\mathsf{e}}). \tag{1.1}$$

This formula requires the controller to match the plant's output as long as the environment's output remains true (specified with the operator weak until $\mathcal{W}$). Since the controller cannot observe the plant's output before making its own decision, constructing a correct controller independently of the plant is impossible: Unless the environment sets its output to false, the controller must know the plant's output one step in advance. We capture this requirement using the prophecies: $\langle o_{\mathsf{p}} \rangle$ indicates that the plant will output true, and $\langle \neg o_{\mathsf{p}} \rangle$ indicates that the plant will output false in the respective cases. A correct controller responds to $\langle o_{\mathsf{p}} \rangle$ by setting o_{c} to true, and to $\langle \neg o_{\mathsf{p}} \rangle$ by setting o_{c} to false at least until $\neg o_{\mathsf{e}}$ was observed.

Prophecies are a useful tool to make synthesizing a *universal controller*, a controller whose outputs are conditioned under prophecies, feasible.

Synthesis of Universal Controllers. Building on this insight, we introduce a new synthesis method that decouples the construction of a *universal controller*, derived solely from the temporal specification, from its subsequent adaptation to a concrete plant model. The universal controller's output choices are annotated with conditions on the plant's structure, referred to as *prophecies*: The controller selects an output only if the plant fulfills the corresponding prophecy, which is verified in a later step. This separation enables the universal controller to be synthesized without reference to the plant, significantly improving scalability.

Example 11. We show a universal controller for the previous specification. For every output and state, i.e., o_p and $\neg o_\mathsf{p}$ and s_0, s_1, s_2 for this example, the prophecies are shown with $\langle o_\mathsf{p} \rangle$ and $\langle \neg o_\mathsf{p} \rangle$, specifying that an output only gets picked if the plant satisfies the condition in $\langle\rangle$. In state s_0, the controller has to copy the value of o_p to its own output variable o_c. In state s_1, no obligation is left for the controller, and the controller can choose the output under the condition *true*. State s_2 is an unreachable state if the controller behaves correctly.



The universal controller can later be refined to an explicit plant by checking the plant's behavior against the prophecies.

We introduce an algorithm for synthesizing universal controllers from safety specifications, representing prophecies using tree automata. Additionally, we demonstrate that universal controllers are *most permissive*, encompassing all controller strategies that are correct for any realizable plant model. We provide an exploration algorithm that derives a concrete controller from the universal controller for a given plant model, and prove that this algorithm is sound and complete for safety specifications. Finally, we present experimental results using our prototype tool, UNICON [107], which show that our approach achieves better scalability than the standard method on synthesis problems involving large plant models.

1.4.3 Monitoring Second-Order Hyperproperties

Hyperproperties, as introduced by Clarkson and Schneider [60], encompass a broad spectrum of requirements, including information-flow security policies and epistemic properties that describe the knowledge of agents in distributed systems. The most widely used specification language for expressing hyperproperties and providing a semantic foundation for various verification techniques is HyperLTL [59]. HyperLTL extends LTL with quantification over traces. Although HyperLTL and its variants have been successfully applied in many contexts [105, 182, 74], their expressiveness is inherently limited, leaving several important hyperproperties beyond their reach. A notable example is *common knowledge*, which is essential in distributed systems for ensuring coordinated actions [81, 130]. Common knowledge within a group of agents means not only that each agent individually knows a condition φ holds, but also that every agent knows that every agent knows φ, and so on, forming an infinite hierarchy of nested knowledge.

The key limitation of HyperLTL is its restriction to *first-order quantification*: it can only quantify over individual traces, not over sets of traces. While this suffices for specifying individual knowledge, where the existence or absence of a single indistinguishable trace violating φ is relevant, it is insufficient for expressing properties like common knowledge, which require quantification over sets of traces and the ability to capture infinite chains of reasoning.

Temporal Logic for Second-Order Hyperproperties. We introduce Hyper^{2}LTL, a temporal logic for hyperproperties that allows for *quantification over trace sets*. In Hyper^{2}LTL, the existence of a trace π where the condition φ is common knowledge can be expressed as follows (using a slightly simplified syntax):

$$\exists \pi.\, \exists X.\ \pi \in X \wedge \left(\forall \pi' \in X.\, \forall \pi''.\, \Big(\bigvee_{i=1}^{n} \pi' \sim_i \pi''\Big) \rightarrow \pi'' \in X\right) \wedge \forall \pi' \in X.\, \varphi(\pi').$$

The second-order quantifier $\exists X$ asserts the existence of a set X of traces such that: (1) π is included in X; (2) X is closed under the observations of each agent, meaning that for every trace π' in X, all traces π'' that are indistinguishable from π' for some agent i are also in X; and (3) every trace in X satisfies φ. The existence of such a set X characterizes precisely when φ is common knowledge on π. In our work, we demonstrate that Hyper^{2}LTL provides a natural and expressive framework for specifying a range of hyperproperties beyond the capabilities of HyperLTL, including epistemic properties like common knowledge.

Monitoring Algorithm. We introduce a monitoring algorithm for second-order hyperproperties for a finite trace semantics of Hyper^{2}LTL, called Hyper^{2}LTL$_\text{f}$. Monitoring provides a practical and scalable approach to ensuring that a system's execution complies with its formal specification. Unlike traditional verification methods such as model checking or theorem proving, monitoring operates directly on the execution traces of the running system rather than analyzing a system model. For the parallel model, we demonstrate that monitoring second-order hyperproperties specified in Hyper^{2}LTL$_\text{f}$ can be reduced to the monitoring of first-order hyperproperties. In the sequential model, while the general monitoring problem is undecidable, we identify a tractable subclass: *Monotone* second-order hyperproperties. A second-order hyperproperty is called $\oplus$-monotone if its satisfaction on a set of traces guarantees satisfaction on all supersets; it is $\ominus$-monotone if a violation on a set implies a violation on all supersets. Monotonicity enables the monitor to provide conclusive results, regardless of future traces. We present an inference system for monotonicity and a monitoring algorithm that incrementally analyzes the observed traces until a definitive verdict can be reached. This algorithm is implemented in the tool `MoSo`, which we evaluate on benchmarks including common knowledge and planning scenarios, demonstrating promising results.

1.5 Publications

This thesis is based on the following peer-reviewed publications.

[97] Bernd Finkbeiner, Niklas Metzger, Yoram Moses. **Information Flow Guided Synthesis**. *34th International Conference on Computer-Aided Verification (CAV 2022)*

[35] Raven Beutner, Bernd Finkbeiner, Hadar Frenkel, Niklas Metzger. **Second-Order Hyperproperties**. *35th International Conference on Computer-Aided Verification (CAV 2023)*

[98] Bernd Finkbeiner, Niklas Metzger, Yoram Moses. **Information Flow Guided Synthesis with Unbounded Communication**. *36th International Conference on Computer-Aided Verification (CAV 2024)*

[34] Raven Beutner, Bernd Finkbeiner, Hadar Frenkel, Niklas Metzger. **Monitoring Second-Order Hyperproperties**. *23rd International Conference on Autonomous Agents and Multi-agent Systems, (AAMAS 2024)*

[99] Bernd Finkbeiner, Niklas Metzger, Satya Prakash Nayak, Anne-Kathrin Schmuck. **Synthesis of Universal Safety Controllers**. *31st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2025)*

1.6 Related Work

We survey the related work relevant to this thesis, with emphasis on knowledge reasoning, information-flow security, distributed and controller synthesis, and logical frameworks for hyperproperties.

Knowledge Reasoning. Reasoning about knowledge dates back to the first epistemic logics by von Wright [211] in 1951 and Hintikka [135] in 1962. The original work on common knowledge dates back to Lewis [163] in 1959 and was first applied to distributed systems by Halpern and Moses [130] in 1990. We refer to [81] for an in-depth discussion. Regarding formal methods for knowledge reasoning, the fundamental temporal logic LTL [183] was extended to epistemic operators in $\text{LTL}_{\text{K,C}}$ [81, 129]. The logic was used for building verification algorithms for model checking [176, 200, 116] and bounded model checking [210], also for strategic versions of the logic [139] and probabilistic knowledge [138]. Regarding reactive synthesis for epistemic logics, there exist explicit [177, 45] and symbolic [140] approaches for automatically constructing one-component systems, and distributed systems [178]. Knowledge as conditions in programs has been introduced as knowledge-based programs [79, 160], facilitating correct-by-construction distributed systems via knowledge reasoning [206, 146, 12].

Information Flow Security. Information-flow security analyzes how sensitive data propagates within a system. Key properties include noninterference [122], which ensures that changes in high-security inputs do not affect low-security outputs, and observational determinism [213], which requires that identical sequences of low-security inputs yield identical low-security outputs. Differential privacy is another important property, guaranteeing that similar inputs result in only limited changes to the outputs [77]. Most related to our approaches is declassification, where some information flow is permitted or even necessary; declassification properties specify the conditions and extent to which information may be released [193]. These properties are inherently relational, as they compare multiple system executions. Most research on information-flow policies focuses on verification; see [25] and [151] for comprehensive overviews. Many information-flow policies are universally quantified over traces, allowing the

verification problem to be reduced to standard, non-relational property checking. A common technique is self-composition [18, 70], which constructs a product system from multiple copies of the original and applies conventional verification methods. Variants of self-composition are central to many approaches for verifying hyperproperties, including those beyond universally quantified cases [105, 82, 121, 33, 31, 66]. In contrast to our contributions, previous information security approaches only verified the absence of information flow, and do not analyze the required information flow.

Distributed Synthesis. Distributed synthesis was first explored by Pnueli and Rosner [184], who immediately concluded that the problem is unrealizable. This result was later refined in [158] to ring architectures and in [106] to information forks. In recent years, bounded synthesis [24] and distributed synthesis via hyperproperty synthesis [204], as well as synthesis of distributed systems with causal memory [100, 90, 134] have been the focus of distributed synthesis research.

Compositional synthesis has been extensively studied in the context of *complete information*, where all processes have access to all environment outputs [156, 153, 88, 103], with recent advancements in [3]. However, in this work, we focus on compositional approaches for distributed systems, where processes operate under incomplete information about the environment outputs. Compositionality has been leveraged to enhance distributed synthesis across various domains, such as reactive controller synthesis [5, 134].

A closely related approach is assume-guarantee synthesis [54, 37], which relies on behavioral guarantees for each process and assumptions about the behavior of other processes. Recent advancements include an iterative refinement of assumptions for distributed systems [169]. Another method is remorse-free dominance [69], avoiding explicit assumptions and guarantees by using a weaker winning condition, resulting in implicit assumptions. Additionally, certificates specifying partial behavior of components have been proposed to guide synthesis iteratively [101]. Unlike these approaches, which rely on behavioral assumptions, our work introduces the novel concept of using information-flow assumptions for synthesis.

In reactive controller synthesis, compositionality offers significant improvements over monolithic synthesis [5, 134]. Assume-guarantee control synthesis, which aligns with our synthesis paradigm, enables local components to infer assumptions about others to achieve their goals [54, 37]. When assumptions are insufficient, they can be iteratively refined [169]. Alternative approaches include weakening acceptance condi-

tions to dominance [69] or using certificates to iteratively specify partial component behavior [101].

Controller Synthesis. Regarding universal controllers, a notable characteristic is their *permissiveness*, which ensures they encompass all control strategies that are valid for any realizable plant. This concept is well-established in supervisory control, where permissiveness is a fundamental property of the computed supervisor [52]. However, in this context, permissiveness pertains to enabling all behaviors that a *single plant* satisfies. Similarly, permissive solutions have been explored in reactive synthesis [27, 44, 112], primarily to ensure variability in resulting implementations. Recent studies have also examined permissiveness in control applications of reactive synthesis [145, 8], though these efforts remain focused on a fixed plant model.

Another area of research investigates the synthesis of control strategies that are valid for a specific set of plant models. For instance, *dominant strategies* [68, 104] are designed to work for all realizable plants, while *admissible strategies* [28, 20] are correct for the largest possible set of realizable plants. Other approaches [7, 49, 97, 98] impose reasonable assumptions on plant models to synthesize control strategies tailored to a restricted set of plants. However, these methods are limited to computing *a single control strategy* that must work for the entire set of considered plants. In contrast, our approach synthesizes a universal controller capable of adapting to *different control strategies* based on the specific plant model that only scales in the size of the specification.

Lastly, *prophecy variables* are a well-known proof technique used to access information about future events [1], commonly used to (partially) eliminate non-determinism. Recent work by Beutner and Finkbeiner [32] leverages prophecy variables to develop a complete game-based algorithm for the automatic verification of hyperproperties. Here, the existential player in the two-player game gains access to prophecies over the future behavior of the universal player and bases its decision on those. Inspired by, but in contrast to, this approach, we use the term *prophecies* to reason about the branching structure of the plant.

Logics for Hyperproperties. In recent years, numerous logics have been introduced for the formal specification of hyperproperties, often by extending temporal logics with explicit path quantification. Examples include HyperLTL and HyperCTL* [59], HyperQPTL [188], HyperPDL [128], and HyperATL* [29, 30]. Other approaches extend first-order logics with equal-level predicates [66, 109], or investigate (ω)-regular [123, 43] and context-free hyperproperties [113], as well as hyperproperties over data and

modulo theories [92, 65]. Hyper^{2}LTL is the first temporal logic designed to reason about second-order hyperproperties, enabling the expression of a wide range of existing and prominent hyperproperties (including epistemic and asynchronous hyperproperties).

Regarding finite trace semantics for temporal logics, the semantics used when monitoring systems against temporal hyperproperties, Giacomo and Vardi [119] introduced LTL$_f$, a logic for interpreting LTL on finite traces, tailored for AI systems. Since its introduction, LTL$_f$ and its extensions have been applied to model checking [15, 189], satisfiability analysis [110], synthesis [120, 126], and planning [51]. The logic combining knowledge and LTL, called LTL$_{K,C}$, originates from Fagin et al. [81] and has been widely used in multi-agent systems (MASs) verification [149, 150, 86]. Similar to infinite trace semantics, most finite trace logics for hyperproperties extend temporal logics with explicit quantification over traces or paths, such as HyperLTL$_f$ [95] and HyperLDL$_f$ [118]. Tools for model checking knowledge in MASs include `MCK` [116] and `MCMAS` [166], with the latter also supporting finite trace semantics [150]. While these tools and approaches are tailored for knowledge, Hyper^{2}LTL$_f$ provides a general framework for second-order hyperproperties that includes common knowledge.

Chapter 2

Preliminaries

In this chapter, we define the preliminaries for this thesis. The model of systems we consider is discrete-time, where the system reacts to the outputs of an uncontrollable environment.[1] The correctness of these systems is defined against temporal logic properties that express behavior over infinite system executions.

2.1 System Models

Architectures. We consider distributed systems of two components that communicate on shared variables. The *architecture* of a distributed system for two processes p and q is defined by a tuple $(I_p, I_q, O_p, O_q, O_e)$, where I_p, I_q, O_p, O_q, and O_e are all subsets of the set $\mathcal{V}$ of boolean variables. The output variables of the components are the sets O_p and O_q. The environment, which is the uncontrollable component of the system, has output variables O_e that are observed by (a subset of) the system components. We use the terms *environment outputs* and *external inputs* simultaneously for O_e. O_p, O_q and O_e form a partition of $\mathcal{V}$. Finally, I_p and I_q are the *input variables* of components p and q, respectively. The inputs and outputs are disjoint, i.e., $I_p \cap O_p = \emptyset$ and $I_q \cap O_q = \emptyset$. Each of the inputs I_p and I_q of the components is either an output of the environment or an output of the other component, i.e., $I_p \subseteq O_q \cup O_e$ and $I_q \subseteq O_p \cup O_e$. For a set $V \subseteq \mathcal{V}$, every subset $V' \subseteq V$ defines a *valuation* of V, where the variables in V' have value *true* and the variables in $V \setminus V'$ have value *false*.

Strategies. A *strategy* for a process p is a function $f_p : (2^{I_p})^* \to (2^{O_p})$ that maps finite traces over p's input variables to a valuation of p's output variables. The strategy

[1] The notion of environment outputs is often called *inputs* in reactive system literature. For clarity, we use the term outputs to distinguish the local inputs of the components from the uncontrolled variables.

chooses the current output with respect to the (complete) history of inputs. Given two strategies f_p and f_q, the *composition* $f_p \parallel f_q$ of the two strategies is the function $f : (2^{O_e})^* \to (2^{\mathcal{V}})$ mapping finite traces over environment output variables to valuations of all variables. We define $f(\epsilon) = f_p(\epsilon) \cup f_q(\epsilon)$ and, for $v \in (2^{O_e})^*, x \in 2^{O_e}$, $f(v \cdot x) = (f_p(g_p(v)) \cup f_q(g_q(v)) \cup x)$, where g_p and g_q map the sequences of environment outputs to the respective sequences of process inputs. Formally, the functions are defined as $g_p(\epsilon) = \epsilon, g_p(v \cdot x) = g_p(v) \cdot ((x \cup f_q(g_q(v))) \cap I_p)$ and $g_q(\epsilon) = \epsilon, g_q(v \cdot x) = g_p(v) \cdot ((x \cup f_p(g_p(v))) \cap I_q)$. The traces produced by the composition of two strategies $f = f_p \parallel f_q$ is defined as

$$Traces(f) = \{x_0 x_1 \ldots \in (2^O)^\omega \mid x_k = f(i_0 i_1 \ldots i_{k-1}) \text{ for all } k \in \mathbb{N} \text{ for some } i_o i_1 i_2 \ldots \in (2^{O_e})^\omega\}.$$

For a trace $t \in \Sigma^\omega$ and index $i, j \in \mathbb{N}$ define, we define $t[i] \in \mathcal{V}$ as the ith element in t; and $t[i, j]$ as the fragment of t from i to j.

Moore machines. For a set of variables (atomic propositions) $\mathcal{V}$ divided into inputs I and outputs O, with $I \cap O = \emptyset$, a 2^O-labeled 2^I-transition system $\mathcal{T}$ is a 4-tuple (S, s_0, τ, o), where S is a set of states, $s_0 \in S$ is an initial state, $\tau : S \times 2^I \to S$ is a transition function, and $o : S \to 2^O$ is a labeling function. An implementation of an architecture $(I_p, I_q, O_p, O_q, O_e)$ is a pair $(\mathcal{T}_p, \mathcal{T}_q)$, consisting of $\mathcal{T}_p$, a 2^{O_p}-labeled 2^{I_p} transition system, and $\mathcal{T}_q$, a 2^{O_q}-labeled 2^{I_q} transition system. The *composition* $\mathcal{T} = \mathcal{T}_p \parallel \mathcal{T}_q$ of the two transition systems $(S^p, s_0^p, \tau^p, o^p)$ and $(S^q, s_0^q, \tau^q, o^q)$ is the $2^{O_p \cup O_q}$-labeled 2^{O_e}-transition system (S, s_0, τ, o), where $T = S^p \times S^q, s_0 = (s_0^p, s_0^q), \tau((s^p, s^q), x) = (\tau^p(s^p, (x \cup o^q(s^q)) \cap I_p), \tau^q(s^q, (x \cup o^p(s^p)) \cap I_q)), o(s^p, s^q) = o^p(s^p) \cup o^q(s^q)$, where $x \in 2^{O_e}$. An initial run $T(i_0, i_1, \ldots) = s_0 s_1 \ldots \in S^\omega$ for an infinite sequence of inputs $i_0, i_1 \ldots \in 2^{O_e}$ is an infinite sequence of states produced by the transition function such that $s_i = \tau(s_{i-1}, i_{i-1})$ for all $i \in \mathbb{N}$ and s_0 is the initial state. For a finite input sequence $t = i_0 i_1 \cdots i_{k-1} \in (2^{O_e})^*$ and process i, $\mathcal{T}$ produces a finite path $s_0 s_1 \cdots s_k$ and an output sequence $o(s_0)o(s_1) \cdots o(s_k) \in (2^{O_i})^*$ such that $\tau(s_j, i_j) = s_{j+1}$. We write $out(\mathcal{T}, t)$ to denote the latest output, i.e., $o(s_k)$. The set of traces $Traces(\mathcal{T})$ of an implementation $\mathcal{T} = (\mathcal{T}^p, \mathcal{T}^q)$ is then defined as all $(o(s_0) \cup i_0)(o(s_1) \cup i_1) \ldots \in (2^{\mathcal{V}})^\omega$ where $T(i_o i_1 \ldots) = s_0 s_1 \ldots$ for some $i_o i_1 i_2 \ldots \in (2^{O_e})^\omega$. For $\mathcal{T} = (S, s_0, \tau, o)$ and $s \in S$, we denote a transition system $\mathcal{T}' = \mathcal{T}(s)$ as a shorthand for $\mathcal{T}' = (S, s, \tau, o)$, where the structure of the system is equivalent, but the initial state is changed to s. We use the term *implementation* and symbol $\mathcal{I}$ whenever we refer to a system that can be either a strategy or a Moore machine.

2.2 Automata

Finite Word Automata. A non-deterministic automaton $\mathcal{A}$ over an alphabet Σ is a tuple (Q, q_o, δ, F) where Q is a set of states, q_o is the initial state, $\delta : Q \times \Sigma \to 2^Q$ is a transition function, and F is a set of accepting states. For an input word $\sigma_0\sigma_1 \ldots \sigma_k \in \Sigma^k$, a finite word automaton (NFA) $\mathcal{F}$ accepts a finite run $q_0q_1 \ldots q_k \in Q^k$ where $q_i \in \delta(q_{i-1}, \sigma_{i-1})$, if $q_k \in F$. An automaton is *deterministic* δ is deterministic.

Infinite Word Automata. An *ω-automaton* $\mathcal{A}$ over an alphabet Σ is defined as a tuple (Q, q_0, δ, Ω), where Q is a finite set of states, $q_0 \in Q$ is the initial state, $\delta\colon Q \times \Sigma \to Q$ is the transition function, and $\Omega \subseteq Q^\omega$ is the acceptance condition. The unique *run* of $\mathcal{A}$ starting from state q on a trace $t \in \Sigma^\omega$, denoted by $run(\mathcal{A}, q, t)$, is a sequence of states $\rho \in Q^\omega$ such that $|\rho| = |t| + 1$, $\rho[0] = q$, and $\delta(\rho[i], t[i]) = \rho[i+1]$ for all $i \in dom(t)$. A run ρ is *accepting* if $\rho \in \Omega$. The language $\mathcal{L}(\mathcal{A})$ consists of all traces t for which the unique run $run(\mathcal{A}, q_0, t)$ is accepting.

A safety automaton (DSA) is a specific type of ω-automaton with a safety acceptance condition $\Omega = safe(F)$, where $F \subseteq Q$ is a set of safe states, and the set of runs only visits states in F. It is known that any LTL specification φ can be translated into an equivalent ω-automaton $\mathcal{A}$, such that $\mathcal{L}(\varphi) = \mathcal{L}(\mathcal{A})$, with a double exponential blow-up [11]. A safety LTL specification φ is an LTL specification for which the corresponding automaton $\mathcal{A}$ is a safety automaton. The procedure for translating a safety LTL specification into an equivalent safety automaton is denoted by LTLtoDSA [157, 161].

Given an automaton $\mathcal{A} = (Q, q_0, \delta, \Omega)$ and a Moore machine $\mathcal{T}$ over inputs I and outputs O with $2^{I \cup O} \subseteq \Sigma$, the composition $\mathcal{A} \times \mathcal{T}$ is defined as a product automaton $(Q \times S, (q_0, s_0), \delta')$, where $\delta'\colon (Q \times S) \times 2^I \to (Q \times S)$ is a partial transition function given by $\delta'((q, s), \sigma) = (\delta(q, \sigma \cup o(s)), \tau(s, \sigma))$. The set of runs of $\mathcal{A}$ starting from q that correspond to runs of $\mathcal{A} \times \mathcal{T}$ is denoted by $Runs(\mathcal{A}, q, \mathcal{T})$. The set of reachable states in $\mathcal{A} \times \mathcal{T}$, starting from the initial state (q_0, s_0), is denoted by $reach(\mathcal{A} \times \mathcal{T})$, where a state (q, s) is reachable if there exists a run ρ of $\mathcal{A} \times \mathcal{T}$ such that $\rho[0] = (q_0, s_0)$ and $\rho[k] = (q, s)$ for some $k \geq 0$. We refer to a 2-tape automaton if the alphabet Σ of an automaton is the product of a different alphabet Σ', i.e., $\Sigma = \Sigma' \times \Sigma'$.

Universal Automata. A *universal* Büchi (or co-Büchi) automaton has the same form (Q, q_0, δ, Ω) as the non-deterministic automaton. The transitions are interpreted *universally*, not existentially, meaning that not one transition can be chosen but all transitions of the transition relation must be taken. A run on an input word $\sigma_0\sigma_1 \ldots$ of a universal automaton is a S-labeled Υ-tree $r : \Upsilon^* \to \Sigma$ such that the root $r(\epsilon) = q_0$,

and for every node $n \in \Upsilon^*$ and every transition $\delta(r(n), \sigma_{|n|}) = q'$ it holds that q' is a child of n: there exists $v \in \Upsilon^*$ such that $r(n \cdot v) = q'$. A run is accepting if all branches of r are accepting. Universal co-Büchi automata are the dual of nondeterministic Büchi automata [125].

Tree Automata. A tree automaton over 2^O-labeled 2^I-trees is defined as a tuple $\mathcal{A} = (Q, q_0, \Delta, \Omega)$, where Q is a finite set of states, $q_0 \in Q$ is the initial state, $\Delta \subseteq Q \times 2^O \times (2^I \to Q)$ is the transition relation, and $\Omega \subseteq Q^\omega$ is the acceptance condition.

A run of the tree automaton $\mathcal{A}$ on a strategy $s : (2^I)^* \to 2^O$ is a Q-labeled 2^I-tree $\eta : (2^I)^* \to Q$ such that $\eta(\epsilon) = q_0$ and, for every input sequence $t\alpha \in (2^I)^*$, there exists a transition $(\eta(t), s(t), f) \in \Delta$ where $f(\alpha) = \eta(t\alpha)$. For an input sequence $\alpha_0\alpha_1 \cdots \in (2^I)^*$, the run (tree) $\eta : (2^I)^* \to Q$ generates a path, i.e., a sequence of states $\eta(\epsilon)\eta(\alpha_0)\eta(\alpha_0\alpha_1)\cdots$. The run (tree) is considered accepting if every path generated by the run satisfies the acceptance condition Ω. The language of the tree automaton, $\mathcal{L}(\mathcal{A})$, is the set of all trees that are accepted by $\mathcal{A}$.

2.3 Specifications

Trace properties. Our specifications are defined over the variables $\mathcal{V}$ which instantiate a set of atomic propositions. For a set $V \subseteq \mathcal{V}$ of variables, a *trace* over V is an infinite sequence $x_0x_1x_2 \ldots \in (2^V)^\omega$ of valuations of V. A *trace property* P over the variables $\mathcal{V}$ is a set $P \subseteq (2^\mathcal{V})^\omega$ of traces, defining which traces belong to the trace property and which ones do not. Two trace sets $V, V' \subset \mathcal{V}$ that are disjoint, i.e., $V \cap V' = \emptyset$, can be *combined* by synchronously iterating over the time steps and unionizing the sets of variables $x_0x_1x_2 \ldots \sqcup y_0y_1y_2 \ldots = (x_0 \cup y_0)(x_1 \cup y_1)(x_2 \cup y_2) \ldots$. Likewise, *projecting* a trace onto a set of variables $V' \subseteq \mathcal{V}$, and thereby generating a new trace, is formed by intersecting the valuations with V' at each position: $x_0x_1x_2 \ldots |_{V'} = (x_0 \cap V')(x_1 \cap V')(x_2 \cap V') \ldots$. An implementation $\mathcal{I}$ satisfies a trace property P if $Traces(\mathcal{I}) \subseteq P$. We classify into sub-categories of trace properties with respect to the traces they contain. A finite trace $t \in (2^\mathcal{V})^*$ is a bad prefix of a trace property P if $t \cdot t' \notin P$ for all $t' \in (2^\mathcal{V})^\omega$.

Definition 1 (Safety properties)**.** *A trace property is a* safety *property if every violation has a* bad prefix*.*

Intuitively, a safety property is violated once some bad state is reached. The dual of safety properties are *co-safety* properties, which must reach a *good state* at some point. Let a finite trace $t \in (2^\mathcal{V})^*$ be a good prefix of a trace property P if $t \cdot t' \in P$ for all

$t' \in (2^{\mathcal{V}})^{\omega}$. Co-safety properties are the dual of safety properties: once the formula is satisfied, it will never be violated again.

Definition 2 (Co-safety properties)**.** *A trace property is a* co-safety *property if every word in the language has a* good prefix*.*

This characterization traces back to Halpern and Pearl's characterization of temporal properties into different expressivity classes [170].

Hyperproperties. A hyperproperty [60], again defined over a set of variables $\mathcal{V}$, is a set $H \subseteq 2^{(2^{\mathcal{V}})^{\omega}}$. It specifies a set of sets of traces such that each set of traces $\mathbb{T} \in H$ defines the traces of a system. An implementation $\mathcal{I}$ satisfies a hyperproperty H iff the set of its traces is an element of H, i.e., $\mathit{Traces}(\mathcal{I}) \in H$. Some hyperproperties, namely 2-hyperproperties, can be defined by binary relations over traces. A relation $R \subseteq (2^{\mathcal{V}})^{\omega} \times (2^{\mathcal{V}})^{\omega}$ of pairs of traces defines the hyperproperty H, where a set $\mathbb{T}$ of traces is an element of H iff for all pairs $t, t' \in \mathbb{T}$ it holds that $(t, t') \in R$. In HyperLTL, 2-hyperproperties are expressed as formulas with two universal quantifiers and no existential quantifiers. 2-hyperproperties can conveniently be represented as 2-tape automata whose accepting traces consist of all the tuples of traces that are related by R. For the 2-hyperproperty defined by R, the language of the 2-tape automaton is then $\mathcal{L}(\mathcal{A}) = \{(\sigma_0, \sigma_0')(\sigma_1, \sigma_1') \ldots \mid (\sigma_0\sigma_1 \ldots, \sigma_0'\sigma_1' \ldots) \in R\}$.

2.4 Temporal Logics

Linear-time Temporal Logic. The most prominent specification language for trace properties is linear-time temporal logic (LTL) [183]. LTL is defined with respect to a set of atomic propositions $\mathcal{V}$, in our case, the variables of an architecture, and evaluated over infinite traces. In contrast to the reactive systems we use in our distributed system model, LTL formulas do not distinguish between inputs and outputs; this is left up to the specifier. We define LTL in the standard way with the following syntax rules, where atomic propositions a are belong to a set of variables $\mathcal{V}$.

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

In addition to Boolean conjunction and negation, LTL contains the temporal operators *next* $\bigcirc$ specifying a formula over the next time-step, and the operator $\varphi_1 \,\mathcal{U}\, \varphi_2$ (named *until*), where φ_1 has to hold until φ_2 holds. The semantic evaluation of an LTL formula is as follows:

$$
\begin{array}{lll}
t, i \vDash a & \text{iff} & a \in t[i] \\
t, i \vDash \neg\varphi & \text{iff} & t, i \nvDash \varphi \\
t, i \vDash \varphi_1 \wedge \varphi_2 & \text{iff} & t, i \vDash \varphi_1 \text{ and } t, i \vDash \varphi_2 \\
t, i \vDash \bigcirc \varphi & \text{iff} & t, i+1 \vDash \varphi \\
t, i \vDash \varphi_1 \,\mathcal{U}\, \varphi_2 & \text{iff} & \exists j.\ j \geq i.\ t, j \vDash \varphi_2 \text{ and } \forall k.\ i \leq k < j.\ t, k \vDash \varphi_1
\end{array}
$$

We use the usual derived Boolean operators disjunction $\varphi_1 \vee \varphi_2$, implication $\varphi_1 \rightarrow \varphi_2$, and equivalence $\varphi_1 \leftrightarrow \varphi_2$, as well as the temporal operators *eventually* $\Diamond \varphi \equiv \mathit{true}\,\mathcal{U}\,\varphi$, *globally* $\Box\varphi \equiv \neg\Diamond\neg\varphi$, *weak until* $\varphi_1 \,\mathcal{W}\, \varphi_2 = (\varphi_1 \,\mathcal{U}\, \varphi_2) \vee \Box\varphi_2$, and *release* $\varphi_1 \,\mathcal{R}\, \varphi_2 = \varphi_2 \,\mathcal{W}\, \varphi_1$. A finite trace t of length n satisfies an LTL formula φ iff the evaluation stops at index n with *true*.

LTL can be extended to reasoning about the past with the operators *previously* $\ominus$ and *since* $\mathcal{S}$. We add the following two semantic rules to the previous ones:

$$
\begin{array}{lll}
t, i \vDash \ominus \varphi & \text{iff} & i > 0 \text{ and } t, i-1 \vDash \varphi \\
t, i \vDash \varphi_1 \,\mathcal{S}\, \varphi_2 & \text{iff} & \exists j.\ j \leq i.\ t, j \vDash \varphi_2 \text{ and } \forall k.\ i \geq k > j.\ t, k \vDash \varphi_1
\end{array}
$$

With the past operators included, one can specifically specify conditions on the exact previous time-step with $\ominus$. In addition, $\varphi_1 \,\mathcal{S}\, \varphi_2$ specifies that sometime in the past φ_2 was true, and since then φ_1 is true. Although formulas in LTL with past operators can be exponentially more succinct [172], it is known that LTL and LTL with past operators have the same expressiveness [115].

Specifications over architectures are conjunctions $\varphi_p \wedge \varphi_q$ of two LTL formulas, where φ_p is defined over $O_p \cup O_e$, i.e., φ_p relates outputs of the component p to the outputs of the environment, and φ_q is defined over $O_q \cup O_e$. We call these specifications the *local* specifications of the component. An implementation *satisfies* a specification φ if the traces of the implementation are contained in the specification, i.e., $\mathit{Traces}(\mathcal{I}^p, \mathcal{I}^q) \subseteq \varphi$. We also use the abbreviation $t \vDash \varphi$ if $t, 0 \vDash \varphi$ and say that t finitely violates φ, denoted by $t \nvDash_f \varphi$ if there exists $n \in \mathbb{N}$ s.t. for all $m > n$ it holds that $t, m \nvDash \varphi$.

Temporal Logic with Knowledge. The common specification logic for knowledge properties follows temporal reasoning in LTL style, and adds operators for knowledge

statements on top. The logic $\mathrm{LTL}_{\mathsf{K},\mathsf{C}}$ [81] extends LTL syntax with the operators for *knows* K, *everyone knows* E, and *common knowledge* C:

$$\psi := a \mid \neg\psi \mid \bigcirc\psi \mid \psi_1 \,\mathcal{U}\, \psi_2 \mid \mathsf{K}_A\psi \mid \mathsf{E}\psi \mid \mathsf{C}\psi$$

where A is a set of agents. Let t, t' be two traces, then $t[0,i] =_{a_j} t'[0,i]$ if t and t' are indistinguishable for agent $a_j \in A$ in the first i steps. Given a set of traces $\mathbb{T}$ and a trace t we define

$$\begin{aligned}
&t, i \vDash_{\mathbb{T}} a && \text{iff} \quad a \in t(i)\\
&t, i \vDash_{\mathbb{T}} \neg\psi && \text{iff} \quad t, i \nvDash_{\mathbb{T}} \psi\\
&t, i \vDash_{\mathbb{T}} \psi_1 \wedge \psi_2 && \text{iff} \quad t, i \vDash_{\mathbb{T}} \psi_1 \text{ and } t, i \vDash_{\mathbb{T}} \psi_2\\
&t, i \vDash_{\mathbb{T}} \bigcirc\psi && \text{iff} \quad t, i+1 \vDash_{\mathbb{T}} \psi\\
&t, i \vDash_{\mathbb{T}} \psi_1 \,\mathcal{U}\, \psi_2 && \text{iff} \quad \exists j.\ j \geq i.\, t, j \vDash_{\mathbb{T}} \psi_2 \text{ and } \forall k.\ i \leq k < j.\, t, k \vDash_{\mathbb{T}} \psi_1\\
&t, i \vDash_{\mathbb{T}} \mathsf{K}_{a_j}\psi && \text{iff} \quad \forall t' \in \mathbb{T}.\, t[0,i] =_{a_j} t'[0,i] \rightarrow t', i \vDash \psi\\
&t, i \vDash_{\mathbb{T}} \mathsf{K}_{A'}\psi && \text{iff} \quad t, i \vDash_{\mathbb{T}} \bigwedge_{a_j \in A'} \mathsf{K}_{a_j}\psi\\
&t, i \vDash_{\mathbb{T}} \mathsf{E}\psi && \text{iff} \quad t, i \vDash_{\mathbb{T}} \bigwedge_{a_j \in A} \mathsf{K}_{a_j}\psi\\
&t, i \vDash_{\mathbb{T}} \mathsf{C}\psi && \text{iff} \quad t, i \vDash_{\mathbb{T}} \mathsf{E}^{\infty}\psi
\end{aligned}$$

The *everyone knows* operator E expresses that all agents know that ψ holds. The semantics of the common knowledge operator C correspond to the infinite iteration, or transitive closure, of "everyone knows that everyone knows that ..." ψ.

Temporal Logics for Hyperproperties. A convenient specification language for hyperproperties is the temporal logic HyperLTL [59], which extends LTL with trace quantification. In addition to the variables $\mathcal{V}$ being the atomic propositions, the logic is defined w.r.t. a set of trace variables V.

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

where $a \in \mathcal{V}$ and $\pi \in V$. The trace variables on atomic propositions make expressing properties like "ψ *must hold on all traces*" possible, expressed by $\forall\pi.\, \psi$. Dually,

one can express that "*there exists a trace on which ψ holds*", denoted by $\exists\pi.\ \psi$ and all combinations of the quantifications. The semantics of HyperLTL is defined w.r.t. a set of traces $\mathbb{T}$, which are the traces of the system, and the variable assignment $\Pi : V \to \mathbb{T}$. During the evaluation of the trace quantifiers, the trace variables are assigned to explicit traces of the trace set $\mathbb{T}$.

$$
\begin{array}{lll}
\Pi, i \vDash_{\mathbb{T}} a_\pi & \text{iff} & a \in \Pi(\pi)(i) \\
\Pi, i \vDash_{\mathbb{T}} \neg\psi & \text{iff} & \Pi, i \nvDash_{\mathbb{T}} \psi \\
\Pi, i \vDash_{\mathbb{T}} \psi_1 \wedge \psi_2 & \text{iff} & \Pi, i \vDash_{\mathbb{T}} \psi_1 \text{ and } \Pi, i \vDash_{\mathbb{T}} \psi_2 \\
\Pi, i \vDash_{\mathbb{T}} \bigcirc\psi & \text{iff} & \Pi, i+1 \vDash_{\mathbb{T}} \psi \\
\Pi, i \vDash_{\mathbb{T}} \psi_1 \,\mathcal{U}\, \psi_2 & \text{iff} & \exists j.\ i \leq j.\, \Pi, j \vDash_{\mathbb{T}} \psi_2 \text{ and } \forall k.\ i \leq k < j.\, \Pi,\ k \vDash_{\mathbb{T}} \psi_1 \\
\Pi, i \vDash_{\mathbb{T}} \forall\pi.\psi & \text{iff} & \forall t \in \mathbb{T}.\, \Pi[\pi \mapsto t], i \vDash_{\mathbb{T}} \psi \\
\Pi, i \vDash_{\mathbb{T}} \exists\pi.\psi & \text{iff} & \exists t \in \mathbb{T}.\, \Pi[\pi \mapsto t], i \vDash_{\mathbb{T}} \psi
\end{array}
$$

An implementation $\mathcal{I}$ satisfies a HyperLTL formula ψ if $\emptyset, 0 \vDash_{Traces(\mathcal{I})} \psi$. Note that we use two versions of trace variable annotations for variables, either as a subscript a_π or as a superscript in_{a}^{π} whenever a subscript already exists.

HyperQPTL [188] adds, on top of the trace quantification of HyperLTL, also propositional quantification (analogous to the propositional quantification that QPTL [201] adds on top of LTL). For example, HyperQPTL can express a promptness property, which states that there must exist a bound (which is common among all traces), up to which an event must have happened. We can express this as $\exists q.\forall\pi.\ \Diamond q \wedge (\neg q)\,\mathcal{U}\, a_\pi$ which states that there exists an evaluation of proposition q such that (1) q holds at least once, and (2) for all traces π, a holds on π before the first occurrence of q. The syntax obtains an additional element, the quantification over propositions:

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

where $a \in \mathcal{V}$ is an atomic proposition in a set of propositions $\mathcal{V}$ and $\exists q.\psi$ and $\forall q.\psi$ is the existential and universal quantification of atomic propositions, respectively. In addition to the trace assignment of HyperLTL, the semantics of HyperQPTL is defined w.r.t. a propositional assignment $\Lambda : \mathcal{V} \to \mathbb{B}^\omega$, where $\mathbb{B}$ is the set containing *true* and

false. The following rules are added to the semantic evaluation

$$\Pi, \Lambda \vDash_{\mathcal{T}} \exists q.\, \varphi \quad \text{iff} \quad \exists \tau \in \mathbb{B}^{\omega}.\, \Lambda[q \mapsto \tau] \vDash \varphi$$
$$\Pi, \Lambda \vDash_{\mathcal{T}} \forall q.\, \varphi \quad \text{iff} \quad \forall \tau \in \mathbb{B}^{\omega}.\, \Lambda[q \mapsto \tau] \vDash \varphi,$$

where $\Lambda[q \mapsto \tau]$ maps the propositional trace τ to the propositional variable q. All other semantic rules are changed accordingly, introducing Λ to the evaluation.

2.5 Reactive Synthesis

Monolithic Synthesis. Reactive synthesis is the task of automatically constructing an implementation that, given the outputs of an unconstrained environment, produces its own outputs in response to satisfy a given specification. More formally, given a specification φ or trace property P, the synthesis problem is to find an implementation $\mathcal{I}$ that satisfies φ or P, respectively.

Definition 3 (Monolithic Synthesis)**.** *Let φ be an LTL specification. The synthesis problem is to find an implementation $\mathcal{I}$ such that $\mathcal{I} \vDash \varphi$.*

A specification φ or trace property P is *realizable* in a given architecture if such an implementation exists, and *unrealizable* if not. The synthesis problem for LTL with one-component systems, also called full-observation synthesis, is 2EXPTIME-complete [186]. We refer to [91] for an overview. The synthesis problem for HyperLTL is known to be undecidable [94].

Distributed Synthesis. Distributed synthesis extends monolithic synthesis to distributed systems, where an architecture specifies the interconnection of components. Given an architecture and one specification φ_i for each component, the distributed synthesis problem is to find an implementation for each component such that all specifications are satisfied.

Definition 4 (Distributed Synthesis)**.** *Let $(I_p, I_q, O_p, O_q, O_e)$ be an architecture and φ_p, φ_q be LTL specifications. The two component distributed synthesis problem is to find a 2^{O_p}-labeled 2^{I_p} transition system $\mathcal{T}_p$ and a 2^{O_q}-labeled 2^{I_q} transition system $\mathcal{T}_q$ such that $\mathcal{T}_p \parallel \mathcal{T}_q \vDash \varphi_p \wedge \varphi_q$.*

For distributed systems that inherently contain partial observation of the components, the problem of synthesizing systems for LTL specification is undecidable, even for two-component systems [184].

Controller Synthesis. Controller synthesis is a special case of two-component distributed synthesis, where the implementation of one component is already provided. We consider a special case of controller synthesis where each component observes all outputs of the other components, i.e., input propositions of a process i are the union of the output propositions of all other processes $I_i = \bigcup_{j \neq i} O_j$. We formalize the controller synthesis problem as follows:

Definition 5 (Controller Synthesis)**.** *Let $(I_c, I_p, O_c, O_p, O_e)$ be an architecture, φ be an LTL specification, and the 2^{O_p}-labeled 2^{I_p} transition system $\mathcal{T}_p$ be the implementation of the plant. The controller synthesis problem is to find a 2^{O_c}-labeled 2^{I_c} transition system $\mathcal{T}_c$ for the controller such that $\mathcal{T}_c \parallel \mathcal{T}_p \vDash \varphi$.*

The controller synthesis problem for LTL is 2EXPTIME-complete [186].

Assume-Guarantee Synthesis. A promising way to synthesize distributed systems is *assume-guarantee synthesis* [54, 169]. It describes the problem of decomposing the synthesis process into multiple synthesis procedures to benefit from the existing full-observation synthesis techniques. This can be achieved by computing assumptions and guarantees on the behavior of other components that, together with the goal specification, are the input to the synthesis procedure. Assumptions can be defined in the following way using trace properties and set theory.

Definition 6 (Assumptions)**.** *Let P and A be trace properties. A is a* sufficient *assumption if $A \rightarrow P$ is realizable. It is a necessary assumption if for every sufficient assumption A' it holds that $A' \subseteq A$.*

An assumption ψ can also be specified in LTL and is sufficient for the LTL property ψ' if the formula $\psi \rightarrow \psi'$ is realizable. We define assume-guarantee in the following way:

Definition 7 (Assume-guarantee synthesis)**.** *Let P_p and P_q be the trace properties as the goals of processes p and q, respectively. The assume-guarantee synthesis problem is to find two assumptions A_p and A_q s.t. $A_p \rightarrow (P_p \wedge A_q)$ and $A_q \rightarrow (P_q \wedge A_p)$ are realizable.*

For goal specifications defined in LTL ψ_p and ψ_q, assume-guarantee synthesis is the problem of finding LTL formulas φ_p and φ_q s.t. $\varphi_p \rightarrow (\psi_p \wedge \varphi_q)$ and $\varphi_q \rightarrow (\psi_q \wedge \varphi_p)$. We will extend the assume-guarantee synthesis to hyperproperty assumptions and guarantees in the following sections.

Chapter 3

Distributed Synthesis with Information Flow Assumptions

Distributed Synthesis [184] is the problem of automatically constructing implementations for processes in a distributed architecture that, together, conform to an intended behavior. This chapter introduces a new class of assumptions for *compositional* synthesis of distributed systems. The new assumptions constrain the information flow between components rather than behavior. Our work is motivated by the observation that *compositional synthesis* can substantially improve the scalability of distributed synthesis [196, 155, 87, 68, 102]. In compositional synthesis, the synthesis task is decomposed into local subproblems, one for each component (see Figure 3.1 and Chapter 1 for further explanations). After decomposition, each process is synthesized individually, with the goal of building a correct overall system after combination with the other processes. The crucial point is that local processes *assume* certain behavior over the other processes that suffices to synthesize a local solution. Finding these assumptions is critical: They must be satisfiable by the other processes and must suffice for the process to be realizable. After the local synthesis, the obtained local implementations are composed to produce the overall system. When applicable, this decomposition alleviates the state-explosion inherent in distributed synthesis.

Our key contribution is the introduction of *information-flow assumptions*, in contrast to the behavioral assumptions used in the literature. Information-flow assumptions are hyperproperties: They constrain relations between traces by requiring that certain differences in the environment become observable to particular processes. We formalize this by identifying indistinguishable trace pairs for a process and by requiring that whenever the goal formula demands an observable difference, the process can distinguish the corresponding traces.

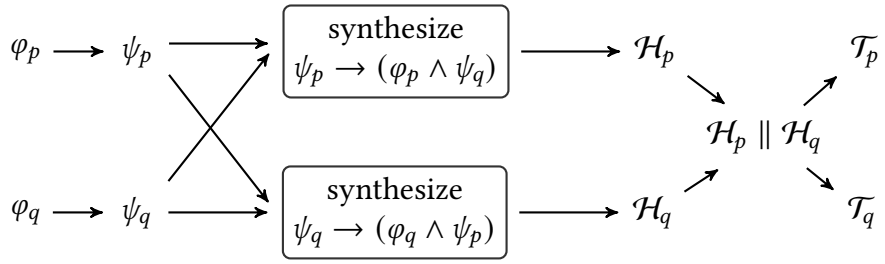


Figure 3.1 An overview of the compositional synthesis algorithm with information-flow assumptions for processes p and q. The first step consists of computing the information-flow assumption for each process. Afterward, the assumptions are exchanged and hyper implementations are synthesized that assume their own assumption and guarantee the local specification and the assumption of the other component. A final composition and decomposition constructs the correct local implementations.

The approach builds on ideas from epistemic reasoning: An agent "knows" a fact when it holds on all traces indistinguishable to that agent. We turn this insight into synthesis assumptions by requiring that, whenever two traces must be distinguishable for correctness, the implementation will make them distinguishable. Such an assumption can be formulated as follows:

$$\psi = \forall\pi\forall\pi'.\,\square\left(\left(in_{\mathsf{a}}^{\pi} \nleftrightarrow in_{\mathsf{a}}^{\pi'}\right) \rightarrow \left(c_{\mathsf{a}}^{\pi} \nleftrightarrow c_{\mathsf{a}}^{\pi'}\right)\right)$$

It specifies that on all trace pairs and at every point in time, whenever in_{a} is different, there must be an observable difference on the communication variables c_{a}, i.e., their values must be different in the same time step. This type of assumption leaves open the encoding used by a sender: It specifies that a distinction will be observable, not how it is encoded. That gap complicates the construction of a concrete, executable implementation for a receiver, because decoding depends on the sender's chosen encoding.

To resolve this, we synthesize hyper implementations: Prototype components that have access to the original (unencoded) information. A hyper implementation can always produce the outputs required to satisfy the specification; the information-flow assumption guarantees that a concrete implementation can later reconstruct the same information from the communication actually received. Thus, hyper implementations serve as correct blueprints that are independent of a sender's internal encoding, while the compositional synthesis ensures that a realizable decoding exists whenever the assumption holds. Figure 3.1 shows a detailed overview of our compositional synthesis approach, separated into the individual computation steps.

Concretely, we introduce necessary information-flow assumptions for two-process systems and provide automatic derivation methods from LTL specifications. The chapter is split into two parts, one for liveness specifications and one for safety specifications. For liveness properties, we strengthen the information-flow assumptions to time-bounded information-flow assumptions, where information must arrive within an arbitrary bound. For safety properties, we use prefix information-flow assumptions that require information-flow at certain fixed time points. We provide synthesis procedures for hyper implementations and composition techniques that yield correct distributed systems for both scenarios. Each approach is accompanied by experimental results and a prototype implementation that demonstrate the practicality and scalability of the proposed compositional synthesis approach.

Outline. We give an introduction to information-flow assumptions in Section 3.1. A refined version for liveness specifications is shown in Section 3.2 with the corresponding synthesis algorithm in Section 3.3. We refine information-flow assumptions for safety specifications in Section 3.4 and provide a synthesis algorithm for such in Section 3.5.

Publications. Section 3.2 and Section 3.3 build on [97]; for this thesis, we have expanded the presentation and included the previously omitted proofs. Section 3.4 and Section 3.5 correspond to [98] and here contain full proofs as well as additional algorithms. The more general treatment of information-flow assumptions in Section 3.1 is new material introduced in this thesis.

[97] Bernd Finkbeiner, Niklas Metzger, Yoram Moses. **Information Flow Guided Synthesis**. *34th International Conference on Computer-Aided Verification (CAV 2022)*

[98] Bernd Finkbeiner, Niklas Metzger, Yoram Moses. **Information Flow Guided Synthesis with Unbounded Communication**. *36th International Conference on Computer-Aided Verification (CAV 2024)*

3.1 Information Flow in Distributed Systems

In this section, we introduce the concept of information flow in distributed systems. In such systems, processes must communicate locally available information to other components to ensure correct behavior. We formalize information flow for linear temporal logic specifications using the concept of distinguishability, which characterizes when an agent must differentiate between specific situations.

3.1.1 Trace Property Assumptions

Assume-guarantee synthesis for distributed systems [6, 37, 54, 169] addresses the challenge of decomposing a distributed synthesis problem into sub-specifications. These sub-specifications serve as assumptions and guarantees for monolithic (single-process) synthesis. If each process satisfies its respective sub-specification, the overall distributed system is correct. Current methods for constructing such assumptions and guarantees typically employ negotiation algorithms [169] among processes to derive *sufficient* properties. After several negotiation rounds, the resulting trace properties are used in monolithic synthesis to generate correct local implementations, provided the assumptions are sufficient. We give an example for assume guarantee synthesis with LTL formulas in the following:

Example 12. We recall Example 5. The bit transmission implementation shown in Figure 1.2 is a possible result of a distributed synthesis algorithm that is given the architecture in Figure 1.1 and the LTL specification $\varphi = \square(in_{\mathsf{a}} \leftrightarrow \bigcirc out_{\mathsf{b}})$. Note that there are infinitely many implementations that can be synthesized for φ: Process a can, at every point in time, either copy the value of in_{a} to c_{a}, or copy the negation of the value of in_{a} to c_{a}. A possible *decomposition* of φ into assumptions is the following:

$$\varphi_{\mathsf{a}} = \square(in_{\mathsf{a}} \leftrightarrow c_{\mathsf{a}}) \qquad\qquad \varphi_{\mathsf{b}} = \square(c_{\mathsf{a}} \leftrightarrow out_{\mathsf{b}})$$

The individual monolithic synthesis tasks for φ_{a} and φ_{b} build a solution for the distributed synthesis problem, exactly the one in Figure 1.2. However, the assumption φ_{a} is not necessary: For all implementations of a where, at some point in time, the dual of the value of in_{a} is copied to c_{a}, the assumption is not suitable for constructing a correct implementation of b.

Assumptions formulated as trace properties inherently restrict the solution space for component implementations, which is one drawback of trace property assume-guarantee synthesis. This limitation necessitates negotiation procedures to compute viable solutions. We formalize this observation in the following proposition:

Proposition 1. *There exist LTL specifications for which no trace property assumption is necessary and sufficient. Furthermore, the number of possible trace property assumptions is unbounded.*

Proof. We show Proposition 1 with the previously introduced example property $in_{\mathsf{a}} \leftrightarrow \Diamond out_{\mathsf{b}}$ for process b in the architecture presented in Figure 1.1. Let $\varphi = in_{\mathsf{a}} \leftrightarrow$ → p. 6

$\bigcirc^n c_{\mathsf{a}}$ be the parameterized assumption for a. It is easy to see that we can construct infinitely many correct assumptions with φ by setting $n \in \mathbb{N}$ to infinitely many specific values. It remains to show that there is no necessary trace property assumption. Recall that a necessary property must be satisfied by every implementation that realizes the specification. Assume there exists a necessary and sufficient property. Then, for every n it must contain $\varphi = in_{\mathsf{a}} \leftrightarrow \bigcirc^n c_{\mathsf{a}}$ and $\varphi' = in_{\mathsf{a}} \nleftrightarrow \bigcirc^n c_{\mathsf{a}}$, as both specifications represent sufficient solutions. A necessary trace property, therefore, must contain the traces for both specifications. The inductive reasoning over n immediately shows that this property must contain all possible traces - the previous reasoning step can be done for every n. The resulting necessary assumption is `true` and is not sufficient. □

Therefore, we define an alternative notion of assumptions based on information flow: Local components must be able to obtain the information required to respond correctly to environment outputs. We specify this in the following with the concept of *distinguishability*.

3.1.2 Distinguishability

System correctness is typically defined with respect to behavioral properties, which specify sets of input and output variables. A system is considered correct if all its produced traces are contained within the specified behavioral property. In monolithic reactive systems, this reduces to determining the correct output for each finite input sequence. In distributed systems, however, each process has only a partial view: Some inputs are observable, while others are hidden and must be communicated by other processes. Thus, correctness involves two key aspects: Ensuring that each process receives the necessary information and guaranteeing that it responds appropriately. The given specification fundamentally determines both aspects.

We begin by examining how information is transmitted between components. A key aspect is determining, for each pair of environment traces, whether a process must produce different outputs in order to satisfy the specification. For trace properties that describe the desired system behavior, a process must react differently to traces whenever producing the same output would violate the property. This requirement is captured by *distinguishability*: Two traces are distinguishable for a process if no implementation exists that can produce the same output sequence for both traces while still satisfying the specification [81]. Formally, we define the distinguishability relation over pairs of environment traces as follows:

Definition 8 (Trace distinguishability)**.** *Let φ_p be an LTL specification of process p. The corresponding* trace distinguishability relation *is defined as*

$$v_{p,\varphi_p} = \{(t_e, t'_e) \in (2^{O_e})^\omega \times (2^{O_e})^\omega \mid \forall t_p \in (2^{O_p})^\omega.\, t_e \sqcup t_p \not\models \varphi_p \text{ or } t'_e \sqcup t_p \not\models \varphi_p\}$$

The trace distinguishability v is parameterized in a component p which must receive the information, and a specification φ that defines the correctness of the system. In more detail, exactly those *infinite* environment output traces are related that have no shared reaction on process p, namely an infinite trace over p's outputs that satisfies the specification φ for both traces. It is worth noting that this definition neither enforces a specific point in time on one of the traces nor the violation or satisfaction of the formula on an explicit trace.

Example 13. Consider the specification of the running example $\varphi = in_\mathsf{b} \leftrightarrow \Diamond out_\mathsf{b}$, where in_b is unobservable for p and out_b is controlled by p. To satisfy the specification, p must distinguish trace pairs t, t' on which the value of in_b is different in the first step. More concretely, the distinguishability relation corresponds to the following, more simplified, set:

$$v_{p,\varphi_p} = \{(t_e, t'_e) \in (2^{\{in_\mathsf{b}\}})^\omega \times (2^{\{in_\mathsf{b}\}})^\omega \mid t_e[0] \neq t'_e[0]\}$$

The relation is symmetric, meaning that for every t_e that must be distinguished from t'_e, it also holds that t'_e must also be distinguished from t_e.

The (infinite) distinguishability relation defines a precise characterization of trace pairs to be distinguished by some component. The definition is based on a *global view* of the system, i.e., it accesses the environment outputs that are unobservable for the component p. In the following, we transform this relation into an assumption on the *behavior of other components* in the system that is observed by p. This yields the first assumption on information flow that can be used for assume-guarantee synthesis.

3.1.3 Information Flow Assumptions

The literature on information flow is usually focused on the absence of information, e.g., in [192]. The arguably most famous property to be verified is *noninterference* [122], stating that a secret cannot be inferred from the observable outputs. One variant of noninterference is *observational determinism* [213]. Observational determinism requires that, whenever the low security inputs to a (one-component) system are equal, then

the low security outputs must also be equal. The values of the high security inputs can, therefore, not be inferred. A HyperLTL formula that states observational determinism is the following

$$\psi = \forall\pi\forall\pi'.\,\square(\pi|_{I_L} = \pi'|_{I_L}) \rightarrow \square(\pi|_{O_L} = \pi'|_{O_L}),$$

where I_L and O_L are subset of the variables $\mathcal{V}$ for the low security inputs and outputs, respectively. For any two traces, it must hold that if they are equal when projected to the low security inputs, then they must also be equal when projected to the low security outputs. If this property is satisfied, then there exists no *information flow* from the variables in I_L to the variables in O_L.

We turn this argument of the absence of information flow to *presence* of information flow. Conceptually, the property we consider is equivalent to observational determinism: Given two relations R and R' over tuples of traces, we enforce that, if two traces are related by R, then they are also related by R'. For observational determinism, R captures the equivalence on the low security inputs, and R' captures the equivalence on low security outputs. In the general case, we build information flow properties as 2-hyperproperties defined by a relation. This resembles the structure of the HyperLTL formula for observational determinism shown above.

Definition 9 (Information flow)**.** *Let R and R' be relations on tuples of traces in $(2^{\mathcal{V}})^{\omega}$. Information flow is the 2-hyperproperty defined by the following relation*

$$R_{\mathcal{F}} = \{(t, t') \in (2^{\mathcal{V}})^{\omega} \times (2^{\mathcal{V}})^{\omega} \mid \textit{ if } (t, t') \in R \textit{ then } (t, t') \in R'\}$$

Recall that a 2-hyperproperty defined by a relation over pairs of traces is all trace sets such that the pairs of traces in the trace set are elements of the relation. Informally, Definition 9 imposes that there is information flow from R to R' in the sense that all trace pairs that are in R will also be in R'. A process that has access only to R' can ensure that two traces *are not related* by R by checking if they are not related by R'. We now instantiate the relations so that the information flow property can be used as an assumption for a process's observed information.

A viable assumption for the assume-guarantee synthesis of local component p should state that process p will be able to distinguish two traces t_e and t'_e if they are related by $\nu_{p,\varphi}$. All those traces are guaranteed not to have a single shared reaction by p. The component can then correctly assume that it will be able to distinguish the traces based on some *local inputs* I_p. The domain of the variables used to specify the assumption must be different from the one for the distinguishability: While the

distinguishability only refers to *all* environment outputs and the outputs of the process p, the assumption has to be specified according to the observable variables of p. Note that p's observable variables are all its input variables, i.e., the set I_p. We now define the trace information-flow assumption for a given trace distinguishability relation.

Definition 10 (Trace information-flow assumption)**.** *Let $\nu_{p,\varphi}$ be the trace distinguishability relation for p. The* trace information-flow assumption $\mathcal{F}_{p,\varphi}$ *is the 2-hyperproperty induced by the relation*

$$R_{\mathcal{F}_{p,\varphi}} = \{(t, t') \in (2^{\mathcal{V}})^\omega \times (2^{\mathcal{V}})^\omega \mid \textit{if } (t|_{O_e}, t'|_{O_e}) \in \nu_{p,\varphi} \textit{ then } t|_{I_p} \neq t'|_{I_p}\}$$

Recall that $t \neq t'$ means that there is some time step i such that $t[i] \neq t'[i]$. As noted in the definition, the trace information-flow assumption defines a *hyperproperty*, which is a set of sets of traces. Remark that relation $R \subseteq (2^{\mathcal{V}})^\omega \times (2^{\mathcal{V}})^\omega$ of pairs of traces defines the hyperproperty H, where a set $\mathbb{T}$ of traces is an element of H iff for all pairs $t, t' \in \mathbb{T}$ it holds that $(t, t') \in R$. It specifies all sets of traces over all variables ($\mathcal{V}$) that have different variables on p's inputs (I_p) on every two traces that are related by the distinguishability relation.

Example 14. We specify the trace information-flow assumption of the distinguishability relation presented in Example 13 as a HyperLTL formula. There are two parts of the formula: Identifying that two traces are related and, under this condition, that a difference between these traces is observable for process p. → p. 39

$$\forall\pi\forall\pi'.(in_{\mathsf{b}}^{\pi} \leftrightarrow in_{\mathsf{b}}^{\pi'}) \rightarrow \Diamond(c_{\mathsf{b}}^{\pi} \leftrightarrow c_{\mathsf{b}}^{\pi'})$$

The premise of the implication specifies when two traces are related by $\nu_{p,\varphi}$, namely if the input bit is different in the first step. If this is the case, then there must be a p-observable difference between the communication bits c_{b} at some point ($\Diamond$) on the traces π and π'.

The trace information-flow assumption can, in general, be used for assume-guarantee synthesis. There are, however, two crucial questions regarding assumptions: (1) Does the assumption rule out viable solutions during the synthesis process, and (2) is the assumption strong enough for a successful synthesis procedure? Specified in formal terms, (1) corresponds to an assumption being *necessary* and (2) to an assumption being *sufficient*. To formally reason about these properties, we lift the definition of assumptions (see Definition 6) to hyperproperties. → p. 33

Definition 11 (Hyperproperty assumptions)**.** *Let P be a trace property and H be a hyperproperty. H is a sufficient assumption if $\forall A \in H$ it holds that $A \rightarrow P$ is realizable. H is a necessary assumption if for all sufficient trace property assumptions B and for all $A \in H$ it holds that $B \rightarrow A$.*

A hyperproperty assumption is therefore sufficient if all trace sets in the hyperproperty are sufficient to realize the problem. It is necessary that all trace sets in the hyperproperty are subsumed by all sufficient assumptions. We now show that trace information-flow assumptions are necessary and not sufficient.

Theorem 1. *The trace information-flow assumption is necessary.*

Proof. We show Theorem 1 by contradiction. Assume that there exists an implementation (f_p, f_q) that satisfies φ_p but not $\mathcal{F}_{p,\varphi}$. Since $\mathcal{F}_{p,\varphi}$ is not satisfied, there exists a pair of traces t, t' such that $(t|_{O_e}, t'|_{O_e}) \in \Delta_p$ and $t|_{I_p} = t'|_{I_p}$. Let $t_e = t|_{O_e}$, and $t'_e = t'|_{O_e}$. Since the inputs to process p are the same on t and t', and since the strategies f_p and f_q are deterministic, the sequence of outputs is also the same. Let $x_0x_1x_2 \ldots = t|_{I_p} = t'|_{I_p}$ be the sequence of inputs. We construct the sequence of outputs $o_0o_1o_2 \ldots$ generated by the implementation as follows: $o_k = f_p(x_0x_1 \ldots x_{k-1})$ for all $k \in \mathbb{N}$. Given that the implementation satisfies φ_p, we have that both $t_e \sqcup o$ and $t'_e \sqcup o$ satisfy φ_p. This, however, contradicts the assumption that $(t|_{O_e}, t'|_{O_e}) \in \Delta_p$. □

Proposition 2. *The trace information-flow assumption is not sufficient.*

We prove this Proposition by constructing an example that is realizable as an assume-guarantee synthesis problem, but for which the trace information-flow assumption does not suffice to find a solution.

Proof. Assume the architecture to be as shown in Figure 1.1 and the local goal specification of the transmitter to be $\varphi = in_{\mathsf{b}} \leftrightarrow \Diamond\, out_{\mathsf{b}}$. The trace information-flow assumption is as given in Example 14: → p. 6

$$\forall\pi\forall\pi'.(in_{\mathsf{b}}^{\pi} \nleftrightarrow in_{\mathsf{b}}^{\pi'}) \rightarrow \Diamond(c_{\mathsf{b}}^{\pi} \nleftrightarrow c_{\mathsf{b}}^{\pi'})$$

This assumption includes all implementations that set $\Box\, c_{\mathsf{b}}$ if in_{b} holds in the first step and $c_{\mathsf{b}} \wedge \bigcirc(c_{\mathsf{b}}\, \mathcal{U}\, \neg c_{\mathsf{b}})$ if in_{b} is *false* in the first step, where some explicit time step greater than 0 is chosen for $\neg c_{\mathsf{b}}$. For this set of implementations, the receiver can never set out_{b} to *true*: Whenever it decides to set the output to true in step n, to react correctly to the input in the first step, it might be the case that it will be told in step $n+1$ that in_{b} was false initially. This is viable behavior for process a because the assumption allows

the implementations that respond to in_b with $c_b \wedge \bigcirc(c_b \mathcal{U} \neg c_b)$. Therefore, the trace information-flow assumption is too weak to suffice for building an implementation. □

We observed that, although necessary, this general definition of information-flow assumptions does not suffice for the synthesis of many distributed systems. Instead, it often must be strengthened in a way similar to what is shown in Section 3.1.3, where the information flow must actively be observable on one trace and not the other. In Section 3.2 and Section 3.4, we introduce strengthened versions of the information-flow assumption that are suitable, in the sense of being effective for synthesis, for different fragments of LTL. However, we first establish the fundamental automata constructions for the distinguishability relation and the trace information-flow assumption. Those are the building blocks for the assume-guarantee synthesis approaches that follow.

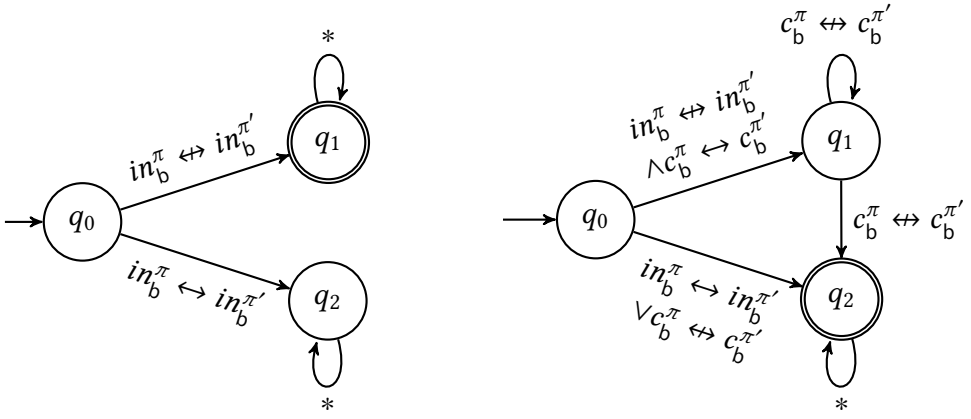
3.1.4 Computing Information Flow Assumptions

The analysis of information-flow properties on distributed systems builds on the comparison of infinite traces. Our model of choice for reasoning about relations of infinite traces is two-tape Büchi automata, whose input alphabets are pairs of letters of the LTL formulas' alphabet. We annotate the symbols of the alphabet according to the tape that they belong to. The annotation is in HyperLTL fashion with trace variable indexes. The automata accept all related pairs of traces, which the component must be able to distinguish.

Example 15. Figure 3.2a presents a Büchi automaton that accepts all pairs of traces that are related by the trace distinguishability relation for the specification $\varphi = in_b \leftrightarrow \Diamond in_b$. The automaton in Figure 3.2a correctly accepts all pairs of traces that differ in the first step. The Büchi automaton runs on a single trace with the alphabet $\Sigma = \{in_b^{\pi}, in_b^{\pi'}\}$ which consists of two copies of the original variable in_b, one for each trace. After consuming the first letter, the automaton either proceeds to accept or reject all suffixes. The automaton for the information-flow assumption is shown in Figure 3.2b. It accepts all pairs of traces that are different on c_b. If this automaton accepts all traces of a self-composition of a system, then this system satisfies the trace information-flow assumption.

We now show how to construct an automaton that accepts the distinguishability for specifications formalized as LTL formulas.

Theorem 2. *There exists a universal automaton accepting the distinguishability relation of an LTL formula φ with exponential size in the size of the LTL formula φ.*



(a) (b)

Figure 3.2 An NBA for the trace distinguishability relation on the running example (a) and the corresponding NBA that recognizes the (b) information-flow assumption.

Construction 1. We begin with an automaton $\mathcal{A}_{\neg\varphi}$ that accepts all traces that violate the LTL formula φ. We self-compose this automaton to build $\mathcal{B}$ over the alphabet $2^{O_e \cup O_p} \times 2^{O_e \cup O_p}$, which accepts all pairs of traces that each violate the formula. To restrict the automaton to pairs of traces that agree on the outputs, we intersect it with an automaton over the alphabet $2^{O_e \cup O_p} \times 2^{O_e \cup O_p}$ that accepts all pairs of traces that agree on the outputs, and obtain C. We combine C with a Büchi automaton over the alphabet $2^{O_e \cup O_p} \times 2^{O_e \cup O_p}$ that accepts all traces that have a difference in the outputs O_p at least once. The last operation must guarantee that only environment traces are accepted for which the choice of the output is irrelevant, i.e., whatever the process outputs, there will be a violation. We achieve this by universal projection of the O_p's and obtain the final universal automaton $\mathcal{D}$. The automaton $\mathcal{D}$ is exponential in the size of the LTL formula; the only blow-up is obtained in the construction of $\mathcal{A}_{\neg\varphi}$.

The distinguishability relation accepts all pairs of traces that a process must distinguish to implement the goal specification. The information-flow assumption adds to the concept of distinguishability: For every two such traces, there must be an observable difference in the process's inputs. We can construct a Büchi automaton that accepts the information-flow assumption.

Theorem 3. *There exists a 2-tape Büchi automaton accepting the trace information-flow assumption with exponential size in size of the LTL formula φ.*

Construction 2. We follow the structure of Definition 10: Either two traces must not be distinguished, or they will at some point be different on the observable inputs of the → p. 41

process. We first translate the LTL specification φ to a Büchi automaton $\mathcal{A}_\varphi$ over the alphabet $2^{O_e \cup O_p}$. This automaton accepts all traces that satisfy φ and has exponential size in the size of φ [208]. After selfcomposition of $\mathcal{A}_\varphi$, we obtain an automaton $\mathcal{B}$ over the alphabet $2^{O_e \cup O_p} \times 2^{O_e \cup O_p}$ that accepts all pairs of traces that satisfy φ. We restrict $\mathcal{B}$ to pairs of traces that are equivalent on the values of O_p and call it $\mathcal{C}$. To remove O_p from the alphabet of the automaton, we existentially project the values O_p on both traces and obtain $\mathcal{D}$ with alphabet $2^{O_e} \times 2^{O_e}$. Automaton $\mathcal{D}$ accepts all pairs of traces that are *not* in the trace distinguishability relation of φ. Recall that for all other traces, we enforce a difference on the process inputs at some time point. A separate automaton $\mathcal{E}$ over the alphabet $2^{I_p} \times 2^{I_p}$ accepts all pairs of traces that differ at some point. The final automaton $\mathcal{F}$ over the alphabet $2^{O_e \cup I_p} \times 2^{O_e \cup I_p}$ is the product of $\mathcal{D}$ and $\mathcal{E}$. The size of the automaton is exponential in the size of the LTL formula, as only the construction of $\mathcal{A}_\varphi$ yields an exponential blow-up; all other operations are sub-exponential.

3.2 Time-Bounded Information Flow

In this chapter, we present compositional synthesis using information-flow assumptions for specifications that do not impose an explicit deadline on message transmission. Such properties are common in distributed systems, including protocols like the alternating bit protocol [19], where messages must eventually be delivered but no fixed time bound is specified. We introduce a strengthened form of information-flow assumptions to enable their use in assume-guarantee synthesis, allowing the synthesis procedure to rely on the existence of some (possibly unknown) deadline for message reception. In the second part of this chapter, we describe a constructive assume-guarantee synthesis method for specifications that require only a finite amount of information transmission. Here, we instantiate the implicit deadline using ghost variables, which are later replaced by the actual deadlines and encoding strategies determined by the synthesized transmitter.

3.2.1 Time-bounded Information Flow Assumptions

The starting point for our analysis of asymmetric information flow is the example specification $\varphi = in_{\mathsf{a}} \leftrightarrow \Diamond out_{\mathsf{b}}$. It specifies that out_{b} must be set to true by process b iff the input to a in the first step was true. In the distributed system, process a is in charge of communicating its own input in_{a} to process b and, to satisfy the specification, a must eventually communicate the value of in_{a} to b. In Section 3.1.3, we already showed that the trace information-flow assumption *cannot* be sufficient in this scenario,

→ p. 42

simply because it can never be certain that the information of in_{a} being true would have already been communicated.

We solve this problem by strengthening the trace information-flow assumption in a way that process b can rely on a time point for the information transmission. The strengthening starts by refining distinguishability, which now relates two traces that must be distinguished in an ordered way, in contrast to the symmetric trace distinguishability in Section 3.1.2. The time-bounded distinguishability, as shown below, relates two traces if every correct output sequence on the first trace yields a *finite* violation on the second trace.

→ p. 38

Definition 12 (Time-bounded distinguishability)**.** *Given a local specification φ_p for process p, the* time-bounded distinguishability relation Λ_p *is the set of trace pairs $(t_e, t_e') \in (2^{O_e})^\omega \times (2^{O_e})^\omega$ of environment outputs such that every trace of local outputs $t_p \in (2^{O_p})^\omega$ either violates the specification φ_p when combined with t_e, or* finitely *violates φ_p when combined with t_e':*

$$\Lambda_p = \{(t_e, t_e') \in (2^{O_e})^\omega \times (2^{O_e})^\omega \mid \forall t_p \in (2^{O_p})^\omega.\ \textit{if}\ t_e \sqcup t_p \vDash \varphi_p\ \ \textit{then}\ \ t_e' \sqcup t_p \nvDash_f \varphi_p\ \}$$

Recall that the $t \nvDash_f \varphi$ means that there exists a finite prefix v of t such that every infinite extension of ω violates the formula φ.

Example 16. Consider again the specification $\varphi = in_{\mathsf{a}} \leftrightarrow \Diamond out_{\mathsf{b}}$ for process b in the architecture of Figure 1.1 and the two traces $t = in_{\mathsf{a}} in_{\mathsf{a}} in_{\mathsf{a}} \ldots$ and $t' = \neg in_{\mathsf{a}} in_{\mathsf{a}} in_{\mathsf{a}} \ldots$. It holds that (t, t') and (t', t) are in the trace distinguishability relation (see Definition 8). The time-bounded distinguishability relation, however, only contains the pair (t, t'), because every correct output sequence for t will set out_{b} to *true* at some time point, which immediately (and unrecoverably) violates the specification for t'. Conversely, the only correct output sequence, which is $(\neg out_{\mathsf{b}})^\omega$, is not finitely violating the specification for t. Every finite prefix of $\neg out_{\mathsf{b}}$'s can be repaired by setting out_{b} to *true* in the successor time step.

→ p. 6
→ p. 38

We have imposed an ordering on the trace pairs within the distinguishability relation. This distinguishes between traces where a process must actively commit to reacting to information, and traces for which there exists an output sequence such that the process does not need to change its output once the information is revealed. Consequently, the time-bounded information-flow assumption is asymmetric: It requires that the necessary information for a process is revealed on the first trace of the pair, while the second trace does not require an explicit observation of this information.

Assume there exists a function $n : (2^{O_e})^\omega \to \mathbb{N}$ that, for each environment output trace t_e, determines the time point $f(t_e)$ at which the process p can distinguish t from t'. The function n is used in the definition of the time-bounded information-flow assumption, and, since we do not know the function in advance, the assumption is the infinite union of all 2-hyperproperties for every possible function n.

Definition 13 (Time-bounded information-flow assumption)**.** *Given the time-bounded distinguishability relation Λ_p for process p, the* time-bounded information-flow assumption *χ_p for p is the (infinite) union over the 2-hyperproperties induced by the following relations R_f, for all possible functions $n : (2^{O_e})^\omega \to \mathbb{N}$:*

$$R_n = \{(t, t') \in (2^{\mathcal{V}})^\omega \times (2^{\mathcal{V}})^\omega \mid \\ \textit{if } (t|_{O_e}, t'|_{O_e}) \in \Lambda_p, \textit{ then } t[0..n(t|_{O_e})]|_{I_p} \neq t'[0..n(t|_{O_e})]|_{I_p}\}$$

Information flow in this context is asymmetric, determined by the time-bounded distinguishability relation. Additionally, we require that the difference between the two traces becomes observable on the process inputs precisely at the time point indicated by n. Any implementation of a process responsible for transmitting the information will realize such a function n.

Proposition 3. *The time-bounded information-flow assumption is not necessary.*

In contrast to the trace information-flow assumption, the time-bounded information-flow assumption is not necessary. We give an example specification that shows the claim in Proposition 3.

Example 17. The specification we present is realizable and does not satisfy the time-bounded information-flow assumption. We show that there exists no function n that only depends on t for the pair (t, t') and can output the time point when t and t' must be distinguished. Consider an abbreviation of the example specification in Example 16 where a new environment output $\mathit{free}_{\mathsf{a}}$ is introduced. This output is only visible to process a and the specification is twofold: If in_{a} is *true* in the first step, then $\mathit{out}_{\mathsf{b}}$ must be true two steps after $\mathit{free}_{\mathsf{a}}$ is true for the first time. This is the environment behavior for t. In the opposite case where in_{a} is *false* in the first step, b must set $\mathit{out}_{\mathsf{b}}$ to *false* two steps after $\mathit{free}_{\mathsf{a}}$ became true, which is the case for t'. This specification is realizable: a sets c_{a} to *false* initially. Once $\mathit{free}_{\mathsf{a}}$ is observed to be *true*, then a sets c_{a} to *true* iff in_{a} is *true*. Process b starts by setting $\mathit{out}_{\mathsf{b}}$ to *false* and waits for c_{a} to become *true*, in which case it switches $\mathit{out}_{\mathsf{b}}$ to *true*. In both cases, the correct output is set by b two steps after $\mathit{free}_{\mathsf{a}}$ is set to *true* by the environment. Clearly, both environment output traces

must be distinguished to satisfy the specification and every pair of traces on which the first value of in_{a} is different will have an unrecoverable violation on the second trace for every correct output sequence on the first trace. Therefore, they are related by the time-bounded distinguishability relation. However, if we analyze the time-bounded information-flow assumption for these traces, no function n exists that can provide the time point when t and t' can be distinguished - this depends on the time point of $free_{\mathsf{a}}$ becoming true on the trace t', and n can only depend on t. The specification is realizable but does not satisfy the time-bounded information-flow assumption, which is a counterexample to the time-bounded information-flow assumption being necessary.

The necessity of information-flow assumptions is rooted in their role as necessary conditions for compositional synthesis. We therefore identify specifications where this necessity persists. Informally, the assumption remains necessary if the finite violation in the time-bounded distinguishability relation occurs within a bounded number of steps. We refer to such distinguishability relations as *uniform*, and formalize this notion as follows:

Definition 14 (Uniform distinguishability)**.** *A time-bounded distinguishability relation* Λ_p *is* uniform *if for every trace* $t_e \in (2^{O_e})^\omega$ *of environment outputs, and every trace* $t_p \in (2^{O_p})^\omega$ *of outputs of* p*, there exists a natural number* $m \in \mathbb{N}$ *such that for all* $t'_e \in (2^{O_e})^\omega$ *such that* $(t_e, t'_e) \in \Lambda_p$ *if* $t_e \sqcup t_p \vDash \varphi_p$ *then* $t'_e \sqcup t_p \nvDash_m \varphi_p$*.*

The specification $\varphi = in_{\mathsf{a}} \leftrightarrow \Diamond out_{\mathsf{b}}$ infers a uniform distinguishability relation, and the time-bounded information-flow assumption built on it is a necessary assumption. We generalize this result to all uniform distinguishability relations.

Theorem 4. *Let* Λ_p *be a uniform time-bounded distinguishability relation derived from process* p*'s local specification* φ_p*. Every computation tree that satisfies* φ_p *also satisfies the time-bounded information-flow assumption* χ_p*.*

Proof. Let (f_a, f_b) be an implementation that satisfies φ_p. We show that the time-bounded information-flow assumption χ_p is satisfied by defining a function $n : (2^{O_e})^\omega \to \mathbb{N}$ such that the 2-hyperproperty given by R_n is satisfied. To compute $n(t'_e)$ for some trace of inputs $t'_e \in (2^{O_e})^\omega$, we consider the trace of outputs $t'_p \in (2^{O_p})^\omega$ obtained by applying the implementation to the prefixes of t'_e. Since Λ_p is uniform, there is a natural number $m \in \mathbb{N}$ such that for all t_e with $(t_e, t'_e) \in \Lambda_p$, we have that $t'_e \sqcup t_p \nvDash_m \varphi_p$. We set $n(t_e)$ to m.

To convince yourself that χ_p is satisfied, suppose, by contradiction, that R_n is violated on some pair $(t_e, t'_e) \in \Lambda_p$ of input traces, i.e., the projection on I_p is the same for t_e and

t'_e on the entire prefix of length $n(t'_e)$. But then, also the output of process p must be the same along the prefix; this, however, means that t'_e will violate φ_p after $m = n(t'_e)$ steps. This contradicts our assumption that the implementation satisfies φ_p. □

3.2.2 Automata for Asymmetric Information Flow

Automata-based computation of information-flow assumptions is essential for compositional synthesis in distributed systems. To enable synthesis algorithms that rely on assumptions and guarantees, we present constructions for Büchi automata that recognize the trace information-flow assumption, the time-bounded distinguishability relation, and the time-bounded information-flow assumption. Additionally, we describe how to verify the uniformity of a time-bounded distinguishability relation using automata emptiness checks.

We begin with the construction of an automaton that accepts the time-bounded distinguishability relation of an LTL formula. This automaton is, as in Construction 1, a two-tape automaton that runs on the alphabet $2^{O_e} \times 2^{O_e}$. It accepts all pairs of traces that are related by the time-bounded distinguishability relation.

Theorem 5. *There exists a universal two-tape automaton accepting the time-bounded distinguishability relation of an LTL formula φ with doubly-exponential size in the size of the LTL formula φ.*

Construction 3. We follow the *if x then b* structure of Definition 12 for building the automaton and change it to *¬x or b*. The starting point is a deterministic ω-automaton $\mathcal{A}_{\neg\varphi_p}$ over the alphabet $2^{O_e \cup O_p}$ that accepts all traces that violate φ_p. This can be done by standard LTL to non-deterministic Büchi automata translations [207] and a following determinization [195]. The second automaton is the deterministic finite-word automaton $\mathcal{B}_{\neg\varphi_p}$ over the alphabet $2^{O_e \cup O_p}$ accepting all bad prefixes of φ obtained by the standard monitoring construction [67]. We combine both automata into a deterministic ω-automaton C over the alphabet $2^{O_e} \times 2^{O_e} \times 2^{O_p}$ that accepts triples of inputs t_e, t'_e, t_p such that $t_e \sqcup t_p$ violates φ_p or $t'_e \sqcup t_e$ *finitely* violates φ_p. The automaton operation to achieve C builds the product with an automaton that accepts all traces 2^{O_p} but restricts the branching structure to follow either $\mathcal{A}_{\neg\varphi_p}$ or $\mathcal{B}_{\neg\varphi_p}$. In the last step of the construction, we have to eliminate the third input trace, namely the output of the process. We universally project the trace t_p from the automaton C and obtain the universal automaton $\mathcal{D}_{\Lambda_p}$ with alphabet $2^{O_e} \times 2^{O_e}$. Constructing $\mathcal{A}_{\neg\varphi_p}$ and $\mathcal{B}_{\neg\varphi_p}$ is both doubly-exponential in the size of φ, all other constructions are at most polynomial. The final size of $\mathcal{D}_{\Lambda_p}$ is doubly-exponential in the size of φ_p.

→ p. 46

Before giving an explicit construction for time-bounded information-flow assumptions, we show how to check the uniformity of a time-bounded distinguishability relation. Intuitively, those are the specifications where the time point of the finite violation of the second trace of a trace pair can arbitrarily be pushed to the future (by the environment). The following theorem provides a lower bound for this check.

Theorem 6. *Whether a time-bounded distinguishability relation for an LTL formula φ is uniform is checkable in quadruply-exponential time in the size of φ.*

Proof. We construct an automaton that is non-empty if a trace exists that has no uniform bound. It contains all traces of environment outputs and process outputs that have no such bounds. Let $\mathcal{A}_{\varphi_p}$ be a universal ω-automaton over alphabet $2^{O_e \cup O_p}$ that recognizes all traces that satisfy the local specification φ_p. This automaton is exponential in the size of φ_p. We build another universal automaton $\mathcal{E}$ over the alphabet $2^{O_e} \times 2^{O_e} \times 2^{O_p}$ by combining $\mathcal{A}_{\varphi_p}$ and the automaton for the time-bounded distinguishability relation $\mathcal{D}_{\Lambda_p}$. This automaton accepts all t_e, t'_e, t_p if $(t_e, t'_e) \in \Lambda_p$ and $t_e \sqcup t_p \vDash \varphi_p$. The first condition is embedded in the automaton for the time-bounded distinguishability relation, and the second condition is by the automaton $\mathcal{A}_{\varphi_p}$. Automaton $\mathcal{E}$ adds no exponential blow-up. In a third step, we replace t'_e with a fresh variable n that becomes true at some time point. The automaton $\mathcal{F}$ accepts all words over $2^{O_e} \times 2^{O_p} \times 2^n$ such that there exists a t'_e for a given t_e, t_p pair in $\mathcal{E}$ such that t_e has a bad prefix after n becomes true for the first time. This yields another exponent because of the projection of t'_e. n now tells us the latest time point for a finite violation of the second trace. Finally, we universally project n from the alphabet of $\mathcal{F}$ to obtain the universal automaton $\mathcal{G}$ over the alphabet $2^{O_e} \times 2^{O_p}$. This automaton contains all traces t_e, t_p that have infinitely many time points for n being true for the first time, which means that there is no bound on the finite violation. The specification φ has a uniform time-bounded distinguishability relation if $\mathcal{G}$ is empty, and we can check this in quadruply-exponential runtime. The last exponent originates in the emptiness check of a universal automaton. □

The goal of this section is to compute the time-bounded information-flow assumption. Compared to the trace information-flow assumption in Section 3.1.3, this is fairly → p. 39
tricky. The reason is that the time-bounded information-flow assumption is not a k-hyperproperty for any value $k \in \mathbb{N}$. We explain this with the following proposition:

Proposition 4. *The time-bounded information-flow assumption is not a k-hyperproperty for any $k \in \mathbb{N}$.*

Intuitively, the assumption requires that an unbounded number of traces must be distinguishable *at the same point in time.* Consider an arbitrary chosen pair t, t_1 s.t.

$(t, t_1) \in \Lambda_p$. We know that t_1 must at some point be different from t on the observable inputs of process p. Let's further assume that there are $t_2, \ldots, t_k$ traces that all are related to t, i.e., $(t, t_i) \in \Lambda_p$. The time-bounded assumption now enforces that there is one time point for all t_i s.t. when compared to t, there is an observable difference on p's inputs. For any k traces, this leads to a k-hyperproperty since the time point must be equivalent for all traces t_i. The crucial observation is that the number of t_i traces is unbounded, as the difference in the traces can happen arbitrarily late. This yields arbitrarily many traces to distinguish from. Therefore, the time-bounded information-flow assumption is not a k-hyperproperty.

We circumvent this issue by introducing a fresh atomic proposition n that is to be read by process p as a new input and is to be computed by process p's environment. The shift of responsibility for observing when the information on a trace is present reduces the assumption to a 2-hyperproperty that can be computed from the time-bounded distinguishability relation.

Theorem 7. *For a process p with local specification φ_p, there exists a non-deterministic two-tape automaton with a doubly-exponential number of states in the size of φ_p that recognizes the time-bounded information-flow assumption χ_p induced by φ_p.*

Construction 4. We construct the automaton by following the same principles as in Construction 2, either a pair of traces must not be distinguished, or we do so in a time-bounded manner. The first step in the construction is the automaton $\mathcal{A}_{\Lambda_p}$ over alphabet $2^{O_e} \times 2^{O_e}$ that accepts all pairs of traces that must be distinguished time-boundedly by process p. We complement this universal automaton to $\overline{\mathcal{A}_{\Lambda_p}}$, which is a non-deterministic automaton that accepts all pairs of traces that are not related by Λ_p. This operation is linear since the dual of a universal automaton is a non-deterministic automaton with the same number of states [125]. We now check that on all traces that are not accepted by $\overline{\mathcal{A}_{\Lambda_p}}$, there is a difference in the inputs by the time n is observed to be *true*. This can be done with an automaton $\mathcal{H}_1$ that checks that n occurs on the left trace, combined with a second automaton $\mathcal{H}_2$ that checks that the traces differ before n occurs on the first trace. The final automaton is the disjunctive connection between $\overline{\mathcal{A}_{\Lambda_p}}$ and the two automata $\mathcal{H}_1$ and $\mathcal{H}_2$. The size of the automaton is doubly-exponential in the size of φ_p as no automaton operation increases the blow-up, and the final automaton preserves the double-exponential size of $\mathcal{A}_{\Lambda_p}$ in the size of φ.

The automaton that accepts the time-bounded information-flow assumption has the alphabet $2^{O_e \cup O_p \cup \{n\}} \times 2^{O_e \cup O_p \cup \{n\}}$ and accepts trace pairs over all variables plus the additional variable n. During compositional synthesis, the currently environmentally

controlled variable n will be replaced by the actual time point at which the information is observed by some process.

3.3 Distributed Synthesis with Time-Bounded Information Flow

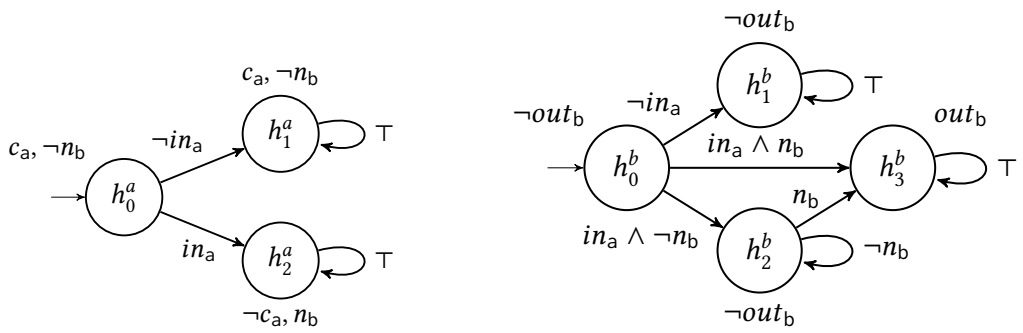
We now present a compositional synthesis approach for distributed systems based on time-bounded information-flow assumptions. Unlike traditional methods that use trace property assumptions, where the synthesis algorithm constructs a solution that satisfies a fixed trace property, our technique employs hyperproperty assumptions, which specify sets of sets of traces. This means the synthesis procedure cannot simply focus on a single trace set, but must account for the possibility that any one of the sets defined by the hyperproperty could be implemented by the sender of information. To address this, we introduce the concept of *hyper implementations*: Implementations synthesized with respect to hyperproperty assumptions.

3.3.1 Hyper Implementations for Time-bounded Assumptions

A hyper implementation is produced by synthesizing a system with respect to a hyperproperty assumption. For time-bounded information-flow assumptions, this assumption is computed as the initial step of the synthesis procedure. The synthesis process proceeds by first constructing an intermediate result—the local synthesis outcome. In assume-guarantee synthesis for processes p and q, the synthesis tasks are decomposed into $\psi_p \rightarrow (\psi_q \wedge \varphi_p)$ and $\psi_q \rightarrow (\psi_p \wedge \varphi_q)$. The synthesized system for process p responds to all environment outputs, its own inputs, and the special variable n_p provided by the time-bounded information-flow assumption. The outputs consist of the local outputs and the special variable n_q, which indicates the time point at which traces become distinguishable for q. This implementation is referred to as a hyper implementation.

Definition 15 (Time-bounded hyper implementation)**.** *Let p and q be processes and e be the environment. A $2^{O_e \cup I_p \cup \{n_p\}}$-branching $2^{O_p \cup \{n_q\}}$-labeled tree h_p is p's* hyper implementation.

We reason about hyper implementations as infinite trees that branch according to three types of variables: The environment outputs, the process inputs, and the special variable n_p. Most importantly, the hyper implementation reads in *unobservable* outputs of the environment. At this stage of the synthesis process, these unobservable



(a) The hyper implementation of a

(b) The hyper implementation of b

Figure 3.3 The hyper implementations of the *bit transmission* protocol. The bit transmission protocol specifies that out_b must eventually be true iff in_a is true in the first step. The hyper implementation of a in Figure 3.3a sets c_a to *false* and n to *true* in case in_a is *true*. The hyper implementation of b in Figure 3.3b sets out_b to *false* and waits for c_a and n to become *true*.

outputs must be artificially presented to the process to obtain a realizable local synthesis task. The labels of the tree are the process outputs in combination with the special variable of the other process, namely n_q. In the following example, we show a hyper implementation as a finite-state abstraction of the corresponding tree.

Example 18. We showcase the concept of hyper implementations with the bit transmission protocol in Figure 3.3. The hyper implementations follow the architecture as presented in Figure 1.1 and the formula $in_a \leftrightarrow \Diamond out_b$. The trees of the hyper implementations are pictured as finite-state Moore machines whose infinite unrolling builds the trees. State labels correspond to the outputs of the process, and edge labels correspond to the branching structure. The hyper implementation of process a in Figure 3.3a outputs the communication bit c_a together with b's special variable n_b. Process b *branches* according to the special variable n_b and the output of the environment in_a. We continue with the explanation of the hyper implementations' behavior after introducing the local synthesis procedure.

→ p. 6

The synthesis of hyper implementations is the problem of generating a system that, assuming the time-bounded information-flow assumption is satisfied by the other process, satisfies the local specification. The crucial part in this process is finding the sweet spot between providing and restricting access to the unobservable environment output. The local synthesis process must access the variables at some point to satisfy the specification. On the other hand, accessing the variables must only be possible if the assumption ensures that the information is provided. We transform this argument into

a formal statement to be guaranteed by the local synthesis process and call it *locality condition.*

Definition 16 (Locality condition)**.** *Given the time-bounded distinguishability relation Λ_p for process p, the* locality condition *η_p for p is the 2-hyperproperty induced by the following relation R:*

$$\begin{aligned} R = \{(t,t') \in (2^{O_e \cup I_p \cup \{n_p\}})^\omega \times (2^{O_e \cup I_p \cup \{n_p\}})^\omega \mid \\ \text{if } (t|_{O_e}, t'|_{O_e}) \in \Lambda_p, \text{ then } t[0..m]|_{O_p} = t'[0..m]|_{O_p} \text{ and} \\ \text{if } (t|_{O_e}, t'|_{O_e}) \notin \Lambda_p, \text{ then } t[0..m']|_{O_p} = t'[0..m']|_{O_p}\} \end{aligned}$$

where m is the smallest natural number such that $n_p \in t[0..m]$ or $t[0..m] \mid_{I_p} \neq t'[m] \mid_{I_p}$ (and ∞ if no such m exists), and m' is the smallest natural number such that $t[0..m'] \mid_{I_p} \neq t'[0..m'] \mid_{I_p}$ or $t[0..m'] \mid_{\{n_p\}} \neq t'[0..m'] \mid_{\{n_p\}}$ (and ∞ if no such m' exists).

The locality condition specifies until what time point the outputs of two traces must be equivalent. The first case is the most intuitive: For traces related by the time-bounded distinguishability relation, the outputs must be equivalent up to the point where n_p is observed on the left trace or until there is a difference on the local inputs. Note that the second condition is necessary for environment outputs that are directly observed by p. In case the traces are not related by Λ_p, then the outputs must be equivalent until the local inputs to p are different on those traces or there is a difference in n_p. Again, the first case implements standard distinguishability whenever local inputs are different, whereas the second one guarantees that there must be an observable difference because the other process sets n_p differently. We now state that the correctness of hyper implementations depends on the specification, the locality condition, and the assumption for the other component.

Definition 17 (Local correctness of hyper implementations)**.** *Let p and q be processes, let φ_p be the local specification of p, let η_p be its locality condition, and let χ_q be the information-flow assumption of q. The hyper implementation h_p of p is* locally correct *if it satisfies φ_p, η_p, and χ_q.*

Example 19. We return to the example hyper implementations in Example 18 and Figure 3.3. Process a in Figure 3.3a satisfies the locality condition, the time-bounded information-flow assumption of process b. Note that its own specification is *true*, so no further requirement must be satisfied, and no trace pair is related in Λ_a. After reading the first value in_{a}, a sets n_{b} in case it was in_{a} and stays at $\neg n_{\mathsf{b}}$ in case of $\neg in$. Similar to → p. 53

n, c_{a} switches to $\neg c_{\mathsf{a}}$ only in case of in_{a}. Process b will be able to observe this difference on n_{b} and c_{a}. In addition to $in_{\mathsf{a}} \leftrightarrow \Diamond out_{\mathsf{b}}$, b now must also satisfy the locality condition for its own assumption. Recall that while constructing b, the implementation of a is not known. Even after observing that in_{a} is true after the first stop, b has to wait to set out_{b} to true until n_{b} is observed. Afterward, it can finally set out_{b} in state h_3^b. This way, the hyper implementation of b will be correct for whatever time point is chosen by a to transmit the information of the value in_{a} being *true*.

The next step in performing assume-guarantee synthesis with time-bounded information-flow assumptions is the composition of the hyper implementations. We assume that we obtained two correct hyper-implementations, one for p and one for q. Recall that these are trees branching over the inputs and labeled with the process's outputs. The main task for composing the implementations is the replacement of the artificial n_p and n_q and the unobservable inputs $O_e \cap I_p$ and $O_e \cap I_q$ with the actual transmittance of information. Only afterward will we have two local, practical implementations of the processes. In a first step, we build the composition of the two hyper implementations. The composition observes every variable, but already limits the local views of the processes and restricts the behavior to the actual implementations. It takes advantage of the information-flow assumption, guaranteeing that the process will be able to distinguish the scenarios *before n* arrives.

Definition 18 (Composition of hyper implementations)**.** *Let p and q be two processes with hyper implementations given as infinite $2^{O_e \cup I_p \cup \{n_p\}}$-branching $2^{O_p \cup \{n_q\}}$-labeled tree h_p for process p, and an infinite $2^{O_e \cup I_q \cup \{n_q\}}$-branching $2^{O_p \cup \{n_p\}}$-labeled tree h_q for process q. We define the composition $h = h_p \parallel h_q$ to be a 2^{O_e}-branching $2^{O_p \cup O_q}$-labeled tree, where $h(v) = (h_p(g_p(v)) \cup h_q(g_q(v))) \cap (O_p \cup O_q)$ and g_p, g_q are defined as follows:*

$$g_p(\epsilon) = \epsilon \qquad g_p(v \cdot x) = g_p(v) \cdot ((x \cap I_p) \cup (h_q(g_q(v)) \cap (I_p \cup \{n_p\})))$$
$$g_q(\epsilon) = \epsilon \qquad g_q(v \cdot x) = g_q(v) \cdot ((x \cap I_q) \cup (h_p(g_p(v)) \cap (I_q \cup \{n_q\})))$$

This composition needs a more detailed explanation before we show the outcome in the example by composing the hyper implementations in Figure 3.3. We first analyze the functions g_p and g_q that output the empty word ϵ of the empty input ϵ. For a finite word $v \cdot x$ where $v \in (2^{O_e})^*$ and $x \in 2^{O_e}$, g_p computes the sequence of inputs that p will observe given $v \cdot x$ and given the implementation h_q of q. It, therefore, computes the subset of the current set of variables x that is observable by p with $(x \cap I_p)$. Furthermore, p observes the output of q for the inputs of the environment until the previous step – there is a one-step offset in the interaction between the two processes. The part

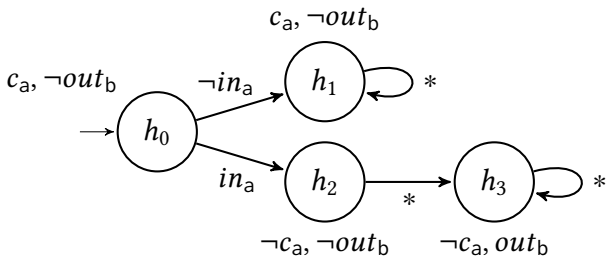


Figure 3.4 The result of composing the hyper implementations in Figure 3.3 as defined in Definition 18.

$(h_q(g_q(v)) \cap (I_p \cup \{n_p\}))$ realizes the previously explained computation by extracting the output of h_q for the variable sequence $g_q(v)$ (which recursively computes the inputs to q) and intersects it with the observable inputs of p combined with n_p. The last part of the function is a recursive function of g_p on v that generates the left part of the input word. Provided with the correctly separated inputs to the processes, h then computes the outputs of the hyper-implementations and eliminates the additional variables n_p and n_q. Note that n_p and n_q are still needed while computing the output in h_p and h_q, but are not part of the output of the composition. Here, we solely rely on the fact that there is information flow and the processes can react to it in time (as guaranteed by the assumptions).

Example 20. Consider the hyper implementations in Figure 3.3 which we compose to the system in Figure 3.4. The initial state h_0 is the combination of states h_0^a and h_0^b as they both start in the initial states, and we copy the outputs from the hyper implementations to the composition while removing any n. In the first branching step, the composition branches for the value of in_a and moves process a to h_1^a or h_2^a for $\neg in_a$ and in_a, respectively. For process b, we will never immediately go to h_3^b from state h_0^b: We read the output of n_b of h_a in the first step, which is $\neg n_b$, forcing b to *not yet* respond to the information that in_a is true and to go to h_2^b. Only after one more step, b gets the output n_b being *true* as input and branches to h_3^b. → p. 53

The composition of the hyper implementations replaces the assumptions with the actual implementations and behavior of the processes. It realizes the goal specification for both processes as a combined, monolithic system. In a last step, we have to decompose the system into two independent process implementations that together satisfy the specification. Note that it is not immediately obvious that this is possible: The composition has full observation of all environment outputs, which is the tree branching h branching over 2^{O_e}, whereas the final implementations will only branch over 2^{I_p}

and 2^{I_q}. More intuitively, the composition can distinguish all traces, whereas the final implementations must only distinguish the traces that are different for their respective local inputs. We identify the set of (finite) traces for which the local process has the same observation as is imposed by the definition of the architecture. Furthermore, we only consider traces that show the other component's actual output behavior, meaning that some component output combinations are not present in the set of traces. We call this set of traces the *knowledge set* for a given sequence of environment outputs.

Definition 19 (Knowledge set)**.** *Let p and q be two processes with composed hyper implementations $h = h_p || h_q$. For a finite trace $v \in (2^{I_p})^*$ of inputs to p, we define the knowledge set $K_p(v)$ to be*

$$K_p(v) \triangleq \{w \in (2^{O_e})^* \mid g_p(w) = v\}.$$

Note that the restriction on the other component's actual behavior is somewhat hidden inside the function g_p. It recursively calls h_q to extract its outputs for the environment and q's outputs. The crucial argument is that, in the composition, the individual processes do not output differently and, therefore, react to unobservable inputs. If this is the case, then we can simply collapse all those traces into one and choose one of the respective outputs as the reaction. We formalize this property in the following lemma.

Lemma 1. *For all $v, v' \in (2^{I_p})^*$, if $K_p(v) = K_p(v')$ then $h(v) \mid_{O_p} = h(v') \mid_{O_p}$.*

Proof. If $K_p(v)$ is a singleton or empty, then the lemma is trivially true. Assume $|K_p(v)| \geq 1$ and there exists $w, w' \in K_p(v)$ s.t. $h(w) \mid_{O_p} \neq h(w') \mid_{O_p}$. Since w and w' agree on the local inputs to p, there exists at least one $a \in O_e \backslash I_p$ s.t. $w \mid_{O_a} \neq w' \mid_{O_a}$. Then, $h_p(w) \neq h_p(w')$ has to hold following the function g_p of Definition 18. Given → p. 55 the locality from Definition 16, this is only possible if n_p was observed in the input → p. 54 to h_p, which is replaced by the output of h_q in Definition 18. Since h_q satisfies the time-bounded information-flow assumption χ_p from Definition 13, h_p observes a differ- → p. 47 ence in I_p before it reacts to the global inputs. Therefore, $h(w) \mid_{O_p} = h(w') \mid_{O_p}$ which contradicts the assumption. □

We now proceed with the decomposition, i.e., how to construct local implementations from the composition.

Definition 20 (Local implementations from hyper implementations)**.** *Let p and q be two processes with time-bounded information-flow assumptions χ_p and χ_q, and $h = h_p \parallel h_q$ be*

the composition of their hyper implementations. For $j \in \{p, q\}$ the strategy f_j, represented as a 2^{I_j}-branching 2^{O_j}-labeled tree for process j, is defined as follows:

$$f_j(\epsilon) = \epsilon \qquad f_j(v) = \begin{cases} \emptyset & \text{if } |K_j(v)| = 0 \\ h(min(K_j(v)))\ |_{O_j} & \text{if } |K_j(v)| > 0 \end{cases}$$

where $min(K_j(v))$ is the smallest trace based on an arbitrary order over $K_j(v)$.

The first case of the definition represents input traces that are impossible, given the behavior of the other component. In this case, the knowledge set is empty as there is no environment output trace for which the computed trace has this input; We set the label of the process to a *null* value. For the second case, we compute the output of the composition for *one* of the traces in the knowledge set and project it to the component outputs. We choose the smallest trace in $K_j(v)$ for some order, as we already know that the outputs will be equivalent for all those traces (see Lemma 1). The correctness of the decomposition is stated in the following theorem:

Theorem 8. *Let p and q be two processes with time-bounded information-flow assumptions χ_p and χ_q, let $h = h_p \parallel h_q$ be the composition of the hyper implementations, and f_p and f_q be the local strategies. Then, for all $v \in (2^{O_e})^*$ it holds that $h(v) = f_p(g_p(v)) \cup f_q(g_q(v))$ where g_p, g_q are defined as follows:*

$$g_p(\epsilon) = \epsilon \qquad g_p(v \cdot x) = g_p(v) \cdot ((x \cap I_p) \cup (f_q(g_q(v)) \cap I_p))$$
$$g_q(\epsilon) = \epsilon \qquad g_q(v \cdot x) = g_q(v) \cdot ((x \cap I_q) \cup (f_p(g_p(v)) \cap I_q))$$

Proof. Proof by induction over $v \in (2^{O_e})^*$. *Base case:* Let $v = \epsilon$, then $f_p(g_p(\epsilon)) \cup f_q(g_q(\epsilon)) = h(\epsilon) = \epsilon$. *Induction Step:* The induction step is shown from $v \in (2^{O_e})^*$ to $v \cdot x \in (2^{O_e})^*$, with $x \in 2^{O_e}$. Inserting g_p from Theorem 8 we obtain $g_p(v \cdot x) = g_p(v) \cdot ((x \cap I_p) \cup (f_q(g_q(v) \cap I_p))$. Since $f_q(g_q(v))$ and $g_p(v)$ are assumed to be correct, we show that the input trace returned by g_p and given to f_p is correct: The input is local to p because $x \cap I_p$ and $f_q(g_q(v)) \cap I_p)$ remove unobservable inputs, and all outputs of the previous step from q are added to the current input. It remains to show that the outputs of the local strategies combined are equal to the output of h: $h(v \cdot x) = f_p(g_p(v \cdot x)) \cup f_q(g_q(v \cdot x))$. Let $v' \cdot x' = g_p(v \cdot x)$. Given Definition 20 and → p. 57 Definition 19, we know that $f_p(v' \cdot x') = h(K(w))\ |_{O_p}$, with $w \in K_p(v' \cdot x')$. Since $K_p(v')$ → p. 57 is assumed to be correct, we show that adding x' to v' still results in the correctness of $h(K(w))$. Following Lemma 1, all elements in $K(v' \cdot x')$ and therefore all corresponding paths in h have the same label, and picking any with $min(K_p(v' \cdot x'))$ is correct. It

follows that $h(v \cdot x) \mid_{O_p} = f_p(g_p(v \cdot x))$. Using the same argument for f_q by interchanging p and q in every index yields the correctness of the theorem, i.e., for all $v \in (2^{O_e})^*$ it holds that $h(v) = f_p(g_p(v)) \cup f_q(g_q(v))$. □

After showing the correctness of the decomposition, we can conclude with the following corollary stating the correctness of the implementation.

Corollary 1. *Let $(I_p, I_q, O_p, O_q, O_e)$ be an architecture and $\varphi = \varphi_p \wedge \varphi_q$ be a specification. If the hyper iplementations h_p and h_q are locally correct, then the implementation (f_p, f_q) satisfies φ.*

3.3.2 Synthesis with Finite Communication

The synthesis with time-bounded information-flow assumptions computes (hyper) implementations by synthesizing processes with *full observation* of the environment outputs. They, therefore, branch according to variables that are not actually observable once the systems are deployed. This is a major drawback during synthesis: Only after enforcing the complex locality condition can we guarantee that access to variables is at least as restricted as in the architecture. In this section, we introduce a synthesis approach that relies on only the variables that are observable by the process. This includes all observable process outputs and the additional information guaranteed by the information-flow assumptions. We achieve this by imposing two assumptions on the architecture: Firstly, we assume that the time-bounded information-flow assumption only relies on variables that the sending process can observe, and secondly, we assume that the time-bounded assumption can be decomposed into a finite set of classes, so-called information classes, on which the receiving process can react. These classes are then encoded into a *trace property*, not a hyperproperty, as the locality condition would be one, and, together with the goal specification, translated into an adapted synthesis goal. We show promising experimental results for this approach, which shows that this approach outperforms existing techniques for distributed synthesis.

We begin with the decomposition of a time-bounded information-flow assumption into a (finite) set of information classes. Information classes work in the following way: We group the traces into classes, so that the process must at some point get aware that the current trace t is in the class $[t]_p$. On these traces, there exists an output that is not a finite violation for any other trace in the set. We define the information classes with respect to the time-bounded distinguishability relation Λ, but take a detour to define the set of traces on which the process will not see any difference. The so-called *ambiguous* traces to a trace t are exactly those that are not related to t in Λ.

Definition 21 (Ambiguous traces)**.** *Given a time-bounded distinguishability relation Λ_p for process p, we define the set of* ambiguous *traces $\alpha(t)$ of a trace $t \in (2^{O_e})^\omega$ as follows:*

$$\alpha(t) = (2^{O_e})^\omega \setminus \{t' \in (2^{O_e})^\omega \mid (t, t') \in \Lambda_p\}$$

Intuitively, this computes the dual of the time-bounded distinguishability relation and a refinement on the information that the process gets: The least information a process can get is that the observed trace could be any trace. This happens when the set of ambiguous traces is the whole set of environment traces. Any ambiguous set that is a *subset* of all the environment traces tells the process which traces are *possible* for the ones that are in the set and *impossible* for the traces that are outside of the set. We now turn ambiguous traces into information classes in order to read off enough information from an observation to react correctly for a whole class of traces. Those classes are built by grouping all traces that share the same ambiguous trace set and, therefore, carry the exact same information. We call them information classes:

Definition 22 (Information classes)**.** *Given a time-bounded distinguishability relation Λ_p for process p, the* information class *$[t]_p$ of a trace $t \in (2^{O_e})^\omega$ is the following set of traces:*

$$[t]_p = \{t' \in (2^{O_e})^\omega \mid \alpha(t) = \alpha(t')\}$$

Example 21. We illustrate the concept of information classes over ambiguous traces with the bit transmission protocol for the formula $\varphi = in_{\mathsf{a}} \leftrightarrow \Diamond out_{\mathsf{b}}$. The time-bounded distinguishability relation implies two information classes for process b. The first class contains all traces where in_{a} is true in the first step and an arbitrary sequence of environment outputs afterward. For any such traces, the ambiguous traces are exactly those that start in_{a} in the first step. It does not contain the traces with $\neg in_{\mathsf{a}}$ in the first step, since for these traces satisfying the formula φ would immediately yield a violation; hence, they are related by time-bounded distinguishability. The second class contains all traces on which $\neg in_{\mathsf{a}}$ holds in the first step of the trace. For these traces, the ambiguous traces are *all* environment traces, as setting $\neg out_{\mathsf{b}}$ will never be a finite violation on any trace. Therefore, they are not related by time-bounded distinguishability and, hence, are part of the ambiguous traces. Based on these two information classes, we will be able to fully synthesize the process b.

In general, the information classes can be arbitrarily precise, and the total number of information classes can be infinite. We enforce the assumption that the formulas we consider only imply a finite number of information classes. For each of the information

classes, we now change the goal specification to be a local specification. The reasoning is similar to the locality condition in Definition 16, which restricts the process to only → p. 54 access environment outputs once the time-bounded assumption allows it to. Here, we constrain the reactions to be equivalent for all the traces in an information class.

Definition 23 (Relativized specification)**.** *For a process p with specification φ_p and an information class ι, the relativized specification $\varphi_{p,\iota}$ is the following trace property over $(I_p \cap O_e) \cup O_p$:*

$$\varphi_{p,\iota} = \{t_e \sqcup t_p \mid t_e \in (2^{I_p \cap O_e})^\omega, t_p \in (2^{O_p})^\omega \text{ s.t. } \forall t_e' \in \iota.\ t_e' \sqcup t_p \vDash \varphi_p\}$$

Compared to the locality condition, the relativized specification is a trace property, not a hyperproperty. The information classes are explicit decompositions of the hyperproperty and have the benefit of being way more efficient to compute. We now assume a finite set of information classes IC for a specification φ and use IC additionally as input alphabet to the process. The remaining task is to replace unobservable environment outputs by sequences of information classes in a way such that the reaction of the process is correct. Once a process receives the information class input $\iota \in IC$, then the process can react to the information that the environment output trace is one of the traces in ι. In the actual implementation, after composition and decomposition, the information class is replaced with the actual outputs of the other process. The component specification, which is the trace property that we finally synthesize, requires that the process satisfies the relativized specification, once revealed, assuming that the correct information classes are provided on every trace.

Definition 24 (Component specification)**.** *For process p with specification φ_p, the component specification $\langle\varphi_p\rangle$ over $(I_p \cap O_e) \cup IC \cup O_p$ is defined as*

$$\langle\varphi_p\rangle = \{t \in (2^{(I_p \cap O_e) \cup IC \cup O_p})^\omega \mid \text{if } t \vDash \psi \text{ then } t \vDash \bigwedge_{\iota \in IC} (\Diamond c \rightarrow \varphi_{p,\iota})\}$$

where ψ is the following trace property over $(I_p \cap O_e) \cup IC \cup O_p$:

$$\psi = \{t \in (2^{(I_p \cap O_e) \cup IC \cup O_p})^\omega \mid \exists t' \in (2^{O_e})^\omega.\ t|_{I_p \cap O_e} = t'|_{I_p \cap O_e} \\ \text{and } t \vDash \Diamond[t'] \text{ and exactly one element of } IC \text{ occurs on } t\}$$

Note that traces of the component specification only range over $I_p \cap O_e$, instead of the full set of environment outputs O_e.

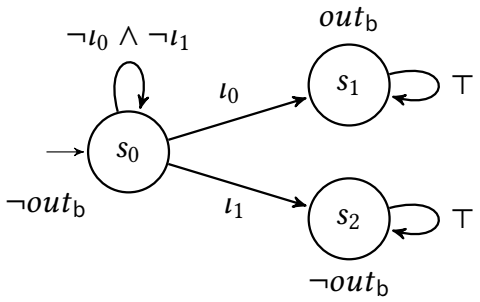


Figure 3.5 The component specification for process b in the bit transmission example. The information classes ι_0 and ι_1 will be replaced by the behavior of process a during composition.

Example 22. We return to the running example of this chapter. Given the previously created two information classes (see Example 21), we introduce a new alphabet $IC = \{\iota_0, \iota_1\}$ to the inputs of b, where c_0 represents the information class of traces where in_a is true in the first step and c_1 the information class of traces where $\neg in_\mathsf{a}$ is true in the first step. For ease of readability, we show the component specification as an LTL formula: → p. 60

$$\langle\varphi_b\rangle = \left(\Diamond\left(\iota_0 \vee \iota_1\right) \wedge \left(\Box\neg\iota_0 \vee \Box\neg\iota_1\right)\right) \rightarrow \left(\left(\Diamond\iota_0 \rightarrow \Diamond out_\mathsf{b}\right) \wedge \left(\Diamond\iota_1 \rightarrow \Box\neg out_\mathsf{b}\right)\right)$$

The left-hand side of the formula represents the assumption ϕ in the component specification, stating global mutual exclusiveness of the information classes and the eventual reveal of the information class on every trace. The right-hand side states the guarantee of the respective relativized specifications, forcing either $\Diamond out_\mathsf{b}$ or $\Box\neg out_\mathsf{b}$. The synthesized result of process b is depicted in Figure 3.5. In a last step, the information class symbols are replaced by the behavior of a separately synthesized process a that satisfies the time-bounded information-flow assumption of process a.

Composition and decomposition of the hyper implementations is analogous to the hyper implementations, as shown in Definition 18 and Definition 20. → p. 55

We show how to compute automata for the component specification. The automaton can then be used to synthesize the process that satisfies the component specification. The algorithm starts by building an automaton for each information class. Recall that the number of information classes is assumed to be finite, and the algorithm that iteratively computes automata for information classes is guaranteed to terminate. The starting points are the automata for the information classes, given the time-bounded information-flow assumption. The number of information classes, although finite, is not

Algorithm 1: Trace Information Classes for Process p

1 **let** traceInformationClasses($\mathcal{A}_{\Lambda_p}$):=
2 **let** infClasses = []
3 **while** $\mathcal{L}(\mathcal{A}_{\Lambda_p}) \neq \emptyset$ **do** // Check if a distinguishable trace is left
4 **let** $\mathcal{A}_v$ = acceptingTrace($\mathcal{A}_\Lambda$)$|_l$ // Choose one trace v
5 **let** $\mathcal{A}^v_\alpha = \left(\overline{\mathcal{A}_{\Lambda_p}} \cap (\mathcal{A}_v \oplus \mathcal{A}_\top)\right)|_r$ // Traces not related to v in Λ_p
6 **let** $\mathcal{A}^0_\alpha = \left(\overline{\mathcal{A}_{\Lambda_p}} \cap (\mathcal{A}_\top \oplus \mathcal{A}^v_\alpha)\right)|_l$ // Traces that share an undistinguished trace
7 **let** $\mathcal{A}^+_\iota = \left((\mathcal{A}_{\Lambda_p} \cap (\mathcal{A}^o_\iota \oplus \mathcal{A}_\top)) \cap (\mathcal{A}_\top \oplus \mathcal{A}^v_\alpha)\right)|_l$ // Disting. from more traces
8 **let** $\mathcal{A}^-_\iota = \left((\overline{\mathcal{A}_{\Lambda_p}} \cap (\mathcal{A}^o_\iota \oplus \mathcal{A}_\top)) \cap (\mathcal{A}_\top \oplus \overline{\mathcal{A}^v_\alpha})\right)|_l$ // Disting. from less traces
9 **let** $\mathcal{A}_\iota = \mathcal{A}^0_\iota \cap \overline{\mathcal{A}^+_\iota} \cap \overline{\mathcal{A}^-_\iota}$ // Remove all traces with different information classes
10 infClasses.append($\mathcal{A}_\iota$) // Store automaton for the information class
11 $\mathcal{A}_{\Lambda_p} = \mathcal{A}_{\Lambda_p} \cap (\overline{\mathcal{A}_\iota} \oplus \mathcal{A}_\top)$ // Remove all traces of the information class
12 **return** infClasses

known in advance. We build an iterative algorithm over the distinguishability relation Λ_p of p that extracts automata accepting the traces of an information class one by one. The algorithm is shown in Algorithm 1. The first step is to check whether a trace exists for which the algorithm has not computed an information class yet. In this case, the emptiness check of the distinguishability automaton returns false. With computing an accepting trace of the distinguishability automaton, we extract one trace v that is distinguished from at least one other trace. Note that the accepting trace actually is a tuple of traces, since $\mathcal{A}_{\Delta_p}$ is a 2-tape automaton, but we project it to the left tape with $|_l$ (and in later steps of the algorithm to the right tape with $|_r$). We use the automaton for the trace to compute the set of ambiguous traces for v by intersecting $\mathcal{A}_v$, the automaton that accepts only the trace v, with the negation of the distinguishability relation. We first compose $\mathcal{A}_v$ with any possible trace to obtain a 2-tape automaton. The intersection of this automaton with the negation of the distinguishability relation accepts all pairs of traces (v, x), where x is any trace that is ambiguous to v. After projecting this 2-tape automaton to the right-hand side, we obtain the automaton that accepts all ambiguous traces of v. The next three steps are necessary to first find all traces that could be in the same information class as v and then compute all traces that can not be in the same class because they have either fewer or more ambiguous traces. We start in Line 6 with an automaton that accepts all such candidates v' and intersect the complemented distinguishability automaton with a 2-tape automaton accepting every trace on the left and the ambiguous traces of v on the right. The resulting automaton contains all v's that have at least one common ambiguous trace with v. Lines 7 and 8 filter this automaton by checking whether some v' is distinguished from more traces and whether

some v' is distinguished from fewer traces. The former is done by intersecting it with the left-hand side of the distinguishability automaton, and an intersection with the ambiguous trace automaton afterward. The traces accepted by the automaton are all v's for which the distinguishability relation has a trace in $\mathcal{A}_i^0$. The latter limits the complement of the distinguishability automaton to the candidate traces and computes the traces with a right-hand side trace that is not in the ambiguous trace set of v. In a last step, we complement the automaton from Lines 7 and 8 and intersect it with the automaton for the candidate traces, thereby limiting the v's in the automaton for the information class to those that have an *equivalent* set of ambiguous traces.

Algorithm 1 does not terminate for all LTL specifications. Consider a specification that has to output every input in the next step, which can be easily expressed in LTL, which is $\square(in_{\mathsf{a}} \rightarrow \bigcirc out_{\mathsf{b}})$. In this case, every trace is its own information class as it must be distinguished from every other trace according to the time-bounded information-flow assumption: Any difference at any point will lead to a finite violation on the other trace. However, for specifications that imply a finite number of information classes, the algorithm terminates and computes the automata for the information classes. We can state an upper bound on the size of the automaton.

Theorem 9. *For a process p with local specification φ_p, there exists a non-deterministic automaton with a quadruply-exponential number of states in the size of φ_p that recognizes an information class of p.*

Proof. We show the complexity of Lines 4 to 11 in Algorithm 1, which, together, compute an automaton for some information class of the distinguishability automaton. Recall that the 2-tape automaton for the distinguishability relation has a doubly-exponential number of states in the size of the specification, as proven in Theorem 5. Since we → p. 49 build the automaton on top of the distinguishability automaton, we already obtain a doubly-exponential blow-up, and any operation that builds on the distinguishability automaton will have at least a doubly-exponential number of states. The distinguishability automaton is a universal automaton; complementing it to a non-deterministic automaton will not increase the number of states. However, any operation with a non-deterministic automaton forces us to nondeterminize the distinguishability automaton, which implies an exponential blow-up. Negating any non-deterministic automaton is also exponential, which happens in Lines 8 and 9. In total, $\mathcal{A}_i^+$ is exponential in $\mathcal{A}_{\Lambda_p}$ because of the non-determinization of $\mathcal{A}_{\Lambda_p}$, and separately, $\mathcal{A}_i^+$ is also exponential in $\mathcal{A}_{\Lambda_p}$ because of the complementation of $\mathcal{A}_\alpha^v$. Both are complemented in Line 10, such that we conclude with two additional exponents in the number of states. □

In the algorithm described above, the information classes are used to build the relativized specification in Definition 23. For every environment trace that is observed, → p. 61 the specification enforces that the output that is used to satisfy the specification on this trace is also correct for any other trace in the information class. Since we already computed the automata that accept all traces in an information class and have the automaton for the LTL specification, we can immediately build the automaton for the relativized specification.

Theorem 10. *For a process p with local specification φ_p and an automaton $\mathcal{A}_\iota$ for one information class c, we can build a universal automaton accepting the relativized specification $\varphi_{p,\iota}$ with a polynomial number of states in $\mathcal{A}_\iota$.*

Proof. The automaton that we build reads in traces over $(I_p \cap O_e) \cup O_p$ and only accepts a trace if the sequence of outputs ($o \in O_p$) is accepted by the specification φ and works for all traces accepted by the automaton for the information class $\mathcal{A}_\iota$. We first restrict $\mathcal{A}_\varphi$ to only accept traces in $\mathcal{A}_\iota$ by intersecting both and obtain $\mathcal{B}$. $\mathcal{B}$ accepts all input sequences that are in c and satisfy the specification. This condition is too weak; we must also verify that the output is accepting for all other traces. Therefore, we universally branch over the traces in the information class automaton $\mathcal{A}_\iota$ and verify that the sequence is in $\mathcal{A}_\varphi$ on all traces simultaneously. The resulting automaton $\mathcal{R}^c$ is a universal automaton that accepts the relativized specification and has a polynomial number of states in the size of $\mathcal{A}_\iota$. □

We can use the construction above for all information classes that we obtained from Algorithm 1. The automaton from the component specification, which is the input → p. 63 to the synthesis process, then unifies all relativized specification automata and jumps to the correct one once the information class is revealed. Note that we now use an additional input alphabet IC, where each $\iota \in IC$ represents one information class. To be more precise, even $\log(|IC|)$ number of additional inputs is enough to represent all information classes, but we use $|IC|$ for readability. We refer to the additional proposition for information class accepted by the automaton $\mathcal{A}_\iota^i$ with c_i for every automaton $\mathcal{A}_\iota^1, \dots, \mathcal{A}_\iota^k$.

Construction 5. The construction of the automaton for the component specification starts with the assumption ψ from Definition 24, which states that all locally (for p) observable → p. 61 traces t eventually observe exactly one c_i on the local inputs if some environment trace t' exists that looks like t to the process p. We transform each automaton accepting all traces in the information class $\mathcal{A}_\iota^i$ to automata $\mathcal{B}_\iota^i$ that accept traces over $I_p \cap O_e$ by

existentially projecting the atomic propositions in $O_p \backslash I_p$ and by adding the condition

$$\varphi = \Diamond \iota_i \wedge \bigwedge_{\iota_j \neq \iota_i} \Box \neg \iota_i.$$

The automaton $\mathcal{A}_\psi$ that accepts the assumption ψ of the component specification is then a disjunction over all $\mathcal{B}_\iota^i$. We now use this automaton to build an automaton for the component specification of $\langle \varphi_p \rangle$. The implication in the definition is transformed to a disjunction over the negation of the left-hand side and the right-hand side. The negation of the left-hand side is the automaton $\overline{\mathcal{A}_\psi}$. For the right-hand side, we first use the automata for the relativized specification per information class $\mathcal{R}_\iota^i$ and disjunctively add the automaton for $\Box \neg c_i$ to obtain $\mathcal{R}_\iota^{i'}$. The *conjunction* over all $\mathcal{R}_\iota^{i'}$ is the right-hand side of the implication, and after disjunction with the negation of the left-hand side $\overline{\mathcal{A}_\psi}$, we obtain the automaton $\mathcal{S}$ for the component specification.

Theorem 11. *For a process p with local specification φ_p and information class automata $\mathcal{A}_\iota^1, \dots \mathcal{A}_\iota^n$, there exists an automaton recognizing the component specification $\langle \varphi_p \rangle$ with an exponential number of states in the size of any $\mathcal{A}_\iota^1$.*

Proof. Construction 5 shows how to construct the component specification. During computation, we have to negate all $\mathcal{A}_\iota^i$ and get the non-deterministic automata for the universal automata $\mathcal{R}_\iota^{i'}$. Both operations are exponential in the size of the information class automata. Therefore, the component specification automaton $\mathcal{S}$ has an exponential number of states in the size of the information class automata. □

→ p. 65

3.3.3 Experiments

We conclude the section on synthesis with time-bounded information-flow assumptions with an experimental evaluation of the approach. The benchmarks we use make the synthesis approach with finite communication usable and show the effectiveness of the technique presented in Section 3.3.2. We presented automata constructions for all steps of the synthesis approach with finite communication. Our evaluation uses manually crafted translations from LTL formulas to information classes and to the component specification afterward. The baseline is an encoding from distributed synthesis to the synthesis of HyperLTL properties, and was presented in [94].

Distributed Synthesis as HyperLTL Synthesis. The encoding from distributed synthesis as HyperLTL formulas was presented in [94]. In this monolithic (in comparison to compositional) synthesis procedure, the observation and absence of observation

of variables is encoded in a HyperLTL formula called *dependence* in the following way:

$$D_{I,O} = \forall\pi\forall\pi'.\Big(\bigvee_{i\in I}(i_\pi \nleftrightarrow i_{\pi'})\Big)\,\mathcal{R}\,\Big(\bigwedge_{o\in O}(o_\pi \leftrightarrow o_{\pi'})\Big)$$

Recall that the release operator $\mathcal{R}$ of HyperLTL enforces that the right-hand side has to hold as long as the left-hand side is violated. The formula D_p states that as long as the local inputs to the process are equivalent to the local variables of p, the local outputs must also be equivalent. If there is necessary information flow from q to p, then the left-hand side must be released by process q and process p will be able to react to the information. The formula to be synthesized in the two-process distributed synthesis case is

$$\forall\pi\forall\pi'.\varphi_p \wedge D_{I_p,O_p} \wedge \varphi_q \wedge D_{I_q,O_q}.$$

After synthesizing this formula with a bounded synthesis technique [94], the obtained implementation is a composition of both processes, with the guarantee that the outputs of the local processes only depend on the local inputs. The system has then to be decomposed into two implementations of two processes p and q.

Encodings of Bounded Distributed Synthesis. The second approach we compare against is a bounded synthesis approach for distributed systems [24]. This work builds on the bounded synthesis techniques built for monolithic LTL synthesis [83], which transform the synthesis problem into a constraint-solving problem for a given bound n on the state space of the resulting implementation. If the constraint solver provides a solution, then this is a solution for the synthesis problem; if not, then the bound is increased and the process is repeated. For monolithic LTL synthesis, this process will eventually terminate; there is an upper bound on the size of the system such that the synthesis formula is satisfied if it is realizable. For distributed synthesis, the same does not hold: As the problem is undecidable, the bound will be increased infinitely if the constraint system is not satisfiable. In [24], the authors extend the encodings to also limit the access to variables as defined in the architecture.

Benchmarks. The three benchmarks that we consider are variations of the bit transmission protocol. We evaluate three parameterized specifications on two architecture versions with the three different approaches. The composition of the two processes is synthesized for the distributed synthesis of HyperLTL and the encodings of bounded distributed synthesis approaches. Since synthesis with information-flow assumptions is

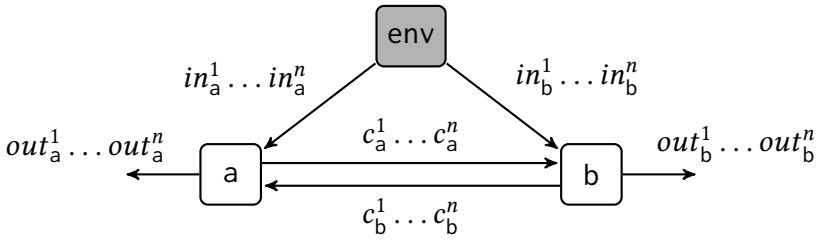


Figure 3.6 The parameterized architecture of a two-component distributed system. There are n inputs, outputs, and communication variables for each process.

a compositional approach, we have two separate synthesis tasks, one for each process. The benchmarks are the following three categories:

Atomic Commit. The benchmark *atomic commit* is a simplified version of the atomic commit protocol. In an atomic commit protocol, the processes must agree on a value, either 0 or 1, depending on their knowledge of other processes' inputs. The specification is parameterized in n and adheres to the following structure:

$$\varphi_b = \Big(\bigwedge_{1 \le i \le n} (in_{\mathsf{a}}^i) \Big) \leftrightarrow \Diamond out_{\mathsf{b}}$$

The formula φ is a big conjunction over indexed variables s.t. only if all of those are true, then the output is also true. Process b has to commit if process a received *true* as an input. As a further experiment, we use the same specification but changed for process a.

$$\varphi_a = \Big(\bigwedge_{1 \le i \le n} (in_{\mathsf{b}}^i) \Big) \leftrightarrow \Diamond out_{\mathsf{a}}$$

We annotate the experiment with only φ_b and $\varphi_a = \mathit{true}$ as *dir* and *bidir* if both processes have to commit to the value.

Eventual Commit. The eventual commit benchmark is an extension of the atomic commit, in which we do not enforce the value on the first step but on any future step. We implement this change in the following formula φ_a.

$$\varphi_b = \Big(\bigwedge_{1 \le i \le n} (\Diamond in_{\mathsf{a}}) \Big) \leftrightarrow \Diamond out_{\mathsf{b}}^i$$

Only if, at some point, all inputs were true, then the output of process b must be true as well. Again, we use two variants of this benchmark, one for the *dir* version and one for the *bidir* version.

Benchmark	Architecture	n	Sender	Recipient	Distributed Bosy [24]	HyperLTL [94]
Atomic Commit	dir	1	0.92	0.70	**1.41**	2.31
	dir	2	**0.36**	**1.28**	2.86	2.30
	dir	3	**0.92**	**0.68**	2.46	2.55
	dir	4	**0.92**	**0.79**	720.60	3.41
	dir	5	**0.92**	**0.68**	TO	9.27
	bidir	1	1.45	-	**0.96**	9.27
	bidir	2	**2.49**	-	TO	TO
	bidir	3	**79.18**	-	TO	TO
	bidir	4	TO	-	TO	TO
Evenutal Commit	dir	1	0.68	1.87	**0.92**	2.56
	dir	2	0.94	1.85	**0.96**	3.90
	dir	3	**202.09**	**1.78**	TO	TO
	dir	4	TO	TO	TO	TO
	bidir	1	**3.77**	-	4.63	147.46
	bidir	2	TO	-	TO	TO
Send All	dir	1	1.31	0.92	2.21	**1.58**
	dir	2	**1.78**	**0.92**	27.47	TO
	dir	3	TO	1.08	TO	TO

Table 3.1 This table shows the result of the experimental evaluation of the three approaches. The first column shows the benchmark name, the second column the architecture type, and the third column the parameter n that initializes the specification. We omit the entry of the receiver when every sender is also a receiver. The next two columns show the time in seconds for the sender and recipient processes, respectively. The last two columns show the time for the encodings of bounded distributed synthesis, and the last column shows the direct encoding into HyperLTL. The best result is highlighted in bold.

Send All. In the third benchmark, we change the specification to commit on isolated inputs, not on a Boolean connection of those. The truth value must be copied to the specific output independently of the other inputs. The formula is as follows:

$$\varphi_b = \bigwedge_{1 \leq i \leq n} \left(\Diamond in^i_{\mathsf{a}} \leftrightarrow \Diamond out^i_{\mathsf{b}} \right)$$

Process b can only set the outputs to true once the respective input is true, meaning that we are in a certain information class. Note that the information class will be changed every time we observe a new input being true.

We compare the evaluation of the three approaches in Table 3.1. Overall, the experiments show that the synthesis process with time-bounded information-flow assumptions scales to larger parameters than the other two approaches. In the atomic

commit benchmarks, which we consider to be the easiest of the three, the information flow synthesis terminates in under a second for all parameters, whereas the other two approaches either time out quickly or increase steadily up to 9 seconds. For the *bidir* case, we immediately beat the other approaches, but also time out for parameter 4. Note that in the bidir case, we actually have a HyperLTL synthesis problem even for the receiver, as the partner's information-flow assumption must be satisfied together with the receiver's component specification. For the eventual commit benchmark, we see a similar behavior, but the hyperproperty synthesis part is already hard for parameter 3, as can be seen for the sender. However, the recipient can be produced within a second until parameter 3. The *bidir* yields a timeout for parameter 2 for all approaches, but the difference between the information-flow synthesis and the HyperLTL encoding is still two orders of magnitude. The most complicated benchmark is send all, where the number of outputs scales with the parameter n, not only the number of inputs. This leads to a timeout throughout all approaches already for parameter 3. Most interestingly, the synthesis of the recipient can still be done in about 1 second - the component specification of the recipient is *only* a trace property, and synthesis of trace properties scales further than synthesis of hyperproperties. However, the sender is synthesized for the information-flow assumption, and we cannot extract a full implementation without the sender.

3.4 Prefix Information Flow

Time-bounded information-flow assumptions have two significant drawbacks. First, they only guarantee that information will eventually be transmitted to the receiving process without specifying an exact time point. While this suffices for liveness specifications, where reactions can be delayed until the information arrives, it is inadequate for safety specifications, where delayed reactions can immediately violate the required reaction window. Consequently, time-bounded information-flow assumptions are not suitable for safety properties, and the compositional synthesis approach described in Section 3.3.2 is restricted to liveness specifications. Second, the previous synthesis → p. 59 procedure limits the sender to transmitting only a *finite* number of information classes, which in turn restricts the amount of information the receiver can react to. This further narrows the range of applicable specifications. To address these issues, we introduce a new class of information-flow assumptions specifically designed for safety specifications with precise deadlines for information reception. Instead of considering entire

traces, we focus on prefixes of infinite traces and compute the latest possible time point for information reception.

3.4.1 Prefix Information Flow Assumptions

We begin by analyzing the traces that must be distinguished by some receiver. The example we use throughout this section is the formula $\varphi = \square(in_{\mathsf{a}} \leftrightarrow \bigcirc out_{\mathsf{b}})$, which states that *every* input to process a must be copied to the output of process b in the next time step. The time-bounded information-flow assumption is not applicable here; it would enforce that on traces t and t' where some a on time step i is true on t but false on t', there will *at some point* be a difference in the observable inputs to process b. This can not be a sufficient condition: Whenever the synthesis procedure for the time-bounded information-flow assumption produces an implementation of the sender that does not immediately forward the input, it will not be correct for any recipient implementation. It needs a more precise way of specifying the information flow between the processes. We do so by introducing a new classification of distinguishable traces, based solely on their prefix.

Definition 25 (Prefix distinguishability)**.** *Let φ_p be a safety LTL formula and the goal for process p. The* prefix distinguishability relation *is defined as*

$$\begin{aligned}\rho_{\varphi_p} = \{(t, t') \in (2^{O_e})^m \times (2^{O_e})^m, m \in \mathbb{N} \mid \forall t_p \in (2^{O_p})^m.\\ t \sqcup t_p \not\models_m \varphi_p \text{ or } t' \sqcup t_p \not\models_m \varphi_p\\ \text{and } \forall n \in \mathbb{N}, n < m.\ \exists t'_p \in (2^{O_p})^n.\\ t[0 \dots n] \sqcup t'_p \models_n \varphi_p \text{ and } t'[0 \dots n] \sqcup t'_p \models_n \varphi_p\}\end{aligned}$$

The prefix distinguishability relation reasons about tuples of *finite* traces of arbitrary length. The first condition states that, for the two related input traces of length m, the specification is violated for all possible output sequences for p of the same length. The second condition enforces that m is the first position at which the trace pair must be distinguished. For all previous positions of the traces, there exists a common output sequence that satisfies the specification on both traces. Every violation of a safety specification has a *minimal bad prefix* [67], and hence every violation that originates in the indistinguishability of two traces is captured by Definition 25.

Example 23. We give an example of prefixes related by ρ_{φ_p}, where $\varphi_p = \square(in_\mathsf{a} \leftrightarrow \bigcirc out_\mathsf{b})$ is the running example of this chapter. Consider the finite environment traces

$$t = \{\neg in_\mathsf{a}\}, \{in_\mathsf{a}\}$$
$$t' = \{in_\mathsf{a}\}, \{in_\mathsf{a}\},$$

both of length $m = 2$. These two traces are related by ρ_{φ_p}, since for every possible output sequence t_p of length $m = 2$, one of the traces will be violated, as the value of in_a must be copied to the output of b in the next time step and b cannot set $\neg out_\mathsf{b}$ and out_b at the same time. Let's consider the extensions of the traces

$$t = \{\neg in_\mathsf{a}\}, \{in_\mathsf{a}\}, \{in_\mathsf{a}\}$$
$$t' = \{in_\mathsf{a}\}, \{in_\mathsf{a}\}, \{in_\mathsf{a}\}.$$

Now, t and t' are not related by ρ_{φ_p}, since the second condition of Definition 25 is violated: There is no output sequence of length $n = 2$ that satisfies the specification for both traces. As a last example, we crop t and t' to length $m = 1$, i.e., $t = \{\neg in_\mathsf{a}\}$ and $t' = \{in_\mathsf{a}\}$. Again, the traces are not related by ρ_{φ_p}, there is a common output sequence t_p of length $n = 1$ that satisfies the specification for both traces, the output in step 0 is not restricted by the specification.

→ p. 71

For liveness specifications, no two traces are related by this definition: One can inductively reason that for every $(t, t') \in v_{\varphi_p}$ this pair of traces is not in ρ_{φ_p}, i.e., $(t, t') \notin \rho_{\varphi_p}$, since for every chosen m, one can find an output trace of p that violates the formula after time point m.

We now turn the prefix distinguishability relation into an information-flow assumption that must be guaranteed by the component that observes the information that is needed by process p. The assumptions include specific time points at which the information must be present for the receiving process to react upon. The information-flow assumption, again, is a 2-hyperproperty enforcing that pairs of traces that are related by the prefix distinguishability relation have an observable difference for the process.

Definition 26 (Prefix information-flow assumptions)**.** *Let ρ_{φ_p} be the prefix distinguishability relation for p. The corresponding* prefix information-flow assumption $\mathcal{P}_p$ *is the 2-hyperproperty induced by the relation*

$$R_{\rho_{\varphi_p}} = \{(t, t') \in (2^{\mathcal{V}})^\omega \times (2^{\mathcal{V}})^\omega \mid \textit{if } \exists m \in \mathbb{N} \textit{ s.t. } (t[0 \dots m], t'[0 \dots m]) \in \rho_{\varphi_p} \textit{ then } t|_{I_p}[0 \dots m-1] \neq t'|_{I_p}[0 \dots m-1]\}$$

The information-flow assumption is a relation over *infinite* traces. For each such trace pair, we search for the time point m, on which the traces are distinguished by the prefix distinguishability relation ρ_{φ_p}. If such a point in time exists, then there must be a difference in I_p before the deadline m, and more specifically, on the time step $m-1$. Restricting the observable difference to happen before the *deadline* m is crucial for the receiving component.

Example 24. We use the formula of the previous example $\varphi_p = \square(in_\mathsf{a} \leftrightarrow \bigcirc out_\mathsf{b})$ on the architecture in Figure 3.6. The information-flow assumption can be specified in the HyperLTL formula → p. 68

$$\forall\pi\forall\pi'.(in_\mathsf{a}^\pi \leftrightarrow in_\mathsf{a}^{\pi'})\,\mathcal{W}(in_\mathsf{a}^\pi \nleftrightarrow in_\mathsf{a}^{\pi'} \wedge c_\mathsf{a}^\pi \nleftrightarrow c_\mathsf{a}^{\pi'}).$$

Compared to the time-bounded information-flow assumption, it is precise in the time point at which a difference must be observed on c_a, namely exactly when the inputs on two traces are different. Recall that the weak until operator $\mathcal{W}$ states that either the left-hand side holds forever, or it holds until the right-hand side is true. The first time the input to traces π and π' differ, the right-hand side must hold, and therefore the difference will be communicated to process p. Note that this immediate communication of the information is only a special case. There can be an arbitrary distance between the information being present on the inputs of a before transmitting it to b is necessary.

Unsurprisingly, every implementation of a distributed system satisfying safety LTL specifications satisfies the corresponding prefix information-flow assumption.

Lemma 2. *The prefix information-flow assumption is necessary for safety LTL formulas.*

Proof. Assume that there exists an implementation (f_p, f_q) satisfying the safety LTL specifications φ_p and φ_q but not $\mathcal{P}_p$ and $\mathcal{P}_q$. Since $\mathcal{P}_p$ is not satisfied, there exists a pair of traces π, t' such that $(t|_{O_e}[0\ldots m], t'|_{O_e}[0\ldots m]) \in \rho_{\varphi_p}$ and $t|_{I_p}[0\ldots m+1] = t'|_{I_p}[0\ldots m+1]$. Therefore, the deterministic system must choose the same output for the time step $m+1$ since the inputs are the same. This contradicts the assumption: Either $t[0\ldots m+1]$ or $t'[0\ldots m+1]$ is a minimal bad prefix since, otherwise, the traces would not be related by the prefix distinguishability relation. □

3.4.2 Automata for Prefix Information Flow Assumptions

We show how to compute automata that accept the prefix information-flow assumption for a given LTL formula in the safety fragment. The prefix information-flow assumptions build on the distinguishability relation, which is a natural starting point for the

construction. We will use the automata construction in the synthesis process that is described in Section 3.5. → p. 76

The biggest difference to trace distinguishability (see Definition 8) is that, although the traces of the implementations are infinite, the traces related by the prefix distinguishability are finite. The automaton accepting pairs of traces is therefore a non-deterministic finite automaton (NFA) instead of a non-deterministic Büchi automaton. We can show that the following theorem holds. → p. 38

Theorem 12. *For a component p and safety LTL formula φ_p, there exists a non-deterministic finite 2-tape automaton with a doubly exponential number of states in the size of φ_p that recognizes the prefix distinguishability relation ρ_{φ_p}.*

Proof. We construct a non-deterministic finite automaton (NFA) $\mathcal{F}$ that accepts precisely all pairs of traces over $(2^{O_e})^m \times (2^{O_e})^m$, where $m \in \mathbb{N}$, that are related by ρ_{φ_p}. Let φ'_p be the formula φ_p where all atomic propositions $a \in AP$ are renamed to a', and let $\mathcal{V}'$ be a set containing a copy v' of every variable $v \in \mathcal{V}$. We build the NBA $\mathcal{B} = \mathcal{A}_{\varphi_p} \times \mathcal{A}_{\varphi'_p}$, where $\mathcal{A}_{\varphi_p}$ and $\mathcal{A}_{\varphi'}$ are constructed with a standard LTL-to-NBA translation respectively, and the operator $\times$ builds the product of two NBAs. $\mathcal{B}$ now accepts all tuples of traces that each satisfy φ_p. Let C be the NBA that restricts the transition relation of $\mathcal{B}$ s.t. edges are only present if the output variables of p are equal, i.e., $\bigwedge_{o \in O_p} o \leftrightarrow o'$ holds, enforcing that both traces agree on the output while satisfying the specification. We now existentially project to the set $O_e \cup O'_e$ to build $\mathcal{D}$, whose alphabet does not contain the components' outputs. To accept the pairs of traces that do not satisfy the formula, we negate $\mathcal{D}$, denoted by $\overline{\mathcal{D}}$. In the last step of the construction, we transform the NBA $\overline{\mathcal{D}}$ to an NFA $\mathcal{F}$ using the *emptiness-per-state* construction of [22]. This yields an NFA that accepts the prefix distinguishability relation. The size of the automaton is doubly exponential in the size of the formula. The first exponent stems from the LTL to NBA construction, and the second from negating the automaton $\mathcal{F}$. □

Example 25. A simple automaton representing the prefix distinguishability relation for the running example $\varphi_p = \square(in_\mathsf{a} \leftrightarrow \bigcirc out_\mathsf{b})$ is shown in Figure 3.7a. The automaton accepts all pairs of finite traces that are equal on in_b until the second-to-last step, and are different on in_b in the last step. The automaton is a non-deterministic finite automaton (NFA) with two states, q_0 and q_1, where q_0 is the initial state and q_1 is the accepting state.

We continue by constructing automata that accept the prefix information-flow assumptions.

(a) The prefix distinguishability automaton. (b) The prefix information-flow automaton.

Figure 3.7 The automaton (NFA) accepting the traces related by the distinguishability relation for the running example is shown in Figure 3.7a. The automaton runs over a tuple of inputs, annotated with t and t', respectively. The relation accepts all pairs of finite prefixes that are equal on in_{b} until the second-to-last step, and are different on in_{b} in the last step. In Figure 3.7b, the automaton for the prefix information-flow automaton accepts all *infinite* traces, whose prefixes are in the automaton in Figure 3.7a, but forced to have a difference on the communication variables before a violation of the formula would happen.

Theorem 13. *For a component p and ρ_{φ_p} be the prefix distinguishability relation of p. There exists a non-deterministic 2-tape Büchi automaton with an exponential number of states in the size of φ_p that recognizes the prefix information-flow assumption $\mathcal{P}_p$.*

Proof. We use the automaton $\mathcal{D}$ from the proof of Section 3.4.2. This automaton accepts all infinite traces on which there must not be a distinction in the observable inputs. The part that is left to construct is the automaton that requires that, whenever the traces are not in $\mathcal{D}$, a difference is actually observable before there is a violation. We use an automaton $\mathcal{E}$ that requires that, on two input traces, we only accept them if they are different on the observable inputs to p at some point. We then build the automaton F of a composition of the bad prefix automaton of the formula φ_p such that F rejects whenever one of the traces reaches a bad prefix. Note that this is not a complement of the automaton but only a relabelling accepting with rejecting states only. We then build the automaton $\mathcal{G}$ that accepts a pair of traces only if they are accepted by $\mathcal{E}$ before being rejected by $\mathcal{F}$, meaning that the good prefix of E is shorter than the bad prefix of $\mathcal{F}$. The final automaton is a disjunction between $\mathcal{D}$ and $\mathcal{G}$, s.t., traces are accepted if they must not be distinguished, or if they are different on the observable inputs before some bad prefix is reached. The size of the resulting automaton is exponential in the size of the formula φ_p, only the translation from φ to $\mathcal{A}_\varphi$ is exponential. □

Example 26. We conclude this section by showing the automaton for the information-flow assumption of our running example. The automaton, whose alphabet is the set of all variables in the architecture, is presented in Figure 3.7b. It must accept all trace pairs on which there is a difference before one of the traces violates the formula, meaning that

one output of a trace was set incorrectly. The automaton accepts exactly those traces by checking if there is a difference in the step in which the process observes a difference in in_{a}. Because of the formula stating that every environment output must be replied to by the process correctly in *the next time step*, the transmission of the information must happen at the same time as the difference is observed. All other traces are rejected.

3.5 Distributed Synthesis with Prefix Information Flow

This section introduces a compositional synthesis approach for distributed systems with prefix information-flow assumptions. The algorithms build upon the prefix distinguishability relation and the prefix information-flow assumption and their automata constructions as presented in Section 3.4. We present a two-fold algorithm for compositional synthesis that decomposes the goal specification (one for each process) into two separate local specifications. The first is called *assume specification* and enforces that a process can rely on the fact that some information will be transmitted to the process *in time*. The second specification is called *guarantee* specification, as it, if it is realized by the sender, guarantees the flow of information. We combine both specifications for every process and obtain an overall specification that can be realized locally under full observation of all variables. The new algorithm is evaluated with an automata-based implementation of the presented approach against HyperBosy [94], a bounded synthesis approach for hyperproperties.

→ p. 70

3.5.1 Hyper Implementations for Prefix Assumptions

Hyper implementations, as introduced in Definition 15, are implementations that we use as intermediate results in the compositional synthesis process. In Section 3.3.1, we introduced hyper implementations for time-bounded information-flow assumptions that branched under full observation and some additional variable n. In this section, we define hyper implementations for the prefix information-flow assumptions and concretize them by immediately branching over information classes. Conversely, the safety hyper implementations will not branch over unobservable inputs but only over the process's inputs.

→ p. 52

We begin by defining the prefix information class. Every class consists of all traces that do not need to be distinguished by some process at a specific time point n. Recall that those traces are exactly the ones that are not related to each other by the prefix

(a) (b)

Figure 3.8 Figure 3.8a displays an automaton for a specific prefix information class of length n, whereas Figure 3.8b shows an automaton that represents infinitely many information classes of the running example specification. For every fixed time point n, all traces of length n that have the same environment output in the last step are in the same information class.

distinguishability relation. In contrast to time-bounded information classes, the prefix information classes are defined over finite traces instead of infinite traces.

Definition 27 (Prefix information class)**.** *Let ρ_p be the prefix distinguishability relation for p. The information class of a trace t at position $n \in \mathbb{N}$ is the set of traces*

$$[t]_p^n = (2^{O_e})^n \backslash \{t' \in (2^{O_e})^n \mid (t, t') \in \rho_p\}$$

In the example specification $\varphi_p = \square(in_a \leftrightarrow \bigcirc out_b)$, the information class at position n is the set of all traces that are equal on in_b until time step $n-1$. Now, assume we want to find the process's output in the next step. We already know that we cannot observe a difference in environment traces that must not necessarily be distinguished. Therefore, the information class specification, which is the specification for one (of the infinitely many) information classes, enforces that the *next* output of the process will be correct for all traces in the information class in the previous step.

Definition 28 (Prefix information class specification)**.** *Let φ_p be the LTL specification for component p, $n \in \mathbb{N}$, and let ι_{n-1} be a prefix information class at position $n-1$. The information class specification $\mathbb{C}_p^n \subseteq (2^{\mathcal{V} \setminus O_q})^\omega$ is defined as*

$$\mathbb{C}_p^n = \{t_e \sqcup t_o \mid t_e \in (2^{\mathcal{V} \setminus O_p})^n, t_o \in (2^{O_p})^n \\ \text{s.t. } \forall t_e' \in \iota_{n-1}.t_e'[0 \dots n] \sqcup t_o[0 \dots n] \vDash_n \varphi_p\}.$$

Example 27. We go back to the sequence transmission specification $\varphi = \square(in_b \leftrightarrow \bigcirc out_b)$ of our running example. Regarding Definition 27, there are infinitely many information classes, for each n, all traces that are equal until n are in the same information

class. Figure 3.8a displays the automaton for $n = 3$, where the environment first outputs in_{a} and then $\neg in_{\mathsf{a}}$. Technically, this yields infinitely many automata, each with n states that proceed from state to state n times before accepting the input trace. However, we can find a finite representation of all such automata that summarizes the information classes by some refinement: For the specification φ, it is sufficient to verify that the last step is equivalent, since for all information classes that behave this way, the same output will be a correct output. Figure 3.8b shows the automaton that accepts all prefixes for which the next output must be $\neg out_{\mathsf{b}}$. It unites all information classes that were equal on all previous steps until state q_1. We will show the computation of such automata for information classes in the following section.

With the assumption that we are given a finite set of information classes as subsets of $(2^{O_e})^*$, we are able to define the safety hyper implementation, branching over the local inputs of the process and a new alphabet ι_p where $\iota \in C_p$ is the variable that represents information class ι.

Definition 29 (Safety hyper implementation)**.** *Let p and q be components, e be the environment, and C_p be a set of information classes. A safety hyper implementation $\mathcal{H}_p$ of p is a 2^{O_p}-labelled $C_p \cup 2^{I_p}$-transition system.*

The remainder of this section is dedicated to the definition of specifications that, if synthesized, produce safety hyper implementations that satisfy the process's goal. The assume specification, which we now introduce, is satisfied by the information recipient and abstracts the actual information flow. It combines two intentions: It is a trace property and not a hyperproperty, and, equally importantly, it restricts access to information to the time point it will be present, not earlier.

Definition 30 (Assume specification)**.** *Let φ_p be the component specification and C be the finite set of information classes, where each $\iota \in C$ is a subset of $(2^{O_e})^*$. The trace property $\mathbb{A} \subseteq (C \cup 2^{O_p})^\omega$ is defined as*

$$\mathbb{A}_p^C = \{t_C \cup t_o \mid t_C \in C^\omega, t_o \in (2^{O_p})^\omega, \forall n \in \mathbb{N}. \forall \iota \in t_C[n-1]. \\ \forall t_e[0 \dots n-1] \in \iota.\ \textit{if } t_\iota \textit{ is consistent, then } t_e \sqcup t_o[0 \dots n] \vDash_n \varphi_p\},$$

where a finite prefix $t_C \in C^n$ is consistent if it holds that for all $0 \leq m < n$, all finite traces in $t_e[0 \dots m]$ have a prefix in $t_e[0 \dots m-1]$.

For a sequence of information classes, the assume specifications collect all process outputs that are correct for all environment outputs in this information class. Similar to

the synthesis approach in Section 3.3.2, we are mapping outputs to information classes, but in a much more precise way. In every step, the process must react correctly to the *current* information class on the input alphabet. This definitely contains sequences of information classes that will not be present in the system's final implementation; the transmitter will decide how to communicate the information. However, the recipient will provide a correct solution for every possible scenario. The consistency criterion requires no traces to have the same output if they have been distinguishable at $n-1$ but are indistinguishable at n; it is easy to see that this is impossible in actual implementations. → p. 59

Lemma 3. *Let C_p be the finite set of information classes for component p. Every implementation satisfying the assume specification $\mathbb{A}_p^C$ also satisfies the information class specification $\mathbb{C}_p^n$ for all $n \in \mathbb{N}$ and $\iota \in C_p$.*

Lemma 3 follows immediately from the definition of the assume specification. It unites all information class specifications that are implied by the set of information classes. Note that we specify the correctness only with respect to the information classes, not the prefix information-flow assumption, which is a hyperproperty. If some distinguished traces do not occur in some information class, then the receiver will not be correct for all implementations of the sender; the assumed specification would not be sufficient.

After defining a trace property for the assumption of information and the recipient's correct reaction to it, we continue by defining a trace property that guarantees the flow of information. This yields another reduction from hyperproperties to trace properties, as the sender's synthesis problem is the combination of his own local specification and the recipient's information-flow assumption. However, there is a subtle difference between the two problems: While the receiver has to be correct for every implementation of the information-flow assumption, the sender can *choose* the strategy of sharing the information. We use this observation to design certain communication strategies for transmitters that simplify the synthesis problem immensely. Our first step is to classify a given specification as a guarantee specification if all of its realizations, i.e., the results of the synthesis process, satisfy the information-flow assumption.

Definition 31 (Guarantee specification)**.** *Let p and q be components and $\mathcal{P}_{\varphi_p}$ be the prefix information-flow assumption for φ_p. The set $\mathbb{G}_{\rho_p} \subseteq (2^{I_q \cup O_q})^\omega$ is a guarantee specification if all 2^{O_q}-labeled 2^{I_q}-transition systems that satisfy $\mathbb{G}_{\rho_p}$ also satisfy $\mathcal{P}_{\varphi_p}$.*

Guarantee specifications are strictly stronger requirements than information-flow assumptions. This breaks one of the intentions of the information-flow assumptions;

the requirement that we only impose *necessary* conditions on the sender. A direct implication is the fact that even though there is a solution to the distributed synthesis problem at hand, a guarantee specification might not be realizable. This happens if the guarantee specification, in conjunction with the goal specification, builds an unrealizable problem. For example, assume that the guarantee specification $\mathbb{G}_{\rho_p}$ states that the message must be sent at every even time step, while the goal specification states that the communication channel must be empty at every even step. This contradiction could be avoided by transmitting the information at every odd time step.

It is left to show that we can effectively construct a guarantee specification from the prefix information-flow assumption. In the following, we will present two variants of guarantee specifications that we found to be useful in practice. The first one is called the *full-information specification*, which, as the name suggests, requires that the sender copies all the information that is provided by the environment immediately to the receiver. The concept of full-information was already introduced in [184] and was extended in [117].

Definition 32 (Full-information specification)**.** *Let p and q be components, and $g : O_e \cap I_q \to O_q \cap I_p$ be a bijection. The* full-information specification *for q is the trace property*

$$\begin{aligned}\mathbb{F}_p = \{t_e \sqcup t_o \mid t_e \in (2^{O_e \cap I_q})^\omega, t_p \in (2^{O_q \cap I_p})^\omega, \forall v \in (O_e \cap I_q),\\ \textit{either } \forall n \in \mathbb{N}. v \in t_e[n] \textit{ iff } g(v) \in t_o[n]\\ \textit{or } \forall n \in \mathbb{N}. v \in t_e[n] \textit{ iff } g(v) \notin t_o[n]\}\end{aligned}$$

The two conditions on the output trace specify two possible ways of communicating the values of the environment output. Either always the same value is communicated on the communication bits, or always the negation. Note that, as we defined the semantics of the distributed system, the process can react to its inputs in the same time step; the requirement to have the reaction immediately at position n is feasible. We can easily follow that every full-information specification is a guarantee specification.

Lemma 4. *The full-information $\mathbb{F}_p$ specification is a guarantee specification.*

Clearly, if every environment output is communicated, which exactly coincides with all information that is present in a system, then the information-flow assumption is satisfied which is *weaker* requirement. We proceed by defining a weaker and more suitable guarantee specification that is based on the same abstraction of information that the assume specification builds on. Instead of transmitting all information, the sender is required to send the current information class of the trace's current prefix. Assuming we

have a finite set of information classes, we require a consistent, i.e., the same encoding for the whole trace, transfer of the information classes to the communication variables.

Definition 33 (Information Class Guarantee)**.** *Let C'_p be the finite set of information classes of p projected to the inputs of q, s.t. the information classes $\iota \in C'_p$ are subsets of $(2^{O_e \cap I_q})^*$. Let furthermore $g : C \to 2^{O_q \cap I_p}$ be a bijection. The information class guarantee $\mathbb{I}_q^C \subseteq (2^{(O_e \cap I_q) \cup (O_q \cap I_p)})^\omega$ is defined as*

$$\mathbb{I}_q^C = \{t_e \cup t_o \mid t_e \in (2^{O_e \cap I_q})^\omega,\ t_o \in (2^{O_q \cap I_p})^\omega, \forall n \in \mathbb{N}, \forall \iota \in C'_p \\ \text{if } t_e[0 \dots n] \in \iota \text{ then } g(\iota) \in t_o[n]\}.$$

We, again, use a function mapping information classes to valuations of communication variables to ensure consistency. The requirement is that for every prefix that is part of some information class ι, the value of ι, as defined by the function, is part of the output of a process. The correctness of the information class guarantee highly depends on the information classes to distinguish as many traces as necessary. We call a set of information classes *sufficient* if no two traces that are distinguished by the prefix distinguishability relation are in the same information class.

Lemma 5. *If a set of information classes C is sufficient to synthesize φ_p, then $\mathbb{I}_q^C$ is a guarantee specification for φ_p.*

The information class guarantee is then *one* reduction from the prefix distinguishability hyperproperty into a trace property. This is not always possible, as the environment can force us to store unboundedly many increasing prefixes, but, for many specifications, the information classes are a valid abstraction of the hyperproperty. We now return to safety hyper implementations and reason about their correctness.

Definition 34 (Correctness of safety hyper implementation)**.** *Let p, q, and e be the components of a distributed system and the environment, φ_p, φ_q be the local specifications, and C be a sufficient set of information classes for φ_p. A* safety hyper implementation $\mathcal{H}_p$ *is correct if it implements $\mathbb{A}_{\varphi_p}^C$ and some $\mathbb{G}_{\varphi_q}$.*

We require that a safety hyper implementation correctly assumes the information will arrive in the given information classes and that it transfers the information in some way. There is a caveat to defining correctness in this regard: The dependence on some explicit information class is a lot less general than the information-flow assumptions and, therefore, not necessary. The transmitter of information *must* at least follow the same information classes or some refinement of them. However, in the algorithms that

we construct next, the information classes are constructed automata-based, which can be done for both the sender and the receiver.

3.5.2 Synthesis with Unbounded Communication

We present automata-based algorithms to construct information classes, assume specifications, and guarantee specifications. We also show how to synthesize the prefix hyper implementation, how to compute the composition of two safety hyper implementations, and how to obtain the implementation of the distributed system afterward. An overview of the overall synthesis approach is shown in Figure 3.9 and consists of all the previously mentioned steps. → p. 85

The starting point for the synthesis algorithm is computing a set of information classes for each process. Definition 27 already shows that there are, in theory, unboundedly many information classes for LTL specifications. Most precisely, because of the infinite branching, every time step $n + 1$ has more information classes than time step n. We show that we can group the information classes by the *output that has to be chosen* in the time step, and therefore forget the explicit past of the time step. This, and the fact that we use finite automata that accept traces of an arbitrary length, yield a finite representation of the information classes. The classes will not be unique, as traces that must not be distinguished from any other in some time step will be added to some *arbitrary* information class, depending on the order of exploration in the automaton. → p. 77

Algorithm 2 iteratively builds automata that accept the traces of information classes. The starting point is the automaton for the prefix distinguishability relation (see Definition 25) $\mathcal{A}_{\rho_p}$ for process p over tuples of finite traces $\Sigma \times \Sigma$. By choosing single left-hand side traces of the 2-tape automaton and adding all traces that can be indistinguishable from the specific trace, the algorithm iteratively removes traces from $\mathcal{A}_\rho$ until it is empty. In more detail, the function acceptingTrace in Line 4 returns an automaton $\mathcal{A}_v$ that accepts exactly one input trace over the alphabet Σ that is part of the automaton. It chooses one explicit proposition combination for each (randomly chosen) edge in the automaton to some accepting state. Note that this can arbitrarily loop in the automaton. Afterward, Line 5 collects all traces in the negation of the prefix distinguishability automaton for which the candidate is a proper left-hand side trace. Those are all prefixes that are not related to the candidate in the prefix distinguishability. Before continuing with the iteration, all traces that are now in the information class are removed from the left-hand side of the prefix distinguishability automaton, as we already computed the information class this trace belongs to. The algorithm terminates when the prefix distinguishability automaton is empty, meaning that all traces are part of some infor- → p. 71

Algorithm 2: Prefix Information Classes for Process p

```
1 let prefixInformationClasses(A_{ρ_p}):= // Automaton for the prefix disting. relation
2   let infClasses = []
3   while L(A_{ρ_p}) ≠ ∅ do // Not all information classes computed
4     let A_v = acceptingTrace(A_{ρ_p})|_l // Obtain one lhs trace
5     let A_ι = ((A_v ⊕ A_⊤) ∩ complement(A_{ρ_p}))|_r
6     infClasses.append (A_ι) // Add indistinguishable traces
7     A_{ρ_p} = A_{ρ_p} ∩ (complement(A_v) ⊕ A_⊤) // Remove traces of the information class
8   return infClasses
```

mation class. It is important to note that, depending on the choice and order of traces, this algorithm groups different traces into different information classes. Assume, for example, a trace that must not be distinguished from *any* other trace. This trace will be added to the information class of the first trace that is chosen in the algorithm, and to no other information class. This is valid. It will carry the same information as the information class, but it could also be part of any other information class computed later. Therefore, it is crucial that the same set of information classes is used for the assume and guarantee specifications. Furthermore, we note that whenever we choose a trace as a candidate, this trace will also be part of the information class, even though we only obtain the right-hand side of the automaton in Line 5 — every trace is related to itself in the complement of the prefix distinguishability relation.

We transform the n information class automata $C_1, \ldots, C_n$ into the assume specification and the guarantee specifications. The following construction first transforms each automaton into the information class specification and then builds the assume specification from it.

Construction 6. We transform the n finite-word automata $C_1, \ldots, C_n$ that accept all finite words in the language of the information classes to automata representing the information class specification (see Definition 31). For every information class automaton, we → p. 79 construct $\mathcal{B}_i$ as the intersection of C_i and the automaton for the goal specification $\mathcal{A}_\varphi$. The automaton $\mathcal{B}_i$ accepts all traces in the information class with outputs as specified by φ. Note that $\mathcal{B}_i$ is an infinite word automaton, as specified by the goal specification φ. In the next step, we add all input and output combinations on which information class does not restrict the behavior, which is $\overline{C_i}$'s environment inputs combined with $\mathcal{A}_\top$ over the outputs of the process. These automata are called $\mathcal{D}_i$. The assume specification is then all traces in the intersection of all $\mathcal{D}_i$.

This automaton serves as the basis for synthesizing the receiving component of a process. In the complete synthesis approach, an additional constraint is imposed to ensure the guarantee specification of the other process, which can be incorporated as an extra requirement on the process behavior. Before presenting the synthesis algorithm, we describe the construction of an automaton that accepts the full-information guarantee for a process.

Construction 7. Let $I = I_q \cap O_e$ be the inputs observed by q and $O = O_q \cap I_p$. We assume that the number of inputs is smaller than or equal to the number of q's outputs; otherwise, we cannot transmit all information. Let $g : I \to O$ be an arbitrary bijection mapping input variables to output variables. The LTL formula

$$\varphi = \bigwedge_{i \in I} \square(i \leftrightarrow \bigcirc f(i)) \vee \square(i \leftrightarrow \bigcirc \neg g(i))$$

enforces that, for every $i \in I$, either the value of i is always written to $g(i)$ or the complement of i's value is written to $g(i)$. It remains to build an automaton for this formula, which is a standard translation from LTL to NBA [207]. The resulting automaton accepts a full-information guarantee specification.

The final step in the compositional synthesis algorithm involves composing and decomposing the hyper implementations to derive the local implementations of the components, and consequently, the distributed system's implementation. During this process, the information class variables must be replaced with the actual variables and their respective values, as chosen by the synthesis procedure. The composition may include unreachable states, which are inconsistent combinations of information classes and environment outputs (i.e., the finite environment output trace does not belong to the information class). These unreachable states are filtered out as described in Definition 36. The composition is formally defined as follows: → p. 85

Definition 35 (Composition)**.** *Let p, q be components and $\mathcal{H}_p = (S^p, s_o^p, \tau^p, o^p)$ and $\mathcal{H}_q = (S^q, s_0^q, \tau^q, o^q)$ be their respective safety hyper implementations. The composition $\mathcal{H} = \mathcal{H}_p || \mathcal{H}_q$ is a $2^{O_p \cup O_q}$-labeled $2^{O_e} \cup C_p \cup C_q$-transition system (S, s_p, τ, o), where $S = S^p \times S^q$, $s_0 = (s_0^p, s_0^q)$, $o((s^p, s^q)) = o^p(s^p) \cup o^q(s^p)$, and*

$$\tau((s^p, s^q), x) = \tau^p(s^p, (x \cup o^q(s^q)) \cap (I_p \cup C_p)), \tau^q(s^q, (x \cup o^p(s^p)) \cap (I_q \cup C_q))$$

The state space is the cross product of the state space of the hyper implementations, and the labeling function is the union of the local hyper implementations' outputs. The transition function ensures that the global inputs over $2^{O_e} \cup C_p \cup C_q$ are distributed

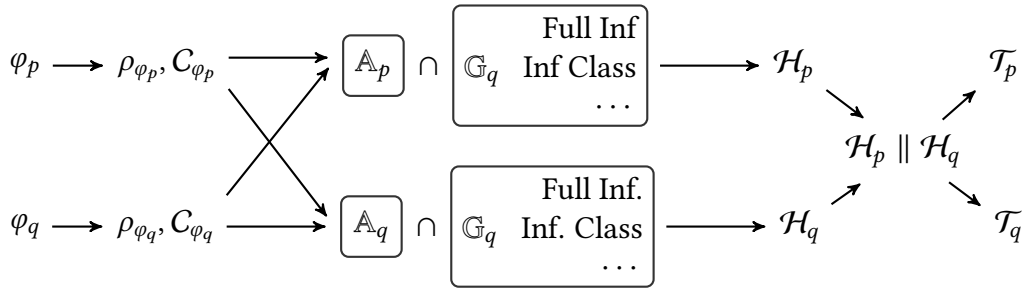


Figure 3.9 An overview of the compositional synthesis algorithm with prefix assumption for processes p and q. The first step consists of computing the prefix distinguishability relation and the information classes of each process. Afterward, automata for the assume specification and some of the possible guarantee specifications are built. After the hyper implementations are synthesized, one obtains the local implementation via composition and decomposition.

into the inputs of the hyper implementations, namely $C_p \cup I_p$ and $C_q \cup I_q$. For every state in the cross-product, the composition branches for every environment output and information class to a local state of a component. Some of these states are, however, unreachable. For our running example, the composition includes a transition with $\neg in_b, \iota'$, although the trace with $\neg in_b$ in the last step cannot be in ι'. We now filter states according to such inconsistencies. We consider $\mathcal{H}(x)$ as the hyper implementation $\mathcal{H}$ terminating in x.

Definition 36 (Filter). *Let $\mathcal{H} = (S, s_0, \tau, o)$ be the composition of the 2^{O_p}-labeled $2^{C_p \cup I_p}$-transition system $\mathcal{H}_p$ and the 2^{O_q}-labeled $2^{C_q \cup I_q}$-transition system $\mathcal{H}_q$. The consistent composition of $\mathcal{H}_p$ and $\mathcal{H}_q$ is the hyper implementation $\mathcal{H}' = (S', s'_0, \tau', o')$, with $S' = S$, $s'_0 = s_0$, $o' = o$, and*

$$\tau'((s^p, s^q), x) = \begin{cases} \tau((s^p, s^q), x) & \text{if } \forall \iota \in x. \mathcal{H}(s^p, s^q) \subseteq \mathcal{L}(\mathcal{F}_\iota) \\ \emptyset & \text{else} \end{cases}$$

A finite trace t of length n over $2^{O_e} \cup C_p \cup C_q$ is considered unreachable if ι appears in $t[n]$, but $t\ |_{O_e}$ does not belong to the information class represented by ι. Determining whether a state is unreachable involves checking language inclusion between the subsystem terminating in that state and the automaton of the corresponding information class, which is a costly task. However, consistency can be enforced more efficiently by monitoring the current information class of a state during a forward traversal of the composed hyper implementations. In the final step, the decomposition projects the composition onto the *observable* outputs of a component. For a given input com-

bination, this results in a set of reachable states, from which one is selected during the decomposition. Note that all such states are valid successors for the given input combination.

Definition 37 (Decomposition)**.** *Let $\mathcal{H} = (S, s_p, \tau, o)$ be the consistent composition of the 2^{O_p}-labeled $2^{C_p \cup I_p}$-transition system $\mathcal{H}_p$ and the 2^{O_q}-labeled $2^{C_q \cup I_q}$-transition system $\mathcal{H}_q$. Furthermore, let min be a function returning the minimal element for a subset of S w.r.t. a total ordering over the states of S. The decomposition $\mathcal{H}|_p$ is a 2^{O_p}-labelled 2^{I_p}-transition system $(S^p, s_0^p, \tau^p, o^p)$ where $S^p = S$, $s_0^p = s_0$, $o^p((s^p, s^q)) = o((s^p, s^q)) \cap O_p$, and*

$$\tau^p(s, x) = min\{s' \mid \exists y \in 2^{(O_e \cup C_p) \setminus I_p}.\ s' = \tau(s, x \cup y)\}$$

The overall synthesis procedure is displayed in Figure 3.9 The synthesis process begins by computing the prefix distinguishability NFAs for the given local specifications. These NFAs are then used to construct the assume and guarantee specifications for both components, which serve as inputs to the local synthesis procedures. The guarantee specification can represent any valid strategy that satisfies the information-flow assumption, such as a specific scheduling paradigm. Next, the automata for the assume and guarantee specifications are intersected. Each component must simultaneously satisfy both the assume and guarantee specifications, which is achieved through trace property synthesis on the intersected automata. If the synthesis problem is unrealizable, it may indicate that the prefix information-flow assumption is insufficient (e.g., additional behavioral assumptions are required) or that the necessary information cannot be communicated to the receiver. Finally, after performing composition, enforcing consistency, and applying decomposition, the algorithm produces two local implementations. Together, these implementations form a distributed system that satisfies both the original specifications.

Corollary 2. *Let p and q be components with local specifications φ_p and φ_q. The distributed system implementation returned by the algorithm depicted in Figure 3.9 satisfies the local specifications.*

3.5.3 Experiments

We implemented the compositional synthesis algorithm described so far in our prototype called FlowSy. The implementation builds on the popular infinite word automaton manipulation tool spot [4] for translation, conversion, and emptiness checking of NBAs. FlowSy implements the support for the finite automata, the construction of prefix distinguishability in Construction 3.4.2, the construction of the information classes in

Algorithm 2, and building automata for the assume specification in Construction 6 and full information specification Construction 7. The synthesis of the hyper implementations is performed by converting the Büchi automata to deterministic parity games and solving them with the solver OINK [72]. We report on two research questions: (1) how do the prefix distinguishability automaton and the information classes scale with respect to formula size and information flow over time, and (2) how does FLOWSY compare to the existing bounded synthesis approach for distributed system HYPERBOSY presented in [94]. Note that, at the time of evaluation, HYPERBOSY was the only tool for distributed synthesis that we were able to compare against. A comparison with the information flow guided synthesis algorithm with bounded communication in Section 3.3 is infeasible since the languages of goal specifications are disjoint. All experiments are run on a 2.8 GHz processor with 16 GB RAM. The timeout was 600 seconds, and the results are shown in Table 3.2.

→ p. 83

We use benchmarks that scale in three dimensions: The number of independent variables, time steps between information reception and corresponding output, and combinatorics over input and output variables.

Sequence Transmission. The first one specifies the communication of n input variables in *sequence transmission*. This parametric version of this chapter's running example has n subformulas that are conjunctively connected.

$$\varphi_b = \bigwedge_{1 \leq i \leq n} \square(in_{\mathsf{a}}^i \leftrightarrow \bigcirc out_{\mathsf{b}}^i).$$

Delay. In the *delay* benchmark, the number of input and output variables is constant, but the number of time steps in between reception and response is parameterized.

$$\varphi_b = \square(in_{\mathsf{a}} \leftrightarrow \bigcirc^n out_{\mathsf{b}}).$$

The parameter n specifies the number of $\bigcirc$ operators that precede the output variable out_{b}.

Conjunctions and disjunctions. The final two benchmarks are Boolean combinations of input and output variables. The *conjunctions* benchmark enforces that the conjunctions over the inputs are mirrored in the outputs.

$$\varphi_b = \square(\bigwedge_{1 \leq i \leq n} in_{\mathsf{a}}^i \leftrightarrow \bigcirc \bigwedge_{1 \leq j \leq n} out_{\mathsf{b}}^j)$$

Benchmark	Parameter	$\lvert\varphi\rvert$	$\lvert\rho\rvert$	$\lvert C\rvert$	FlowSy	HyperBosy [94]
Delay	1	5	4	2	1.74	0.97
	2	6	8	2	**1.87**	TO
	3	7	16	2	**1.84**	TO
	4	8	32	2	**1.94**	TO
	5	9	64	2	**2.36**	TO
Sequence Transmission	1	5	4	2	1.83	**1.42**
	2	11	6	4	**5.28**	TO
	3	16	10	8	**36.81**	TO
Conjunctions	1	5	4	2	3.18	**0.92**
	2	9	4	4	**4.35**	91.80
	3	13	4	8	**9.20**	TO
	4	17	4	16	TO	TO
Disjunctions	1	5	4	2	**3.25**	6.26
	2	9	4	4	**5.63**	60.08
	3	13	4	8	**12.14**	TO
	4	17	4	16	TO	TO

Table 3.2 This table presents the experimental results. The "Benchmark" and "Par." columns indicate the benchmark type and its parameter value, respectively. The columns $|\varphi|$, $|\rho|$, and $|C|$ represent the size of the formula, the number of states in the prefix distinguishability automaton, and the number of computed information classes, respectively. The final two columns show the runtime of FlowSy and HyperBosy in seconds.

The *disjunctions* benchmarks have the same structure but with conjunctions replaced by disjunctions.

$$\varphi_b = \square(\bigvee_{1\leq i\leq n} in_{\mathsf{a}}^i \leftrightarrow \bigcirc \bigvee_{1\leq j\leq n} out_{\mathsf{b}}^j)$$

We summarize the results of our experiments with regard to scaling and the comparison with HyperBosy.

Scaling. FlowSy scales primarily with the number of computed information classes. Interestingly, in the *delay* benchmark, the number of information classes remains constant at 2, even though the size of the prefix distinguishability automaton grows exponentially. This is because the automaton for the information class only checks that the current position matches the position n steps earlier, which can be represented by just two information classes. For the conjunction and disjunction benchmarks, the situation is reversed. While the prefix distinguishability automaton remains constant,

the number of information classes grows exponentially with the parameter, as it must account for all possible combinations of input variables. In the sequence transmission benchmark, all reported values scale with the input parameter, leading to a predictable increase in runtime until a timeout occurs at step 4 (not shown in Table 3.2).

Comparison to HyperBosy. FlowSy demonstrates a clear advantage over HyperBosy in terms of scalability. This is particularly observable in the delay benchmark, where FlowSy maintains an almost constant runtime. Since the number of information classes remains unchanged, the synthesis process primarily scales with the transmission of information. In contrast, HyperBosy must store all values for $\bigcirc^n$ steps during synthesis, which rapidly inflates the search space to an unmanageable size. For the conjunction and disjunction benchmarks, although the number of information classes grows exponentially, FlowSy significantly outperforms HyperBosy. For instance, HyperBosy already requires 91 seconds for parameter 2 in the conjunction benchmark, whereas FlowSy completes the task much faster. In summary, the compositional approach of FlowSy consistently proves advantageous, reducing execution time, especially when complex communication in distributed systems can be abstracted into a small number of information classes.

Chapter 4

Controller Synthesis with Prophecies

In this chapter, we present a compositional synthesis procedure for a restricted class of distributed systems: The synthesis of controllers. Here, the system consists of a controller, whose implementation is the result of the synthesis procedure, and a given implementation of a plant. The controller and plant together must satisfy a target specification, a temporal logic specification, against every possible environment behavior. The conventional approach constructs a model of the plant [152, 209] and composes it with an automaton for the temporal specification before applying a synthesis algorithm. This yields an ω-regular game that can be solved using established techniques from reactive synthesis [2, 142, 143]. However, this approach does not scale well: Modeling all possible plant behaviors in an automaton makes the resulting game expensive to solve. We propose a new algorithm for synthesizing controllers via the construction of a *universal* controller instead. A universal controller is a generalized controller that is correct for every plant for which a solution exists, and it avoids representing the plant's explicit state space.

The key idea is that one can avoid exploring the plant's state space by solving the controller synthesis problem *abstractly* from the specification alone. We do this by introducing assumptions about the plant's future behavior, which we call *prophecies*. To motivate prophecies, consider reactive synthesis: Synthesis must build sufficient memory to ensure satisfaction of the specification, since obligations that depend on past or future behavior must be tracked. The memory required determines the required size of the implementation's state space. In controller synthesis, an additional complication arises: The plant's state and memory may affect what the controller must remember. The plant's behavior, therefore, *restricts* the controller's correct actions. Prophecies, abstract conditions on the future structure of the plant, let us avoid explicitly exploring

the plant's state space by computing the semantic criterion on the future of the plant for certain moves.

Consider, for example, the controller be c, the plant p, and the external environment e. Each process controls a Boolean output: o_{c}, o_{p}, and o_{e}. At each step, every process chooses its output based on the history of all outputs. The temporal requirement is

$$\varphi \coloneqq (o_{\mathsf{c}} \leftrightarrow o_{\mathsf{p}})\, \mathcal{W}(\bigcirc \neg o_{\mathsf{e}}). \tag{4.1}$$

This formula requires the controller to match the plant's output while the environment's output remains true (using weak until $\mathcal{W}$). Since the controller cannot observe the plant's current output before making its choice, synthesizing a controller independently of the plant is impossible: Unless the environment eventually drives o_{e} to false, the controller must anticipate the plant's output one step ahead. We express this anticipation via prophecies: $\langle o_{\mathsf{p}} \rangle$ means "the plant will output true" and $\langle \neg o_{\mathsf{p}} \rangle$ means "the plant will output false". A correct controller reacts to $\langle o_{\mathsf{p}} \rangle$ by setting o_{c} to true, and to $\langle \neg o_{\mathsf{p}} \rangle$ by setting o_{c} to false.

Prophecies are semantic artifacts that make it possible to synthesize a *universal controller* whose choices depend on conditions over the plant's future. Based on this idea, we develop a synthesis method that separates the construction of a universal controller, obtained solely from the temporal specification, from the later step of adapting it to a concrete plant model. In the universal controller, each output choice is guarded by a prophecy: The controller only commits to an output when the corresponding prophecy about the plant holds. This separation permits synthesizing the universal controller without access to the plant, greatly improving scalability.

We give an algorithm to synthesize universal controllers for safety specifications, encoding prophecies with tree automata. We prove that universal controllers are *most permissive*: They capture all controller strategies that are correct for some realizable plant. We also provide an exploration algorithm that, given a plant model, refines a universal controller into a concrete controller; for safety specifications this refinement is sound and complete. Finally, we report experiments with our prototype tool, UNICON [107], demonstrating that our method scales better than the standard plant-composition approach on benchmarks with large plant models.

Outline. Section 4.1.3 introduces the concept of controllers with prophecy annotations and the characterization of universal controllers. For safety LTL specification, Section 4.1.4 shows an automata-based approach for computing appropriate prophecies. Section 4.2 presents the synthesis algorithms for controllers based on prophecy

annotations, including the computation of universal controllers, refinement to specific plant behavior, and experimental evaluation.

Publications. The results on universal controller synthesis presented here build on [99]. This thesis extends those findings with additional examples and further theoretical developments.

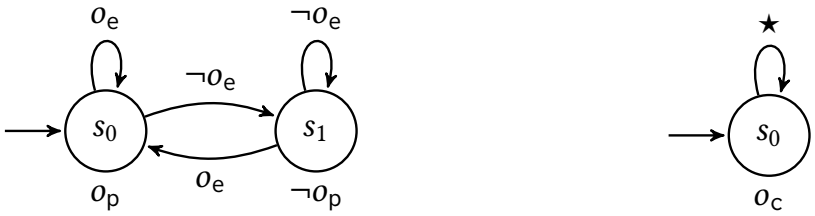
[99] Bernd Finkbeiner, Niklas Metzger, Satya Prakash Nayak, Anne-Kathrin Schmuck. **Synthesis of Universal Safety Controllers**. *31st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2025)*

4.1 Universal Controllers

We now turn to the controller synthesis problem. In the previous sections, we synthesized solutions for distributed systems consisting of two components, where solutions consisted of two implementations. In controller synthesis, one component is already fixed, namely the *plant*, and we synthesize a controller that, together with the plant, is correct for every environment output. We introduce the notion of universal controllers, controllers that are correct for all plants, and prophecies, semantic conditions on the branching structure of the plant for outputs of the universal controller. We describe an algorithm for synthesizing universal controllers from safety specifications, using a finite representation of prophecies as tree automata. Additionally, we demonstrate that universal controllers are *most permissive*, encompassing all controller strategies that are valid for any realizable plant model. Such universal controllers are also *correct*, as they contain a correct explicit controller implementation for each plant.

4.1.1 Controller Synthesis

Logical controller synthesis is the problem of synthesizing a controller for a plant in a distributed system. The input to the problem is a special case of the distributed synthesis problem where an architecture is built over an unknown controller c, a plant p whose implementation $\mathcal{T}_\mathsf{p}$ is provided, and the usual environment e. For an architecture that defines the observable variables of the different components and an LTL specification φ, the output of a controller synthesis procedure is a controller implementation $\mathcal{T}_\mathsf{c}$ such that the composition of controller and plant satisfies the specification, or the answer *unrealizable* if no such controller exists. We use the following example specification throughout this part of the thesis.



(a) A plant. (b) A controller.

Figure 4.1 The plant and controller of the running example of this part of the thesis.

Example 28. Consider the following specification: At every time step, each process determines the values of its output propositions based on the history of all prior outputs. The temporal specification is expressed as the LTL formula shown in Equation 4.1. The formula specifies that the controller's output matches the plant's output as long as the environment's output remains true. However, since the controller cannot observe the plant's output before determining its own, it is impossible to construct a correct controller independently of the plant. Specifically, unless the environment sets its output to false, the controller must predict the plant's output one time step ahead. Consider the plant as given in Figure 4.1a that outputs o_p as *true* whenever the environment outputs *true* on the variable o_e. A correct controller for this plant must output o_c as *true* as long as the environment outputs *true* on the variable o_e. After the first time, the output of the environment is satisfied, and the specification is unconditionally satisfied. A correct explicit controller is presented in Figure 4.1b, which totally ignores the environment's output but sets its own output equivalently to the plant. Note that this controller is correct for any plant that sets o_p as long as o_e holds, independent of what the plant does whenever o_e does not hold.

The standard algorithm to solve controller synthesis is computing the deterministic automaton for the specification φ, building the composition of the specification automaton and plant, and then computing the implementation by solving a parity game of environment against system over the product [168]. Building the deterministic automaton ensures that the formula can be satisfied, and building the product with the plant ensures that the controller can correctly react to the plant's behavior. In contrast to reactive synthesis of distributed systems, the plant's state space in the product is crucial for controller synthesis for the computed solution to be correct. As an example, consider an extension of the plant in Figure 4.1a, where every third step the outputs switch from true to false for the environment output being true, and from false to true otherwise. To correctly reply to the plant's behavior, the plant must be able to

distinguish every third state from all the others. Therefore, the standard approach adds the plant's state space to the synthesis problem but remains 2EXPTIME-complete in the size of the LTL formula.

4.1.2 Controllers with Prophecies

We begin by classifying plants into admissible and inadmissible ones. Admissible plants are those for which there exists a controller that satisfies the specification. Inadmissible plants are those for which no controller can satisfy the specification. We are particularly interested in admissible plants, as on the one hand, a controller that satisfies the specification together with the plant exists, and on the other hand, the set of admissible plants builds the space of controller solutions. We define admissible plants formally:

Definition 38 (Admissibility)**.** *Let φ be a specification. A plant $\mathcal{T}_\mathsf{p}$ is called* admissible *for φ if there exists a controller $\mathcal{T}_\mathsf{c}$ s.t. $\mathcal{T}_\mathsf{c} \parallel \mathcal{T}_\mathsf{p} \vDash \varphi$.*

Recall that the *realizability* question in *reactive* synthesis is, for a given LTL specification, whether there exists an implementation that satisfies the specification. In controller synthesis, realizability works differently. For a given specification *and* a plant, the problem is realizable if there exists a controller that satisfies the specification. More concisely, the problem is realizable if the specification is realizable in reactive synthesis and the plant is admissible for this specification. As an implication, one can infer that for every unrealizable specification, the set of admissible plants is empty.

For a given plant implementation $\mathcal{T}_\mathsf{p}$, the standard controller synthesis method examines the entire state space of $\mathcal{T}_\mathsf{p}$ to either construct a controller or determine unrealizability of either the plant or the specification. However, plants are often immensely large, as they encode abstractions of real-world implementations of cyber-physical system dynamics that require numerous states. This results in significant computational overhead, making synthesis algorithms impractical for such cases. The key insight of this chapter is that this exhaustive exploration can be avoided by addressing the controller synthesis problem passively, without relying on a specific plant implementation. This is accomplished by introducing assumptions about the plant, which we refer to as *prophecies*.

Definition 39 (Prophecies)**.** *A prophecy $\theta \subseteq \mathbb{P}$ is a set of plants.*

Prophecies allow the controller to base its current output on predictions about the plant's future behavior. To clarify, consider the controller synthesis problem from

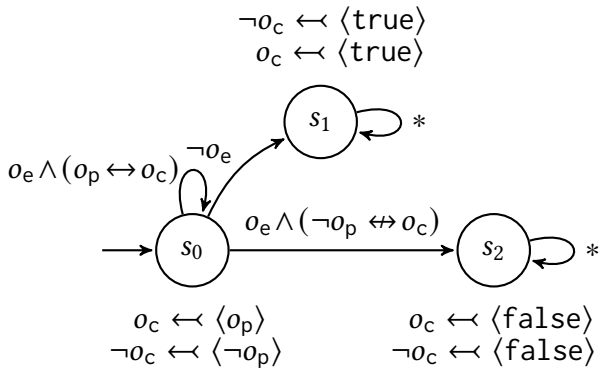


Figure 4.2 A controller with prophecies W with prophecy annotation κ (denoted by $\leftarrow\!\!\prec$). The annotations are $\kappa(s_1, *) = \langle \texttt{true} \rangle$ for the state s_1, $\kappa(s_2, *) = \langle \texttt{false} \rangle$ for state s_2, and $\kappa(s_0, o_c) = \langle o_p \rangle$ as well as $\kappa(s_0, \neg o_c) = \langle \neg o_p \rangle$ for the initial state. $\langle \texttt{true} \rangle$ denotes the set of all plants, $\langle \texttt{false} \rangle$ the set of no plants, and $\langle o_p \rangle$ and $\langle \neg o_p \rangle$ the sets of plants that output o_p and $\neg o_p$ in the current time step, respectively.

Figure 4.1. The controller must output o_c in the first time step if the plant outputs o_p in the same step; otherwise, it must output $\neg o_c$. However, the controller cannot observe the plant's output before deciding its own. Therefore, it relies on a prophecy about the plant's behavior to determine its output, namely whether the plant outputs o_p or $\neg o_p$. Plant strategies can be highly complex, involving complex branching and a high number of propositions. By synthesizing a controller that conditions its output on a *prophecy*, we isolate the relevant information about the plant's future needed for controller synthesis, independent of the plant's specific implementation. When a concrete plant implementation is provided, the controller verifies all prophecies to determine the correct output at each time step. This approach results in what we call a *controller with prophecies*, as formally defined below.[1]

→ p. 93

Definition 40 (Controllers with prophecies)**.** *A* controller with prophecies *(or short prophecy controller) W over alphabet Σ and prophecy set $\mathbb{F}$ is a tuple (S, s_0, τ, κ) consisting of a finite set of states S, an initial state $s_0 \in S$, a transition function $\tau : S \times \Sigma \to S$, and a prophecy annotation $\kappa : S \times 2^{O_c} \to \mathbb{F}$.*

Note that the transition function of a controller with prophecies operates over the entire alphabet Σ, rather than being restricted to the controller's inputs. Additionally, each state is associated with a *condition* derived from a prophecy for every possible controller output. When a specific plant is provided, these prophecies can be validated

[1]Note that we used the term *universal controllers* for controllers with prophecies in [99]. For clarification, we differentiate between prophecy controllers and universal controllers in the following.

to determine the appropriate controller output for each state, thereby defining the controller's labeling function. Once the labeling function is established, the transition function is refined to depend solely on the inputs.

Example 29. We revisit the controller synthesis problem from Figure 4.1. Consider the controller with prophecies W depicted in Figure 4.2. In the initial time step, since the environment's future behavior is unknown (recall that this is a Moore machine), the controller must set o_c to *true* if the plant also sets o_p to *true*. Thus, the prophecy for the output o_c at the initial state s_0 in Figure 4.2, i.e., $\kappa(s_0, o_c) = \langle o_p \rangle$, represents the set of all plants that output o_p in the current time step. Similarly, the prophecy for the output $\neg o_c$ at state s_0, i.e., $\kappa(s_0, \neg o_c) = \langle \neg o_p \rangle$, represents the set of all plants that output $\neg o_p$ in the current time step. Furthermore, if the environment outputs $\neg o_e$, the system transitions to state s_1, where the controller can choose any output since the specification is satisfied regardless of the controller's actions. The prophecy for both outputs o_c and $\neg o_c$ at state s_1 is $\langle \texttt{true} \rangle$, indicating that any output is valid for any plant. Conversely, if the environment outputs o_e and the plant's and controller's outputs do not match, the system transitions to state s_2, where the specification is violated regardless of the controller's actions. Hence, the prophecy for both outputs o_c and $\neg o_c$ at state s_2 is $\langle \texttt{false} \rangle$, indicating that no output is valid for any plant.

→ p. 93

4.1.3 Universal Controllers

Controllers with prophecies contain prophecies over plant behavior, mapping the controller's output to a set of plants. These prophecy annotations can be used to derive explicit implementations of controllers. However, without any additional constraints on both the state space and the prophecies, the controller with prophecies cannot be used for controller synthesis. We add two explicit requirements on controllers with prophecies that qualify them as universal controllers; controllers with prophecies that solve the controller synthesis problem. We begin with the definition of consistency:

Definition 41 (Consistency)**.** *Let $W = (Ss_0, \tau, \kappa)$ be a controller with prophecies and $\mathcal{T}_p$ be a controller. A controller $\mathcal{T}_c = (S^c, s_0^c, \tau^c, o^c)$ is said to be* consistent *with W w.r.t. $\mathcal{T}_p$, denoted by $\mathcal{T}_c \vDash W \parallel \mathcal{T}_p$, if for all $(S, s^p, t^c) \in reach(W \times (\mathcal{T}_p \parallel \mathcal{T}_c))$, it holds that $\mathcal{T}_p(s^p) \in \kappa(S, o^c(S^c))$.*

At each state of the universal controller, the plant's structure starting from the associated plant state must fulfill the prophecy corresponding to the controller's output from that state. Given a prophecy controller W and a plant $\mathcal{T}_p$, an explicit controller $\mathcal{T}_c$ (if

it exists) can be constructed by checking the plant against all prophecies in the current state of the universal controller and selecting a controller output whose corresponding prophecy is satisfied by the plant.

Example 30. Consider the controller with prophecies W depicted in Figure 4.2 and the plant implementation $\mathcal{T}_\mathsf{p}$ illustrated in Figure 4.1a. To construct a controller implementation $\mathcal{T}_\mathsf{c}$ consistent with W w.r.t. $\mathcal{T}_\mathsf{p}$, we verify the prophecies as follows: Starting from the initial state s_0, the plant $\mathcal{T}_\mathsf{p}$ sets o_p to true, satisfying the prophecy $\kappa(s_0, o_\mathsf{c})$. Consequently, a consistent controller must set o_c to *true* in the first time step. As long as the environment outputs o_e, the plant $\mathcal{T}_\mathsf{p}$ continues to set o_p to true. Therefore, a consistent controller must also set o_c to *true* under the same condition. When the environment outputs $\neg o_\mathsf{e}$, the controller with prophecies W transitions to state s_1, where the prophecies for both o_c and $\neg o_\mathsf{c}$ are $\langle \mathtt{true} \rangle$. This allows the controller to choose any output. By selecting the prophecy $\kappa(s_1, \neg o_\mathsf{c})$, the controller $\mathcal{T}_\mathsf{c}$ sets o_c to false for all subsequent time steps. This results in the controller implementation $\mathcal{T}_\mathsf{c}$ shown in Figure 4.1b, which is consistent with W w.r.t. $\mathcal{T}_\mathsf{p}$. Notably, since the controller outputs are determined by verifying the prophecies for the plant $\mathcal{T}_\mathsf{p}$, there is no need to verify prophecies at state s_2 in the controller with prophecies W, where the specification is violated.

→ p. 95

Our primary objective is to identify a controller with prophecies such that, for any admissible plant, the controller implementation consistent with it resolves the logical controller synthesis problem. We refer to such a controller as *correct*, as defined below.

Definition 42 (Correctness)**.** *Let φ be a specification. The prophecy controller W is* correct *for φ if for all plant strategies $\mathcal{T}_\mathsf{p}$, it holds that $\mathcal{T}_\mathsf{c} \vDash W \parallel \mathcal{T}_\mathsf{p}$ implies $\mathcal{T}_\mathsf{p} \parallel \mathcal{T}_\mathsf{c} \vDash \varphi$.*

In addition to correctness, we also require the controller with prophecies to be *permissive*. Permissiveness is a widely used concept in control problems [44, 27, 52, 112], emphasizing that a controller should be correct for as many plants as possible. In the context of logical controller synthesis, we refine this notion: A controller with prophecies is considered *most permissive* if all correct controllers for an admissible plant are consistent with it. This is notably different from correctness: The controller must contain not only one correct controller for each plant but also all correct ones.

Definition 43 (Permissiveness)**.** *Let φ be a specification. The controller with prophecies W is* most permissive *for φ if for all plant strategies $\mathcal{T}_\mathsf{p}$, it holds that $\mathcal{T}_\mathsf{p} \parallel \mathcal{T}_\mathsf{c} \vDash \varphi$ implies $\mathcal{T}_\mathsf{c} \vDash W \parallel \mathcal{T}_\mathsf{p}$.*

Example 31. Consider the controller with prophecies W depicted in Figure 4.2, the plant implementation $\mathcal{T}_\mathsf{p}$ illustrated in Figure 4.1a, and the controller implementation $\mathcal{T}_\mathsf{c}$ shown in Figure 4.1b, which is consistent with W w.r.t. $\mathcal{T}_\mathsf{p}$. The controller with prophecies W satisfies both correctness and permissiveness for the specification φ in our running example. Consequently, the controller implementation $\mathcal{T}_\mathsf{c}$ provides a solution to the logical controller synthesis problem for φ with the given plant implementation $\mathcal{T}_\mathsf{p}$.

We summarize this section by unifying the previous definition for controllers with prophecies. The controller that satisfies both correctness and permissiveness is called a *universal controller*.

Definition 44 (Universal controller)**.** *Let φ be a specification and W be a controller with prophecies. The controller with prophecies W is called a* universal controller *for φ if it is correct and most permissive for φ.*

A universal controller preserves the same solution space as standard controller synthesis but is independent of the plant's state space. We continue with the computation of prophecies for safety specifications.

4.1.4 Prophecies for Controller Synthesis

The computation of a universal controller is equivalent to determining appropriate prophecies. The next section introduces an automata-based approach for deriving prophecies tailored to safety specifications. These prophecies serve as the cornerstone of our universal controller synthesis algorithm. Here, we define a specific category of prophecies, referred to as *safe prophecies*, which are sufficient to construct a universal controller for safety specifications that ensures correctness and most permissiveness.

Prophecies for Safety Specifications. Safe prophecies focus solely on the current and next states of the execution to address the safety aspect of the given specification. Conceptually, for a specific controller output, a safe prophecy identifies a set of plants for which this choice is *safe*, meaning it *can* result in an accepting run within the automaton. The fact that it *can* result in an accepting run, compared to it *will* result in an accepting run, is a crucial distinction that we will later clarify. We formally define these prophecies for a given automaton as follows:

Definition 45. *Let $\mathcal{A} = (Q, q_0, \delta, \Omega)$ be an automaton, $q \in Q$ be a state in $\mathcal{A}$, and $\alpha \in 2^{O_\mathsf{c}}$ be an output of the controller. The* safe prophecy $\mathbb{S}_{\mathcal{A}}^{q,\alpha}$ *is the set*

$$\mathbb{S}_{\mathcal{A}}^{q,\alpha} = \{\mathcal{T}_\mathsf{p} \in \mathbb{P} \mid \exists \mathcal{T}_\mathsf{c} \in \mathbb{C}.\ out(\mathcal{T}_\mathsf{c}, \epsilon) = \alpha \wedge Runs(\mathcal{A}, q, \mathcal{T}_\mathsf{p} \parallel \mathcal{T}_\mathsf{c}) \subseteq \Omega\}.$$

Recall that $out(\mathcal{T}_c, \epsilon)$ refers to the current output of the controller. For a given state $q \in Q$ and a controller output $\alpha \in 2^{O_c}$, the set $\mathbb{S}_{\mathcal{A}}^{q,\alpha}$ comprises all plant strategies $\mathcal{T}_p$ for which there exists a controller implementation $\mathcal{T}_c$ starting from q that aligns with the output α and ensures that all runs of the automaton with $\mathcal{T}_p$ are accepting, i.e., $Runs(\mathcal{A}, q, \mathcal{T}_p \parallel \mathcal{T}_c) \subseteq \Omega$. In essence, the safe prophecy for a controller's output identifies all plant strategies where this choice is guaranteed to be safe. Thus, for any safety LTL formula, safe prophecies impose the necessary constraints on the controller's decisions. We formalize this claim in the following theorem.

Theorem 14. *Let φ be a safety LTL specification and $\mathcal{A} = (Q, q_0, \delta, \Omega = safe(F))$ its equivalent safety automaton. The prophecy controller $W = (Q, q_0, \delta, \kappa)$ with $\kappa(q, \alpha) = \mathbb{S}_{\mathcal{A}}^{q,\alpha}$ for all $q \in Q, \alpha \subseteq 2^{O_c}$ is most permissive for φ.*

Proof. Let $\mathcal{T}_p = (S^p, s_0^p, \tau^p, o^p)$ be an admissible plant and let the controller implementation $\mathcal{T}_c = (S^c, s_0^c, \tau^c, o^c)$ be such that $\mathcal{T}_p \parallel \mathcal{T}_c \vDash \varphi$. We need to show that $\mathcal{T}_c \vDash W \parallel \mathcal{T}_p$. Consider the composition $W' = W \times (\mathcal{T}_p \parallel \mathcal{T}_c) = (S', s_0', \tau')$. Let $s' = (s, s^p, s^c) \in reach(W')$. Then, it is enough to show that $\mathcal{T}_p(S^p) \in \kappa(s, o^c(S^c)) = \mathbb{S}_{\mathcal{A}}^{s,o^c(S^c)}$. It follows that it is enough to show that for implementation $\mathcal{T}_c' = \mathcal{T}_c(S^c)$ and $\mathcal{T}_p' = \mathcal{T}_p(S^p)$, it holds $Runs(\mathcal{A}, q, \mathcal{T}_p' \parallel \mathcal{T}_c') \subseteq \Omega$.

Let ρ be a run in $Runs(\mathcal{A}, q, \mathcal{T}_p' \parallel \mathcal{T}_c')$, then by definition, there exists a run ρ' in W' with $\rho'[i] = (\rho[i], *, *)$ for all $i \geq 0$. Let $\rho^f = (q_0, s_0^p, s_0^c) \dots (q_k, s_k^p, s_k^c)$ be a path in W' from s_0' to s'. Then, by construction, $(\rho^f[0, k-1])\rho'$ is a run from s_0' in W'. As $\mathcal{T}_c \parallel \mathcal{T}_p \vDash \varphi$ and $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\varphi)$, the corresponding run of $(\rho^f[0, k-1])\rho'$ in $\mathcal{A}$ is accepting, i.e., $q_0 q_1 \dots q_{k-1}\rho \in \Omega$. By the definition of safety acceptance, we have $\rho \in \Omega$. As ρ is arbitrary, we have $Runs(\mathcal{A}, q, \mathcal{T}_p' \parallel \mathcal{T}_c') \subseteq \Omega$, and hence, $\mathcal{T}_c \vDash W \parallel \mathcal{T}_p$. □

The theorem states that any controller implementation valid for a given plant is included in the prophecy set. By ensuring the controller avoids unsafe decisions, the controller can satisfy the safety specification. Furthermore, we can establish that safe prophecies are adequate for synthesizing a correct universal controller that fulfills any safety specification when a plant implementation is provided.

Theorem 15. *Let φ be a safety LTL specification and $\mathcal{A} = (Q, q_0, \delta, \Omega = safe(F))$ its equivalent safety automaton. The prophecy controller $W = (Q, q_0, \delta, \kappa)$ with $\kappa(q, \alpha) = \mathbb{S}_{\mathcal{A}}^{q,\alpha}$ for all $q \in Q, \alpha \subseteq 2^{O_c}$ is correct for φ.*

Proof. Let $\mathcal{T}_p = (S^p, s_0^p, \tau^p, o^p)$ be an admissible plant and let the controller implementation $\mathcal{T}_c = (S^c, s_0^c, \tau^c, o^c)$ be consistent with W w.r.t. $\mathcal{T}_p$, i.e., $\mathcal{T}_c \vDash W \parallel \mathcal{T}_p$. We need to show that $\mathcal{T}_p \parallel \mathcal{T}_c \vDash \varphi$. Let $t \in Traces(\mathcal{T}_p \parallel \mathcal{T}_c)$ with $\rho = run(\mathcal{A}, q_0, t)$ being the

corresponding unique run. As $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\varphi)$, to prove $t \in \mathcal{L}(\varphi)$, we only need to show that $\rho \in \Omega$, i.e., $\rho[i] \in F$ for all $i \geq 0$.

Let $\mathcal{T}_\mathsf{p} = (S^\mathsf{p}, s_0^\mathsf{p}, \tau^\mathsf{p}, o^\mathsf{p})$ and $\mathcal{T}_\mathsf{c} = (S^\mathsf{c}, s_0^\mathsf{c}, \tau^\mathsf{c}, o^\mathsf{c})$. Then, as $\mathcal{T}_\mathsf{c} \vDash W \parallel \mathcal{T}_\mathsf{p}$, by definition, for every $(q, s^\mathsf{p}, s^\mathsf{c}) \in \mathit{reach}(W \times (\mathcal{T}_\mathsf{p} \parallel \mathcal{T}_\mathsf{c}))$, it holds that $\mathcal{T}_\mathsf{p}(S^\mathsf{p}) \in \kappa(q, o^\mathsf{c}(S^\mathsf{c})) = \mathbb{S}_\mathcal{A}^{q, o^\mathsf{c}(S^\mathsf{c})}$.

As $t \in \mathit{Traces}(\mathcal{T}_\mathsf{p} \parallel \mathcal{T}_\mathsf{c})$, by construction, there exists a run ρ' from the initial state in $W \times (\mathcal{T}_\mathsf{p} \parallel \mathcal{T}_\mathsf{c})$ corresponding to t such that $\rho'[i] = (\rho[i], s_i^\mathsf{p}, s_i^\mathsf{c})$. Hence, for all $i \geq 0$, we have $\rho'[i] \in \mathit{reach}(W \times (\mathcal{T}_\mathsf{p} \parallel \mathcal{T}_\mathsf{c}))$, and hence, $\mathcal{T}_\mathsf{p}(S_i^\mathsf{p}) \in \mathbb{S}_\mathcal{A}^{\rho[i], o^\mathsf{c}(S_i^\mathsf{c})}$. Then, by definition of safe prophecies, for each $i \geq 0$, there exists $\mathcal{T}_\mathsf{c}^i \in \mathbb{C}$ with $\mathit{out}(\mathcal{T}_\mathsf{c}^i, \epsilon) = o^\mathsf{c}(S_i^\mathsf{c})$ and $\mathit{Runs}(\mathcal{A}, \rho[i], \mathcal{T}_\mathsf{c}^i \parallel \mathcal{T}_\mathsf{p}(S_i^\mathsf{p})) \subseteq \Omega$. This means, for each $i \geq 0$, there exist runs from $\rho[i]$ in $\mathcal{A}$ that are accepting, and hence, by the definition of safety acceptance Ω, we have $\rho[i] \in F$. □

The prophecy controller depicted in Figure 4.2 is annotated with safe prophecies, ensuring both correctness and most permissiveness for the safety specification φ in our running example. It is, indeed, a universal controller. We conclude with a corollary that states that, with prophecies as defined in Definition 45, it is a universal controller, which immediately follows from the previous theorems. → p. 95 → p. 98

Corollary 3. *Let φ be a safety LTL specification and $\mathcal{A} = (Q, q_0, \delta, \Omega = \mathit{safe}(F))$ its equivalent safety automaton. The prophecy controller $W = (Q, q_0, \delta, \kappa)$ with $\kappa(q, \alpha) = \mathbb{S}_\mathcal{A}^{q,\alpha}$ for all $q \in Q, \alpha \subseteq 2^{O_\mathsf{c}}$ is a universal controller for φ.*

Automata for Prophecies. We continue with an automata-based construction of safe prophecies. Prophecies, as introduced in Definition 39 and Definition 45, represent sets of plants that align with a specific controller decision. Tree automata, which recognize sets of trees (as opposed to word automata, which recognize sets of traces), serve as appropriate models for representing prophecies. In the language of such an automaton, each tree corresponds to a plant implementation, which itself is an infinite tree. We demonstrate that it is possible to construct a tree automaton that accurately captures the safe prophecy for a given controller decision. The construction follows a similar approach to the automaton-to-two-player-game transformation used in LTL synthesis [91]. A plant satisfies the prophecy if it provides a valid response to every possible controller input. Furthermore, we constrain the controller's output in the automaton's initial step to match the output for which the prophecy is being computed. This leads to the following result. → p. 94

Theorem 16. *Let $\mathcal{A} = (Q, q_0, \delta, \Omega)$ be a safety automaton for the safety LTL formula φ and $q_s \in Q, \alpha_s \in 2^{O_\mathsf{c}}$ be an output of the controller. There exists a linear tree automaton $\mathcal{A}'$ s.t. $\mathcal{L}(\mathcal{A}') = \mathbb{S}_\mathcal{A}^{q_s, \alpha_s}$.*

Proof. We construct a tree automaton $\mathcal{A}'$ over (2^{O_p})-labelled 2^{I_p} trees that accepts all plants for which a controller implementation exists with the current choice α_s that ensures the specification. The idea is to simulate the plant and controller behavior on the word automaton $\mathcal{A}$, where the automaton is implicitly labelled with both plant and controller outputs, and only branches over environment outputs. To make the tree automaton labelled with plant outputs, we remove each controller output β_c from the labels and make the corresponding node branch over all possible plant inputs, i.e., outputs of both the controller and the environment, by taking all branches with inconsistent controller outputs (i.e., $\neq \beta_\mathsf{c}$) to an accepting sink state. In addition, we simulate $\mathcal{A}$ from q_s and restrict the initial choice of the controller to α_s by introducing a new initial state q_0'. Formally, the automaton $\mathcal{A}' = (Q', q_0', \Delta', \Omega')$ is defined as follows:

- $Q' = Q \uplus \{q_0', q_{sink}\}$,
- $\Delta' \subseteq Q \times 2^{O_\mathsf{p}} \times (2^{I_\mathsf{p}} \to Q)$ is the minimal set such that
  - for all $\beta_\mathsf{p} \in 2^{O_\mathsf{p}}$, there is a transition $(q_0', \beta_\mathsf{p}, g)$ such that for all $\beta \in 2^{I_\mathsf{p}}$, $g(\beta) = \begin{cases} \delta(q_s, \beta_\mathsf{p} \cup \beta) & \text{if } \beta \cap O_\mathsf{c} = \alpha_s, \\ q_{sink} & \text{otherwise,} \end{cases}$
  - for all $q \in Q$, $\beta_\mathsf{p} \in 2^{O_\mathsf{p}}$, and $\beta_\mathsf{c} \in 2^{O_\mathsf{c}}$, there is a transition (q, β_p, g) s. t. for all $\beta \in 2^{I_\mathsf{p}}$, $g(\beta) = \begin{cases} \delta(q, \beta_\mathsf{p} \cup \beta) & \text{if } \beta \cap O_\mathsf{c} = \beta_\mathsf{c}, \\ q_{sink} & \text{otherwise,} \end{cases}$
  - for all $\beta_\mathsf{p} \in 2^{O_\mathsf{p}}$, there is a transition $(q_{sink}, \beta_\mathsf{p}, g)$ s.t. $f(\beta) = q_{sink}$ for all $\beta \in 2^{I_\mathsf{p}}$.
- $\Omega' = safe(F')$ with $F' = F \cup \{q_0', q_{sink}\}$.

Compared to $\mathcal{A}$, the automaton $\mathcal{A}'$ has two new states q_0' and q_{sink}. For all transitions from q_0', we fix the controller output to be α_s and quantify over all possible plant outputs. Afterwards, we branch over all possible plant inputs but only simulate the automaton $\mathcal{A}$ when the controller output is α_s, otherwise, we go to the sink state q_{sink}. Similarly, in the general case, we quantify over all possible controller outputs β_c and plant outputs β_p, but only use the plant output to explicitly label the tree. Then, we branch over all possible plant inputs but only simulate the automaton $\mathcal{A}$ when the controller output is β_c, otherwise, we go to the sink state q_{sink}. The runs of this automaton are all trees that branch over environment and controller outputs and are labeled with plant outputs that satisfy the prophecy $\mathbb{S}_{\mathcal{A}}^{q_s,\alpha_s}$. Since the transition function follows a similar construction

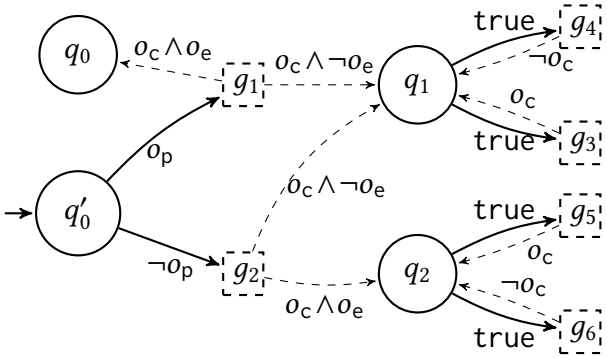


Figure 4.3 Parts of the tree automaton (without transitions of q_0 and q_{sink}) with safe states $F' = \{q'_0, q_0, q_1\}$ for prophecy $\mathbb{S}^{q_0,o_c}_{\mathcal{A}_\varphi} = \langle o_p \rangle$ of Figure 4.2. Each function g_i is represented by dashed edges.

for reactive synthesis (without a plant) in [91], the correctness of the construction follows immediately. As $|Q'| = |Q| + 2$, the size of $\mathcal{A}'$ is linear in the size of $\mathcal{A}$. □

Example 32. Consider the universal controller depicted in Figure 4.2 for the running example. A tree automaton $\mathcal{A}'$ representing the prophecy $\kappa(q_0, o_c) = \mathbb{S}^{q_0,o_c}_{\mathcal{A}_\varphi}$ is shown in Figure 4.3. Since the prophecy originates from the initial state q_0, a new initial state q'_0 is introduced as a copy of q_0. From q'_0, the controller output is fixed to o_c for simulation, and the automaton branches over all possible plant outputs. For instance, given the plant output o_p from q'_0, the automaton $\mathcal{A}_\varphi$ is simulated using the function g_1 when the controller output is o_c (otherwise, the automaton transitions to the sink state q_{sink}). This results in branching to the states q_0 and q_1 depending on the environment outputs. Similarly, for the plant output $\neg o_p$ from q'_0, the automaton $\mathcal{A}_\varphi$ is simulated using the function g_2 when the controller output is o_c. From all other states, the automaton branches over all possible plant and controller outputs. For example, from state q_1, the controller output is fixed to o_c (or $\neg o_c$), and the automaton $\mathcal{A}_\varphi$ is simulated using the function g_3 (or g_4) with consistent controller output. Since every transition in $\mathcal{A}_\varphi$ from q_1 leads back to q_1, the function g_3 maps every input with a consistent controller output to q_1 (and an inconsistent controller output to q_{sink}).

→ p. 95

It is not possible to represent safe prophecies with a word automaton, as we are looking for an automaton that accepts all plant strategies of any size, which are infinite trees. Intuitively, the word automaton that verifies if all traces of a plant are contained in an automaton, i.e., the model checking problem, checks whether there exists a controller *trace* rather than a controller *implementation*. This over-approximates the set of plants and hence, the prophecy set. We formalize this observation in the following theorem.

Theorem 17. *There exists a safety automaton $\mathcal{A}$ such that for some state q of $\mathcal{A}$ and controller output α the prophecy set $\mathbb{S}_{\mathcal{A}}^{q,\alpha}$ cannot be represented by a word automaton.*

Proof. Assume that there is a *word* automaton $\mathcal{A}'$ that accepts the prophecy set $\mathbb{S}_{\mathcal{A}}^{q,\alpha}$. This automaton must guess a controller trace for each plant trace and verify that they satisfy the specification. Assume the specification $\varphi = \square(o_c \leftrightarrow \bigcirc o_p)$ and the set of variables $\mathcal{V} = \{o_e, o_c, o_p\}$. This specification enforces that the plant's output only depends on the inputs up to the previous step, since the controller can only output based on the inputs up to that point. The automaton for the prophecy of any state q must contain accepting runs for both of the following (continuation) traces, which are part of *different* controllers as they react differently on the same inputs:

$$t_c = \{o_e, o_c\}, \{o_e, o_c\} \ldots$$
$$t'_c = \{o_e, o_c\}, \{o_e, \neg o_c\} \ldots$$

For a plant that sets o_p in the third step independent of o_e in the third step, t_c is the correct controller trace, and t'_c is the correct trace for the plant that sets $\neg o_p$ in that scenario. Now, assume the plant that contains the following traces that cannot be in any prophecy, since no correct controller can act correctly:

$$t_p = \{o_e, o_p\}, \{o_e, o_p\}, \{\neg o_e, o_p\} \ldots$$
$$t'_p = \{o_e, o_p\}, \{o_e, o_p\}, \{o_e, \neg o_p\} \ldots,$$

The word automaton, however, would still wrongly accept this plant as part of the prophecy because t_p will be matched with t_c, and t'_p will be matched with t'_c even though they originate from different controllers. Therefore, word automata cannot precisely capture prophecies as they cannot enforce the structure of the implementations. □

4.2 Controller Synthesis with Universal Controllers

In this chapter, we introduce an automata-theoretic method for synthesizing universal controllers that adhere to safety LTL specifications. Additionally, we describe an algorithm that, given a plant and a universal controller, generates a controller compatible with the plant while meeting the specified requirements. We continue to call the controllers with prophecies universal controllers since the results of our algorithms and procedures satisfy the requirements for universal controllers, i.e., correctness and most permissiveness.

Algorithm 3: Universal Controller Synthesis

```
1 let universalController(φ) :=
2   let A_φ = (Q, q_0, δ, Ω) ← LTLtoDSA(φ) // Construct determistic automaton
3   let κ ← [] // Initialize prophecy annotation
4   for q ∈ Q and α ∈ 2^{O_c}
5     let κ(q, α) ← prophecy(q, α, A_φ) // Comute prophecy for every state – action pair
6   return U = (Q, q_0, δ, κ)
```

4.2.1 Computing Universal Controllers

We begin with an algorithm that computes the universal controller, relying on the tree automaton construction that we presented in the previous chapter. Algorithm 3 outlines this algorithm that takes the safety LTL specification φ as input and constructs the universal controller. The process begins by generating a safety automaton $\mathcal{A}_\varphi$ whose language is equivalent to the given specification φ. This is a standard translation from LTL to deterministic safety automata (DSA). The states of this automaton form the state space of the universal controller. Next, the algorithm iterates over all combinations of states and controller outputs, constructing the corresponding prophecy using the method described in Section 4.1.4. Once the map κ is populated with all state-output pairs, the construction of the universal controller U is finished. The universal controller operates as follows: For a given environment output and state s, it selects an output α such that the plant behaves according to the tree automaton $\kappa(s, \alpha)$. Since the alphabet of $\mathcal{A}_\varphi$ encompasses all propositions, the transition function δ is defined over the plant outputs. During composition, the prophecy is verified to align with the plant's current output, ensuring consistency.

→ p. 100

Instead of constructing a separate tree automaton for each state-output pair in $\mathcal{A}_\varphi$, we construct a global tree automaton that represents all possible plants for the specification, modifying only the initial states and transitions as needed. This global tree automaton is similar to prophecy$(q, \alpha, \mathcal{A})$, but here q is the initial state q_0 of $\mathcal{A}_\varphi$, and α is not explicitly considered during the construction. The resulting tree automaton $\Omega_s = (Q, q_0, \Delta, \Sigma)$ accepts all plant-controller pairs that satisfy the specification. Within the loop of Algorithm 3, the construction is simplified to updating the initial state in $\mathcal{A}_s$ to a duplicate of q and adding a transition from q with α to the transition relation δ. For the implementation of this algorithm, as explained in Section 4.2.3, it is sufficient to store the updated initial state and the controller outputs. The modified version of the global tree automaton is computed only during the verification step in Algorithm 4, i.e., once this specific state and output pair are explored.

4.2.2 Computing Controllers for Plants

After computing the universal controller, it can be composed with a plant to derive a plant-specific controller that aligns with the plant's behavior. In general, this can be done by building the cross-product of the universal controller and plant and evaluating the plant against the tree automata on each output of the universal controller. To avoid the full cross-product, we elaborate on an on-the-fly exploration of the product state-space of the universal controller and the plant, as described in Algorithm 4. The initial state of the composed controller is the tuple (q_0, q_0'), combining the initial states of the universal controller and the plant. For clarity, we omit double parentheses in function calls involving tuples of universal controller and plant states when the context is clear. The algorithm employs a **while** loop to explore states in the composition until no further states remain. Within the loop, spanning lines 5 to 12, a one-step composition is computed for the current state (q_c, q_p) using stepComposition$(q_c, q_p, U, \mathcal{T}_\mathsf{p})$. This function takes the current states, the universal controller, and the plant as input, and returns a list of edges, each consisting of an alphabet assignment and a future state reachable in the composition. Not all computed edges are valid for the given plant, so each edge's prophecy must be verified before being added to the explicit controller. To perform this verification, the tree automaton $\mathcal{A}_s$ representing the prophecy is obtained using $\kappa(q_c, \sigma)$, and the plant is modified to $\mathcal{T}_\mathsf{p}'$, starting in q_p with σ. The membership check, performed in line 9 via isMember$(\mathcal{T}_\mathsf{p}', \mathcal{A}_s)$, confirms whether the tree generated by $\mathcal{T}_\mathsf{p}'$ is an accepting run of $\mathcal{A}_s$. If the check succeeds, the edge is added to the controller's edges, and its target state is queued for exploration if it hasn't already been visited. The algorithm ultimately returns all explored states and edges, with the starting state being (q_0, q_0').

Tree Automata Checking. The most computationally intensive step is the verification step, where the modified plant must be checked against the prophecy tree automaton. In Algorithm 4, this verification is carried out by the function isMember$(\mathcal{T}_\mathsf{p}', \mathcal{A}_s)$. As described in the literature [144], this procedure is encoded as a 2-player game where the environment player selects the plant's inputs, and the system player must respond with valid outputs for each input. If the system player has a winning strategy, the infinite tree generated by the plant's Moore machine is guaranteed to be contained in the language of the tree automaton. The game's winning condition is a *safety* property, derived from the acceptance condition of the tree automaton, and the state space of the game is proportional to the combined size of the tree automaton and the plant.

Algorithm 4: On-the-fly Composition

```
1  let composition(U = (Q, q_0, δ, κ), T_p = (Q', q'_0, δ', o)) :=
2    let s ← (q_0, q'_0), states ← Set(), edges ← Set()
3    let queue ← Queue(s)
4    while (q_c, q_p) ← queue.pop() do // Pick state pair in controller and plant
5      let edgelist ← stepComposition(q_c, q_p, U, T_p) // One-step unrolling
6      for (σ, (q'_c, q'_p)) in edgelist do
7        A_s ← κ(q_c, σ) // Obtain the prophecy
8        T'_p ← modifiedPlant(T_p, q_p, σ) // Initialize plant for prophecy
9        if isMember(T'_p, A_s) then
10         edges.add((q_c, q_p), σ, (q'_c, q'_p)) // Add action to explicit controller
11         queue.push(q'_c, q'_p) if not states.contains(q'_c, q'_p)
12     states.add(q'_c, q'_p)
13   return (states, (q_0, q'_0), edges)
```

Hashing Intermediate Results. The number of automaton checks in Algorithm 4 is given by $|\Sigma| \times |U| \times |\mathcal{T}_\mathsf{p}|$, which corresponds to the product of the alphabet size, the size of the universal controller, and the size of the plant. To significantly reduce the number of automaton checks, we employ a hashing mechanism for intermediate results: When solving a game, we identify sets of *winning* and *losing* states. These states are then mapped back to the composition, allowing us to maintain a function $\mathrm{h}(\sigma, (q_c, q_p))$ that records whether a given edge-state pair has already been computed as winning or losing. Before invoking isMember, we first check if the current edge has been hashed and, if so, return the cached result. As shown in our experiments in Section 4.2.3, the → p. 107 number of *actual* containment checks performed without hashing is reduced to *one* for nearly all benchmarks. This occurs when every state is reachable for every initial transition from (q_0, q'_0), as the game is computed at least for this state first.

Early Termination. The local exploration of the state space enables the application of heuristics for early termination during both the composition of the universal controller with the plant and the construction of the game. One such heuristic is based on the transition function of the currently discovered edges: If the explored set of edges includes a transition for every possible environment output, we can stop exploring the current edges and proceed to the next state in the queue. This check is performed before returning to line 6 from line 11. Additionally, the edgelist is ordered to prioritize already explored states, ensuring that such edges are processed before verifying edges leading to new states.

The semantics of the specification are leveraged to reduce the size of the game solved during verification. Since the acceptance condition depends solely on the specification automaton, exploration of the plant's state space can terminate early when a clearly accepting or rejecting loop, such as a terminal state, is detected. These early termination heuristics are particularly effective due to a key observation: The specific implementation details of the plant can often be disregarded when solving the controller synthesis problem.

To comply with existing tools for word automata, the controller is constructed as a Mealy machine, where edge labels are used instead of state labels, as is standard for automata. This representation can easily be converted into a Moore machine by ignoring the first environment output and associating the controller output with the source state of each edge.

Correctness and Complexity. Algorithm 3 and Algorithm 4 collectively synthesize a controller for a given specification and plant using the automata constructions detailed in Section 4.1.4. The correctness of the approach directly follows from corollary 3. Since the tree automata for the prophecies are linear in size relative to the deterministic specification automaton, solving the game in polynomial time yields the same complexity as the traditional controller synthesis problem [91]. Furthermore, as the number of prophecies is proportional to the size of the specification automaton, the overall time complexity of universal controller synthesis remains doubly exponential in the size of the specification. To summarize, we derive the following result:

Corollary 4. *Let φ be a safety LTL formula. Algorithm 3 returns a universal controller U that is correct and most permissive for φ. Furthermore, given an admissible plant $\mathcal{T}_\mathsf{p}$, Algorithm 4 returns a controller $\mathcal{T}_\mathsf{c}$ such that $\mathcal{T}_\mathsf{c} \parallel \mathcal{T}_\mathsf{p} \vDash \varphi$. The overall time complexity of computing $\mathcal{T}_\mathsf{c}$ using Algorithm 3 and Algorithm 4 is doubly-exponential in the size of the specification φ and linear in the size of the plant $\mathcal{T}_\mathsf{p}$.*

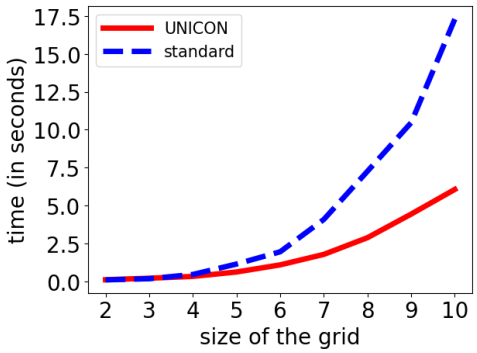
4.2.3 Experiments

We have developed a prototype tool, UNICON [107], to implement our approach. For benchmarking, UNICON also includes the standard controller synthesis algorithm, which reduces the synthesis problem to game solving [91]. The tool is written in F# and leverages FSOMEGALIB [114] and SPOT [76] for automata operations, as well as OINK [72] for game solving. All experiments were conducted on a machine equipped with an Apple M1 Pro 8-core CPU and 16GB of RAM.

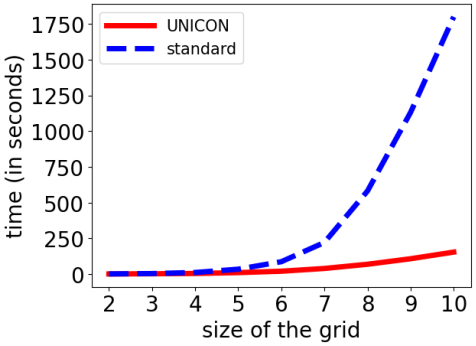
We assess our approach using benchmarks where the controller directs a robot through a maze-like environment, avoiding collisions with walls and a second (uncontrollable) robot. The workspace is modeled as a grid of cells, with walls between some cells and along the grid's boundary. The grid size is parameterized. Initially, the controller robot starts at the grid's center, while the plant robot begins in the bottom-left corner. At each step, both robots can either stay in place or move in one of four directions: up, down, left, or right. The controller robot is equipped with sensors to detect adjacent obstacles (walls or the plant robot) or collisions. The controller's behavioral objective is specified using LTL formulas. We evaluate three types of specifications: Specification type 1 is an invariant $\square \neg col$, requiring the controller to always avoid collisions. Specification type 2 extends this with the formula $\neg col \, \mathcal{W} \, \neg \mathtt{assump}_\mathsf{p}$, where $\mathtt{assump}_\mathsf{p}$ assumes that a collision can only occur if, in the previous step, the controller moved toward an obstacle or stayed in place with an obstacle nearby. In the latter case, the uncontrolled robot could crash into the controller robot in the next time step. Specification type 3 imposes a restriction on the first three steps of execution, requiring that the robot does not detect an obstacle in the upward direction during these steps: $\bigcirc \neg u_{obs} \wedge \bigcirc\bigcirc \neg u_{obs} \wedge \bigcirc\bigcirc\bigcirc \neg u_{obs}$.

Experiments on Robot Benchmarks The experimental results are summarized in Figure 4.4, which compares the execution times of UNICON (including the time for synthesizing the universal controller and deriving a correct controller using Algorithms 3 and 4) with the standard controller synthesis approach. For all three specification types, our approach demonstrates better scalability with increasing grid size, whereas the standard approach's runtime grows rapidly and becomes impractical. This difference arises because the standard synthesis method constructs the full composition of the plant and the specification automaton to solve the problem, while the universal controller synthesis performs the composition of the plant and the prophecies on the fly. Since the size of the prophecies is independent of the plant size, the universal controller synthesis handles larger grids more efficiently. Specification type 2 outperforms type 1 due to the additional assumption, which simplifies the prophecies. The simplest specifications, like type 3, are the most advantageous, as the prophecies only need to account for three future steps.

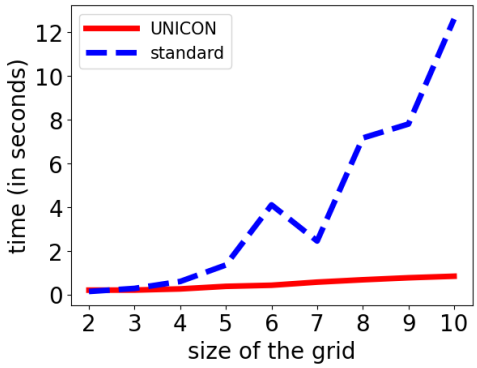
Experiments on SYNTCOMP benchmarks. The benchmarks of the annual SYNTCOMP reactive synthesis competition [141] are often used to evaluate synthesis algorithms. Since the benchmarks do not contain plant models, we cannot use them to



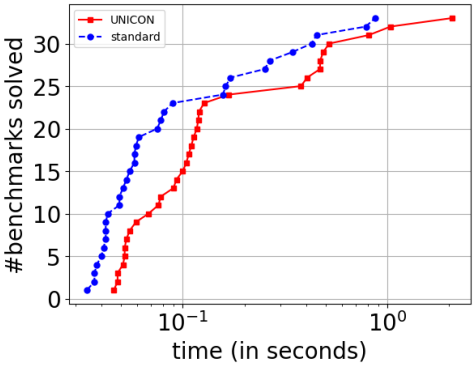
(a) Specification Type 1



(b) Specification Type 2



(c) Specification Type 3



(d) SYNTCOMP benchmarks

Figure 4.4 The execution times of UNICON and the standard synthesis approach. Figure 4.4a to Figure 4.4c illustrate the results for three specification types across increasing robot benchmarks. Figure 4.4d presents a cactus plot showing the number of SYNTCOMP benchmarks solved within a given time budget.

evaluate the scalability of our approach, but we still include them here as a baseline to evaluate the overhead caused by the computation of the universal controller. We selected the 68 benchmarks that have safety requirements and explicit environment assumptions, and translated the assumption formulas into plant models. As shown in Figure 4.4d, UNICON shows similar performance to standard synthesis. Because of the small size of the plant models resulting from the assumption formulas, the overhead for the universal controller is not compensated for by the improved performance in the size of the plant model. Still, computation times remain close, showing that UNICON is able to scale to practical problem instances.

Chapter 5

Monitoring Second-Order Hyperproperties

In this chapter, we move beyond standard epistemic reasoning and introduce a new logic for hyperproperties that is capable of expressing more complex properties such as common knowledge. The new logic is accompanied by a monitoring algorithm that verifies whether a distributed system satisfies a given second-order hyperproperty during execution time. Temporal hyperproperties, originally formalized by Clarkson and Schneider [60], cover a wide range of system requirements, from information-flow policies to properties about agents' knowledge in distributed systems. The dominant temporal logic for hyperproperties is HyperLTL [59], which extends LTL with trace quantifiers. Despite its success in many applications [105, 182, 74], HyperLTL is inherently limited: It only permits first-order quantification over individual traces, not over sets of traces. Once HyperLTL is combined with propositional quantification, the logic captures individual knowledge (where a single set of indistinguishable traces is relevant) but fails for notions like common knowledge that intrinsically quantify over sets of traces and require reasoning about arbitrarily deep nestings of knowledge.[1]

To overcome this, we introduce Hyper^{2}LTL in Section 5.1, a temporal logic for hyperproperties that permits quantification over sets of traces. Using a simplified syntax, the fact that there exists a trace π on which a temporal property expressed in LTL φ is common knowledge can be defined as:

$$\exists \pi.\, \exists X.\ \pi \in X \wedge \left(\forall \pi' \in X.\, \forall \pi''.\, \left(\bigvee_{i=1}^{n} \pi' \sim_i \pi'' \right) \rightarrow \pi'' \in X \right) \wedge \forall \pi' \in X.\, \varphi(\pi').$$

[1] We refer to [46] for a detailed expressiveness analysis between temporal logics with knowledge and temporal hyperlogics.

Here the second-order quantifier $\exists X$ defines a set of traces X such that (i) X contains π; (ii) X is closed under each agent's indistinguishability relation—if π' is in X then any π'' indistinguishable from π' for some agent i is also in X; and (iii) every trace in X satisfies φ. The existence of such an X exactly captures that φ is common knowledge on trace π. Beyond common knowledge, Hyper^{2}LTL enables specification of hyperproperties outside HyperLTL's expressiveness, including constructs arising in Mazurkiewicz trace theory and asynchronous hyperproperties.

We also develop monitoring techniques for second-order hyperproperties over finite traces in Section 5.2, for a finite-trace semantics of Hyper^{2}LTL, called Hyper^{2}LTL$_\text{f}$. Runtime monitoring offers a scalable, model-free alternative to verification: It inspects actual executions rather than a system model. Monitoring hyperproperties can follow two execution paradigms: The parallel model (see Section 5.2.2), where a fixed collection of traces is observed concurrently, and the sequential model (see Section 5.2.3), where traces arrive one by one [95]. For the parallel setting, we show that monitoring Hyper^{2}LTL$_\text{f}$ specifications can be reduced to monitoring first-order hyperproperties. In the sequential setting, we show that the monitoring problem is undecidable. However, we identify a useful decidable fragment: Monotone second-order hyperproperties (see Section 5.2.4). A property is $\oplus$-monotone if satisfaction on a set of traces persists under supersets of the current trace set, and $\ominus$-monotone if a violation persists under supersets of the current trace set. Monotonicity enables the monitor to reach definitive verdicts that remain valid as more traces are observed. We present a syntactic inference system to detect monotonicity and give an incremental monitoring algorithm (see Section 5.2.5) that examines incoming traces until a conclusive result is obtained. This approach is implemented in the tool `MoSo`, which we evaluate on benchmarks including common-knowledge scenarios and planning problems in Section 5.2.6, yielding encouraging performance.

Outline. Section 5.1 introduces Hyper^{2}LTL illustrated by the muddy children puzzle. We present the infinite trace semantics of Hyper^{2}LTL and its restriction to quantification over minimal and maximal sets of traces Hyper^{2}LTL$_\text{m}$ in Section 5.1.2. The finite trace semantics of HyperLTL with second-order quantification, called Hyper^{2}LTL$_\text{f}$, is presented in Section 5.1.4. In Section 5.2, we show a monitoring algorithm for Hyper^{2}LTL$_\text{f}$ formulas that is based on the monotonicity (see Section 5.2.4) of the formula.

Publications. The logics Hyper^{2}LTL and Hyper^{2}LTL$_\text{m}$ were presented in [35]. While the logic Hyper^{2}LTL is a shared contribution with Raven Beutner, the fixpoint version of

Hyper^{2}LTL and model-checking results, which are part of [35], are no contributions of this thesis. Monitoring second-order hyperproperties was presented in [34], including the finite-trace semantics for Hyper^{2}LTL in Hyper^{2}LTL$_\text{f}$. While fixpoints are a useful part of the semantics of Hyper^{2}LTL$_\text{f}$, they were introduced by Raven Beutner for Hyper^{2}LTL.

[35] Raven Beutner, Bernd Finkbeiner, Hadar Frenkel, Niklas Metzger. **Second-Order Hyperproperties**. *35th International Conference on Computer-Aided Verification (CAV 2023)*

[34] Raven Beutner, Bernd Finkbeiner, Hadar Frenkel, Niklas Metzger. **Monitoring Second-Order Hyperproperties**. *23rd International Conference on Autonomous Agents and Multiagent Systems, (AAMAS 2024)*

5.1 HyperLTL with Second-Order Quantification

In the remainder of this thesis, we extend temporal reasoning in hyperlogics to encompass properties that require common knowledge-style reasoning. We refer to these as *second-order hyperproperties*, since they involve quantification over sets of traces, in contrast to (first-order) hyperproperties that quantify over individual traces. To illustrate the reasoning techniques behind second-order hyperproperties, we use the notion of common knowledge [81] and apply it to a variation of the classic muddy children puzzle [81]. We then introduce a new logic capable of specifying second-order hyperproperties. This logic is analyzed with respect to model checking complexity and expressiveness, particularly in comparison to linear temporal logic with knowledge operators—an extension of LTL that can express knowledge and common knowledge.[2] In standard HyperLTL, trace quantification is first-order and ranges over all system traces, which limits its ability to express properties. One such property is common knowledge, which necessitates the definition of a set of traces, on which a certain (first-order) hyperproperty has to hold. To address this, we extend HyperLTL with second-order quantification by introducing variables that range over sets of traces and allowing quantification over traces within these sets. We propose two variants of this logic: One for infinite traces and one for finite traces. Both Hyper^{2}LTL and its finite-trace counterpart Hyper^{2}LTL$_\text{f}$ are highly expressive and intuitive logics.

[2] While LTL with knowledge is not designed as a hyperlogic, it can express one specific class of hyperproperties, namely knowledge.

5.1.1 The Muddy Children Puzzle

Consider a distributed system with three agents, Alice, Bob, and Suzy, in a variant of the classic Muddy Children problem [81]. In this scenario, Suzy wears a blue helmet, while Alice and Bob each wear black helmets. The agents stand in a circle so that each can see the others' helmets. The protocol requires that, in turn, each agent raises their hand precisely when they know the color of their own helmet. It is common knowledge that all helmets are either blue or black, and that there is at least one blue helmet. In this example, there is exactly one blue helmet.

We now analyze the knowledge of the different agents, starting with Bob (B): Bob observes that Suzy's helmet is blue and Alice's is black. Can he infer the color of his own helmet? To answer this, Bob must consider all situations that are indistinguishable to him based on his observations. There are two such possibilities: One in which only Suzy wears a blue helmet and another in which both Suzy and Bob wear blue helmets. Since the statement "Bob wears a blue helmet" is true in one scenario and false in the other, Bob cannot yet determine the color of his own helmet. Using the knowledge operator for Bob and the atomic proposition $blue_B$ (true if and only if Bob's helmet is blue), we write

$$K_B(blue_B)$$

to express that Bob knows that his helmet is blue. It is not true that Bob knows that his helmet is blue. A similar argument applies to Alice. Suzy's situation, however, is different. From her perspective, neither of the other agents is wearing a blue helmet. Given that it is common knowledge that there is at least one blue helmet, Suzy can deduce that her own helmet must be blue. Since all scenarios that are indistinguishable to her confirm this, she knows the color of her helmet and immediately raises her hand to indicate her knowledge. This finished the first round of the muddy children puzzle.

The reasoning proceeds to the second round. Alice and Bob each observe only one blue helmet and notice that Suzy raised her hand in the first round, indicating she did not see another blue helmet. If she had, her action would not have been justified. Consequently, after the second round, all agents are able to deduce the color of their helmets, as there are no longer any indistinguishable scenarios.

Knowledge in Distributed Systems. In a multi-agent system, an agent is said to know a fact if that fact holds on every execution of the system that the agent cannot distinguish from the current one — this inherently relates multiple traces of the system. HyperQPTL [188] is a logic specifically designed to express such hyperproperties, in-

cluding knowledge. Using HyperLTL, we can formalize the requirement that whenever an agent raises its hand, it must know the color of its helmet:

$$\forall\pi.\forall\pi'.\,(\pi \sim_S \pi' \,\mathcal{U}\, raiseHand_S^{\pi}) \rightarrow (blue_S^{\pi} \leftrightarrow blue_S^{\pi'})$$

The relation $\sim_S$ specifies that all atomic propositions that Suzy can observe, including the color of Alice and Bob's helmets and whether they raised their hands, must be equivalent on the traces π and π'. This specifies the *indistinguishability* property. Note that this is a similar property to *distinguishability*, as discussed in Section 3.1.2 and → p. 38 Section 3.4, where it is necessary that some traces *can* be distinguished from others. → p. 70 The formula $(blue_S^{\pi} \leftrightarrow blue_S^{\pi'})$ specifies that both traces must agree on the color of Suzie's helmet. The universal quantification over pairs of traces at the beginning of the formula arbitrarily chooses pairs of traces in the system. If those traces happen to be indistinguishable for Suzy, then she knows that her helmet is blue if it is true for every such trace. This is specified by the formula's implication and the fact that it has to hold for every trace that is indistinguishable. We also call this set the set of indistinguishable traces. Note that this set changes for different numbers of blue helmets in the system: The formula specifies that Suzy knows her helmet's color when raising her hand, which can take $n + 1$ time steps, where n is the number of blue helmets.

Common Knowledge as a Hyperproperty. Quantification over pairs of traces suffices to express knowledge and, more generally, relational properties between traces. However, can we extend this by formalizing that everyone knows that everyone knows, and so on, the color of their helmet? This property is called common knowledge. For explanation, we first consider the property "everyone knows," denoted as $E\varphi$, which is an operator in the temporal logic of knowledge (see Chapter 2). This property requires → p. 24 verifying the knowledge of each agent individually, without relating the knowledge states of different agents. For the muddy children scenario, it is evident that after two steps, each agent knows the color of their own helmet. The corresponding formula is similar to the one for individual knowledge, but includes a conjunction over all agents. In general, for n blue helmets, every agent knows their helmet color after $n+1$ steps [81].

Common knowledge, on the other hand, involves agents having knowledge about the knowledge of others, recursively. The common knowledge operator C in $\mathrm{LTL}_{\mathrm{K,C}}$ is defined as

$$C\varphi \text{ iff } E^{\omega}\varphi$$

meaning that there is an infinite chain of E operators: Everyone knows that everyone knows that everyone knows, and so on, that φ holds. Thus, not only does everyone know the fact, but everyone also knows that everyone knows it, and so forth. In our example, we can state that after $n + 1$ steps, each helmet color is common knowledge. The process of establishing common knowledge is inherently iterative: Each agent begins with their own limited perspective, and we systematically consider all worlds that are indistinguishable to every agent based on their observations. For instance, starting from Suzy's viewpoint, she is aware of the helmet colors of the other agents but not her own. If we then consider Alice's perspective from Suzy's view, we must also include the scenario where Alice could be wearing a blue helmet, as this is a plausible situation for Alice. Through this iterative process, we accumulate all possible worlds by considering every agent's local view and the corresponding indistinguishable scenarios. In the muddy children example, common knowledge emerges only after the second round of communication, at which point it coincides with the state where everyone knows that everyone knows.

Common knowledge is not expressible in HyperLTL. In fact, it is not expressible in any decidable hyperproperty language with first-order quantification over traces [176]. Instead, second-order quantification, i.e., quantification over sets of traces, is necessary. In the following section, we introduce Hyper^{2}LTL, which is a logic of second-order hyperproperties. We give a first glimpse of the logic by observing its use for defining common knowledge:

$$\exists \pi.\, \exists X.\ \pi \in X \wedge \left(\forall \pi' \in X.\, \forall \pi''.\, \Big(\bigvee_{i=1}^{n} \pi' \sim_i \pi'' \Big) \rightarrow \pi'' \in X \right) \wedge \forall \pi' \in X.\, \varphi(\pi').$$

The second-order quantifier $\exists X$ asserts the existence of a set X of traces such that: (1) π is a member of X; (2) X is closed under each agent's observations, meaning that for every trace π' in X, any trace π'' that is indistinguishable from π' for some agent i must also be in X; and (3) every trace in X satisfies φ. The existence of such a set X precisely characterizes when φ is common knowledge on π. Again, in our example, φ are the colors of the helmets as trace properties. With the formula above, we can precisely express common knowledge and determine whether a system satisfies it. In the following, we define the logic Hyper^{2}LTL and its finite trace semantics, and show an algorithm for monitoring second-order hyperproperties.

5.1.2 Second-Order Quantification on Infinite Traces

We begin by defining an infinite trace logic that can be used to specify second-order hyperproperties over infinite traces. In addition to the set V of trace variables in HyperLTL, we introduce a set $\mathfrak{V}$ of second-order variables, typically denoted by uppercase letters $X, Y, \ldots$. We designate a special variable $\mathfrak{S} \in \mathfrak{V}$ to represent the set of traces generated by the system under consideration, and another variable $\mathfrak{A} \in \mathfrak{V}$ to denote the set of all possible traces.

Hyper^{2}LTL. The syntax of Hyper^{2}LTL is defined by the following grammar:

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

Here, $a \in \mathcal{V}$ denotes an atomic proposition, $\pi \in V$ is a trace variable, $X \in \mathfrak{V}$ is a second-order variable, and $\exists, \forall$ are existential and universal quantifiers, respectively. We also allow the standard derived Boolean constants (`true`, `false`), connectives ($\vee$, $\rightarrow$, $\leftrightarrow$), and temporal operators such as *eventually* ($\Diamond \psi := \texttt{true}\,\mathcal{U}\,\psi$) and *globally* ($\Box \psi := \neg \Diamond \neg \psi$). For a set of atomic propositions $V \subseteq \mathcal{V}$ and trace variables π, π', we use the abbreviation $\pi =_V \pi$ for the Boolean formula $\bigwedge_{a \in V} (a_\pi \leftrightarrow a_{\pi'})$, specifying the equivalence of π and π' on the propositions in V.

In addition to the trace assignment Π used in HyperLTL semantics, we introduce a second-order assignment $\Delta : \mathfrak{V} \rightharpoonup 2^{\Sigma^\omega}$, which maps each second-order variable to a set of traces. For $X \in \mathfrak{V}$ and $A \subseteq \Sigma^\omega$, we write $\Delta[X \mapsto A]$ for the assignment that updates X to A. Quantifier-free formulas ψ are interpreted as in HyperLTL with respect to a fixed trace assignment. The semantics for the quantifier prefix are defined as follows:

$$\begin{aligned}
&\Pi, \Delta \vDash \psi && \text{iff} \quad \Pi \vDash \psi \\
&\Pi, \Delta \vDash \exists \pi \in X.\, \varphi && \text{iff} \quad \exists t \in \Delta(X).\, \Pi[\pi \mapsto t], \Delta \vDash \varphi \\
&\Pi, \Delta \vDash \forall \pi \in X.\, \varphi && \text{iff} \quad \forall t \in \Delta(X).\, \Pi[\pi \mapsto t], \Delta \vDash \varphi \\
&\Pi, \Delta \vDash \exists X.\, \varphi && \text{iff} \quad \exists A \subseteq \Sigma^\omega.\, \Pi, \Delta[X \mapsto A] \vDash \varphi \\
&\Pi, \Delta \vDash \forall X.\, \varphi && \text{iff} \quad \forall A \subseteq \Sigma^\omega.\, \Pi, \Delta[X \mapsto A] \vDash \varphi
\end{aligned}$$

Second-order quantification modifies the assignment Δ by associating a variable with a set of traces, while first-order quantification updates the trace assignment Π by ranging over traces contained in the set specified by Δ.

To evaluate a formula, we start with an empty trace assignment and assign the special second-order variable $\mathfrak{S}$ to the set of traces generated by the system, and $\mathfrak{A}$ to the set of all possible traces. Formally, for a system $\mathcal{T}$ and a Hyper^{2}LTL formula φ, we write $\mathcal{T} \vDash \varphi$ if $\emptyset, [\mathfrak{S} \mapsto \mathit{Traces}(\mathcal{T}), \mathfrak{A} \mapsto \Sigma^\omega] \vDash \varphi$, where $\emptyset$ denotes the empty trace assignment. The model-checking problem for Hyper^{2}LTL asks whether this satisfaction relation holds.

We introduce some syntactic sugar for Hyper^{2}LTL formulas. While Hyper^{2}LTL allows quantification over traces within a second-order variable, it does not directly permit expressing, within the body of a formula, that a particular trace belongs to a second-order variable. To address this, we define the expression $\pi \triangleright X$ as syntactic sugar for $\exists \pi' \in X. \square(\pi' =_{\mathcal{V}} \pi)$; that is, π is considered a member of X if there exists a trace in X that matches π on all atomic propositions. It is important to note that $\pi \triangleright X$ may only be used outside the scope of temporal operators, ensuring that the resulting formula remains in the syntactic fragment supported by Hyper^{2}LTL.

Hyper^{2}LTL$_\mathrm{m}$. The semantics of Hyper^{2}LTL quantifies over arbitrary sets of traces, making even approximations to its semantics challenging. We propose Hyper^{2}LTL$_\mathrm{m}$ as a restriction that only quantifies over sets that are subject to an additional minimality or maximality constraint.[3] For large classes of formulas, it has been shown that this admits effective model-checking approximations. We define Hyper^{2}LTL$_\mathrm{m}$ with the following syntax grammar:

$$\varphi := \mathbb{Q}\, \pi \in X. \varphi \mid \mathbb{Q}\, (X, \lozenge, \varphi). \varphi \mid \psi$$
$$\psi := a_\pi \mid \neg\psi \mid \psi \wedge \psi \mid \bigcirc \psi \mid \psi \,\mathcal{U}\, \psi$$

where $a \in \mathcal{V}$, $\pi \in V$, $X \in \mathfrak{V}$, $\mathbb{Q} \in \{\forall, \exists\}$, and $\lozenge \in \{\wedge, \vee\}$ determines if we consider smallest ($\vee$) or largest ($\wedge$) sets. For example, the formula $\exists\, (X, \vee, \varphi_1). \varphi_2$ holds if there exists some set of traces X that satisfies both φ_2 and is the smallest set that satisfies φ_1. Such minimality and maximality constraints with respect to a (hyper)property arise naturally in many properties. Examples include common knowledge (cf. Section 5.1.1), → p. 113 asynchronous hyperproperties [23, 127, 48], and causality in reactive systems [62, 61].

For path formulas, the semantics of Hyper^{2}LTL$_\mathrm{m}$ is defined analogously to that of Hyper^{2}LTL and HyperLTL. We mainly adapt the semantics of Hyper^{2}LTL to trace set quantification for the expression $\mathbb{Q}(X, \lozenge, \varphi_1). \varphi_2$.

[3] In [35], this logic was called Hyper^{2}LTL$_\mathrm{fp}$. For clarification, we refer to the same logic as Hyper^{2}LTL$_\mathrm{m}$, identified by the minimal and maximal set conditions.

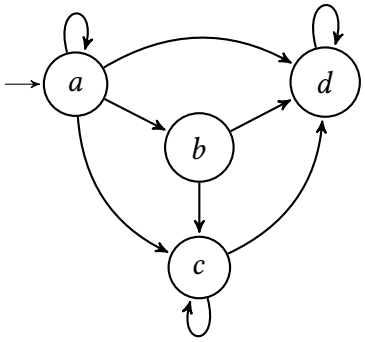


Figure 5.1 An example of a multi-agent system with two agents — agent 1 observes a and d, agent 2 observes c and d.

For the quantifier prefix, we define:

$$
\begin{array}{lll}
\Pi, \Delta \vDash \psi & \text{iff} & \Pi \vDash \psi \\
\Pi, \Delta \vDash \exists \pi \in X.\, \varphi & \text{iff} & \exists t \in \Delta(X).\, \Pi[\pi \mapsto t], \Delta \vDash \varphi \\
\Pi, \Delta \vDash \forall \pi \in X.\, \varphi & \text{iff} & \forall t \in \Delta(X).\, \Pi[\pi \mapsto t], \Delta \vDash \varphi \\
\Pi, \Delta \vDash \exists (X, \lozenge, \varphi_1).\, \varphi_2 & \text{iff} & \exists A \in \mathit{sol}(\Pi, \Delta, (X, \lozenge, \varphi_1)).\, \Pi, \Delta[X \mapsto A] \vDash \varphi_2 \\
\Pi, \Delta \vDash \forall (X, \lozenge, \varphi_1).\, \varphi_2 & \text{iff} & \forall A \in \mathit{sol}(\Pi, \Delta, (X, \lozenge, \varphi_1)).\, \Pi, \Delta[X \mapsto A] \vDash \varphi_2
\end{array}
$$

where $\mathit{sol}(\Pi, \Delta, (X, \lozenge, \varphi_1))$ denotes all solutions to the minimality/maximality condition given by φ_1, which we define by mutual recursion as follows:

$$
\begin{aligned}
\mathit{sol}(\Pi, \Delta, (X, \vee, \varphi)) &:= \{A \subseteq \Sigma^\omega \mid \Pi, \Delta[X \mapsto A] \vDash \varphi \wedge \forall A' \subsetneq A.\, \Pi, \Delta[X \mapsto A'] \nvDash \varphi\} \\
\mathit{sol}(\Pi, \Delta, (X, \wedge, \varphi)) &:= \{A \subseteq \Sigma^\omega \mid \Pi, \Delta[X \mapsto A] \vDash \varphi \wedge \forall A' \supsetneq A.\, \Pi, \Delta[X \mapsto A'] \nvDash \varphi\}
\end{aligned}
$$

A set A fulfills the minimality or maximality constraint if it satisfies φ and is minimal (for $\lozenge = \vee$) or maximal (for $\lozenge = \wedge$) among all sets satisfying φ. Note that $\mathit{sol}(\Pi, \Delta, (X, \lozenge, \varphi))$ may contain multiple sets or even be empty; that is, there may not always exist a unique minimal or maximal set satisfying φ. Note that, if the result is unique, we may omit the preceding quantifier. Accordingly, Hyper^{2}LTL$_\mathrm{m}$ quantifies over all sets that fulfill the minimality or maximality constraint.

Example 33. To illustrate the concept of common knowledge, we adapt an example from [174] and formalize it in Hyper^{2}LTL$_\mathrm{m}$. Figure 5.1 depicts a transition system with two agents: Agent 1, who observes a and d, and agent 2, who observes c and d. We are interested in the following property: *starting from the trace $t = a^n d^\omega$ for some fixed*

$n > 1$, is it common knowledge among the two agents that a holds at the second step? While $\bigcirc a$ clearly holds on t, common knowledge requires considering all traces that are observationally indistinguishable to either agent, potentially forming an infinite chain. For instance, agent 2 cannot distinguish between $a^n d^\omega$ and $a^{n-1} b d^\omega$, so agent 2 only knows that $\bigcirc a$ holds on t if it also holds on $t' = a^{n-1} b d^\omega$. To achieve common knowledge, agent 1 must also know that agent 2 knows $\bigcirc a$, which means that for all traces indistinguishable from t or t' for agent 2, $\bigcirc a$ must also hold. This process adds $t'' = a^{n-1} c d^\omega$ to the set of relevant traces, and the reasoning continues as follows:

$$
\begin{aligned}
t &= a^n d^\omega \\
K_2(t) &= a^{n-1} b d^\omega \\
K_1 K_2(t) &= a^{n-1} c d^\omega \\
K_2 K_1 K_2(t) &= a^{n-2} b c d^\omega \\
&\dots \\
K_1 K_2 \dots K_2(t) &= a c^{n-1} d^\omega
\end{aligned}
$$

Ultimately, the trace $a c^{n-1} d^\omega$ is included, demonstrating that $\bigcirc a$ is *not* common knowledge.

We specify the common knowledge property in Hyper^{2}LTL$_\mathrm{m}$. Let $obs(\pi_1, \pi_2) := \square(\pi_1 =_{\{a,d\}} \pi_2) \vee \square(\pi_1 =_{\{c,d\}} \pi_2)$ defining two traces π_1 and π_2 to be observationally equivalent for either agent 1 or agent 2.

$$
\begin{aligned}
&\forall \pi \in \mathfrak{S}.\Big(\bigwedge_{i=0}^{n-1} \bigcirc^i a_\pi \wedge \bigcirc^n \square d_\pi\Big) \rightarrow \\
&\qquad \Big(X, \curlyvee, \pi \triangleright X \wedge \big(\forall \pi_1 \in X. \forall \pi_2 \in \mathfrak{S}.\, obs(\pi_1, \pi_2) \rightarrow \pi_2 \triangleright X\big)\Big).\, \forall \pi' \in X.\, \bigcirc a_{\pi'}
\end{aligned}
$$

For a trace t of the form $a^n d^\omega$, we call the set X the *common knowledge set* for t, meaning that it contains all traces on which we have to verify the property $\bigcirc a$ in order to verify common knowledge. This set X is the smallest set that contains t, and is closed under observations by either agent, as specified by $obs(\pi_1, \pi_2)$. If $t \in X$ and π_2 is a trace in the system that is observationally equivalent to t_1 for some agent, then t_2 must also be part of the set X. Since we constrain this set to be minimal with the operator $\curlyvee$, this defines a unique set. As the last operation, we verify that all traces in X satisfy the LTL formula $\bigcirc a$. Our given system is a counterexample to this formula, since X must

contain $ac^{n-1}d^{\omega}$, and $\bigcirc a$ does not hold on this trace. This also means that no X exists that satisfies the defined property, and the formula evaluates to *false*.

5.1.3 Expressiveness

We continue by exploring the expressiveness of the proposed logics. Hyper^{2}LTL extends HyperLTL by introducing second-order quantification over arbitrary sets of traces. Because these sets can be chosen freely, Hyper^{2}LTL is expressive enough to capture the full power of HyperQPTL.

Lemma 6. *Hyper^{2}LTL (and Hyper^{2}LTL$_m$) subsumes HyperQPTL and, therefore, also subsumes HyperLTL.*

Proof. HyperQPTL extends HyperLTL by allowing quantification over atomic propositions, which is equivalent to quantification over arbitrary traces—not just those generated by the system. Any HyperLTL trace quantifier $\mathbb{Q}\pi.\varphi$ can be represented as $\mathbb{Q}\pi \in \mathfrak{S}.\varphi$ in Hyper^{2}LTL, while a HyperQPTL propositional quantifier $\mathbb{Q}\tau.\varphi$ corresponds to $\mathbb{Q}\pi_\tau \in \mathfrak{A}.\varphi$ where π_τ is a fresh trace variable. In the body φ of the formula, we replace every occurrence of the propositional variable τ by a_{π_τ} for some fixed proposition $a \in \mathcal{V}$. The propositional variable is, therefore, transformed into a trace variable, and the variable τ now refers to all possible traces, not only the system traces. Here, $\mathfrak{A}$ denotes the set of all possible traces, unconstrained by the system. Therefore, every HyperQPTL formula can be effectively translated into an equivalent Hyper^{2}LTL formula. The same proof shows that Hyper^{2}LTL$_\mathrm{m}$ subsumes HyperQPTL since Hyper^{2}LTL$_\mathrm{m}$ only restricts second-order quantification to minimal and maximal sets and allows the same first-order quantification as Hyper^{2}LTL. □

Hyper^{2}LTL is expressive enough to subsume a multitude of hyper logics, but a definite order to the hierarchy of hyper logics [63, 48] is still an open problem. We now relate Hyper^{2}LTL and Hyper^{2}LTL$_\mathrm{m}$.

Theorem 18. *Any Hyper^{2}LTL$_m$ formula φ can be effectively translated into a Hyper^{2}LTL formula φ' such that for all transition systems $\mathcal{T}$ we have $\mathcal{T} \vDash \varphi$ iff $\mathcal{T} \vDash \varphi'$.*

Proof. We translate the Hyper^{2}LTL$_\text{m}$ formula φ into a Hyper^{2}LTL formula $[\![\varphi]\!]$:

$$
\begin{aligned}
[\![\psi]\!] &:= \psi \\
[\![\exists \pi \in X.\, \varphi]\!] &:= \exists \pi \in X.\, [\![\varphi]\!] \\
[\![\forall \pi \in X.\, \varphi]\!] &:= \forall \pi \in X.\, [\![\varphi]\!] \\
[\![\exists (X, \vee, \varphi_1).\, \varphi_2]\!] &:= \exists X.\, [\![\varphi_1]\!] \wedge \left(\forall Y.\, Y \subsetneq X \Rightarrow \neg [\![\varphi_1[Y \leftarrowtail X]]\!]\right) \wedge [\![\varphi_2]\!] \\
[\![\forall (X, \vee, \varphi_1).\, \varphi_2]\!] &:= \forall X.\, \left([\![\varphi_1]\!] \wedge \left(\forall Y.\, Y \subsetneq X \Rightarrow \neg [\![\varphi_1[Y \leftarrowtail X]]\!]\right)\right) \Rightarrow [\![\varphi_2]\!] \\
[\![\exists (X, \wedge, \varphi_1).\, \varphi_2]\!] &:= \exists X.\, [\![\varphi_1]\!] \wedge \left(\forall Y.\, Y \supsetneq X \Rightarrow \neg [\![\varphi_1[Y \leftarrowtail X]]\!]\right) \wedge [\![\varphi_2]\!] \\
[\![\forall (X, \wedge, \varphi_1).\, \varphi_2]\!] &:= \forall X.\, \left([\![\varphi_1]\!] \wedge \left(\forall Y.\, Y \supsetneq X \Rightarrow \neg [\![\varphi_1[Y \leftarrowtail X]]\!]\right)\right) \Rightarrow [\![\varphi_2]\!]
\end{aligned}
$$

Path formulas and first-order quantification can be translated verbatim. To translate (fixpoint-based) second-order quantification, we use additional second-order quantification to express the fact that a set should be a least or greatest. We write $Y \subsetneq X$ as a shorthand for

$$\left(\forall \pi \in Y. \pi \triangleright X\right) \wedge \left(\exists \pi \in Y. \forall \pi' \in Y. \Diamond \neg (\pi =_\mathcal{V} \pi')\right)$$

and $\varphi_1[Y \leftarrowtail X]$ denotes the formula where all free occurrences of X are replaced by Y.

Note that above formula is no Hyper^{2}LTL formula as it is not in prenex normal form. However, no (first or second-order) quantification occurs under temporal operators, so we can easily bring it into prenex normal form. It is easy to see that any system satisfies φ in the Hyper^{2}LTL$_\text{m}$ semantics iff it satisfies $[\![\varphi]\!]$ in the Hyper^{2}LTL semantics. □

The inverse direction does not hold, and Hyper^{2}LTL is more expressive than Hyper^{2}LTL$_\text{m}$: One can easily specify second-order sets that are neither maximal nor minimal sets w.r.t. the semantics of Hyper^{2}LTL$_\text{m}$.

In the following, we point to a logic that can express knowledge and can naturally be encoded within our second-order hyperlogics Hyper^{2}LTL and Hyper^{2}LTL$_\text{m}$. The logic we compare against is LTL$_{\mathsf{K},\mathsf{C}}$, which extends LTL with the knowledge operator K and the common knowledge operator C. For some subset of agents A, the formula $\mathsf{K}_A\psi$ holds in time step i, if ψ holds on all traces equivalent to some agent in A up to time step i. LTL$_\mathsf{K}$ and HyperCTL* have incomparable expressiveness [46], but the knowledge operator K, without the common knowledge operator, can be encoded by either adding a linear past operator [46] or by adding propositional quantification (as in HyperQPTL) [188].

Using Hyper^{2}LTL$_\mathrm{m}$ we can encode LTL$_{\mathsf{K,C}}$, featuring the knowledge operator K *and* the common knowledge operator C. Recall that Cψ requires that ψ holds on the closure set of equivalent traces, up to the current time point [136]. Note that LTL$_{\mathsf{K,C}}$ cannot be encoded by only adding propositional quantification or the linear past operator.

Proposition 5. *For every* LTL$_{\mathsf{K,C}}$ *formula* φ *there exists an* Hyper^{2}LTL$_\mathrm{m}$ *formula* φ' *such that for any system* $\mathcal{T}$ *we have* $\mathcal{T} \vDash_{LTL_{\mathsf{K,C}}} \varphi$ *iff* $\mathcal{T} \vDash \varphi'$.

Proof. We follow ideas from [188] and extend the proof to common knowledge. We use an abbreviated Hyper^{2}LTL$_\mathrm{m}$ syntax throughout this proof with explicit propositional quantification as in HyperQPTL for readability. As shown by Lemma 6, this formula can easily be translated to plain Hyper^{2}LTL$_\mathrm{m}$. Let $\{A_1, \ldots, A_n\}$ be the set of agents for the jth knowledge operator $\mathbb{K} \in \{\mathsf{K}, \mathsf{C}\}$ in φ. Furthermore, assume that $1, \ldots, n$ maintains the syntactic ordering of knowledge operators such that for $i < j$, $\mathbb{K}_{A_j}$ is either a subformula of the knowledge operator with $\mathbb{K}_{A_i}$ or syntactically independent. Initially, we replace every knowledge operator $\mathbb{K}_{A_i}$ by $\mathbb{K}^{\pi}_{A_i}$ to denote the reference trace of the knowledge evaluation resulting in the formula φ^* and assume that φ was in negation normal form. The starting formula is then $\psi_n = \varphi^*_n[\pi]$ where $[\pi]$ adds the trace variable π to every atomic proposition.

We now replace every occurrence of knowledge operators starting from n to 1 and from ψ_n with no quantifier prefix. For the jth step, assume the formula to be in the form $\mathbf{Q}.\psi_{j+1}$, where $\mathbf{Q}$ is the quantifier prefix and ψ_{j+1} is the quantifier-free formula from the previous computation. We use fresh propositional variables k and t. For the positive occurrence of $\mathbb{K}^{\pi}_{A_j}\varphi_j$, we translate $\mathbf{Q}.\psi_{j+1}$ to

$$\exists k.\forall t.\Big(Y_j, \vee, \big(\pi \in Y_j \wedge \forall \pi_1 \in Y_j.\forall \pi_2 \in \mathfrak{S}.(equiv_{\mathbb{K}}(\pi_1,\pi_2)\,\mathcal{U}\,t \rightarrow \pi_2 \triangleright Y_j)\big)\Big).$$
$$\forall \pi' \in Y_j.\mathbf{Q}.\psi_{j+1}|_{\mathbb{K}^{\pi}_{A_i}\varphi_j \rightarrow k} \wedge \big((\neg t\,\mathcal{U}(k \wedge t \wedge \bigcirc\Box\neg t)) \rightarrow (\Box(t \rightarrow \varphi_j|_{\pi\rightarrow\pi'}))\big)$$

and for the negated knowledge operator, we translate ψ_i to

$$\exists k.\forall t.\Big(Y_j, \vee, \big(\pi \in Y_j \wedge \forall \pi_1 \in Y_j.\forall \pi_2 \in \mathfrak{S}.(equiv_{\mathbb{K}}(\pi_1,\pi_2)\,\mathcal{U}\,t \rightarrow \pi_2 \triangleright Y_j)\big)\Big).$$
$$\exists \pi' \in Y_j.\mathbf{Q}.\psi_{j+1}|_{\mathbb{K}^{\pi}_{A_i}\varphi_j \rightarrow k}$$
$$\wedge \big((\neg t\,\mathcal{U}(k \wedge t \wedge \bigcirc\Box\neg t)) \rightarrow (\Box(t \rightarrow \neg\varphi_j|_{\pi\rightarrow\pi'}))\big), \text{ where}$$

$$equiv_{\mathsf{K}_{A_i}} := \pi_1 \leftrightarrow_{A_i} \pi_2 \qquad equiv_{\mathsf{C}} := \bigvee_{i\in[n]} \pi_1 \leftrightarrow_{A_i} \pi_2$$

π' is a fresh trace variable, $\psi_{j+1}|_{\neg \mathbb{K}^{\pi}_{A_i} \varphi_j \to k}$ denotes that $\mathbb{K}^{\pi}_{A_i} \varphi_j$ is replaced by k. This adds a new variable for the operator such that the original temporal structure of the knowledge operators is preserved. The formula $\varphi_j|_{\pi \to \pi'}$ changes all occurrences of π, the ones that are not yet bound to a particular knowledge proposition, to the new trace variable π'.

The existentially quantified variable k guesses the points on which the knowledge condition holds for the current operator that will be replaced. Each such point is represented by the universally quantified proposition t. For each t, we build a set of traces Y_j that collects all traces that are equivalent (w.r.t. knowledge *or* common knowledge). We then remove the subformula $\mathbb{K}^{\pi}_{A_i} \varphi_i$ from the formula of the previous step and replace it with the newly introduced k. In the positive case, we have to verify that (common) knowledge over the subformula holds, which is done by the right-hand side of the formula. Whenever the t is chosen as the until formula specifies, namely, it is true at exactly one time point where k is also true, then we verify the formula guarded by the knowledge operator. If not, we immediately evaluate to true. Note that the formula φ_j may include further knowledge operators that we removed before, which is the reason for the strict replacement order of the knowledge operators - innermost before outer. In the negative case, we build the same sets but verify that, for all k-chosen time-points, there exists a trace in the set that is correct on t but violates ψ. The new quantifiers are preceding the previous quantifiers $\mathbf{Q}$ as the old ones are restricted by the new ones. Traversing from n to 1 ensures that we correctly inherit the order of knowledge operators and that we do not add new knowledge operators to the overall formula by changing φ_j. The final formula is $\forall \pi . \mathbf{Q} . \psi_n$, which is the formula obtained after the nth step with an additional universal trace quantifier. This trace quantifier resembles the $\text{LTL}_{\text{K,C}}$ semantics that implicitly quantifies over all traces of a system.

□

5.1.4 Second-Order Quantification on Finite Traces

We propose a finite trace logic for second-order hyperproperties for monitoring whether system behavior is correct or incorrect, called $\text{Hyper}^2\text{LTL}_{\text{f}}$. Monitoring provides a practical and scalable approach to ensuring that a complex system's behavior adheres to its formal specification. In contrast to model checking, monitoring operates directly on the execution traces of the running system rather than analyzing a system model. Monitoring has been extensively explored for various classes of trace properties, including invariants and properties expressible in standard temporal logics like LTL [108, 133, 21].

Hyper^{2}LTL$_\mathrm{f}$ uses second-order trace quantification defined by *fixpoints*, guaranteeing the uniqueness of second-order trace sets and the effective computation of such.

Hyper^{2}LTL$_\mathbf{f}$. Let $\mathcal{V}$ be a finite set of atomic propositions and $\Sigma := 2^{\mathcal{V}}$. We introduce Hyper^{2}LTL$_\mathrm{f}$ as the finite-trace analogue of Hyper^{2}LTL. Let $V = \{\pi, \pi', \pi_1, \ldots\}$ denote a set of (first-order) trace variables, and $\mathfrak{V} = \{X, Y, \ldots\}$ a set of (second-order) set variables. We designate a special second-order variable $\mathfrak{S} \in \mathfrak{V}$ to represent the set of all system traces. Formulas in Hyper^{2}LTL$_\mathrm{f}$ are defined by the following grammar:

$$\begin{aligned}\varphi :=\ & a_\pi \mid \neg\varphi \mid \varphi \wedge \varphi \mid \bigcirc\varphi \mid \ominus\varphi \mid \varphi\, \mathcal{U}\, \varphi \mid \varphi\, \mathcal{S}\, \varphi \\ & \mid \exists \pi \in X.\, \varphi \mid \forall \pi \in X.\, \varphi \mid \exists X.\, \varphi \mid \forall X.\, \varphi\end{aligned}$$

where $a \in \mathcal{V}$, $\pi \in V$ and $X \in \mathfrak{V}$. The logic includes the future temporal operators *next* ($\bigcirc$) and *until* ($\mathcal{U}$), as well as their past counterparts *previously* ($\ominus$) and *since* ($\mathcal{S}$). We also use the standard derived Boolean constants (*true*, *false*), logical connectives ($\vee$, $\rightarrow$, $\leftrightarrow$), and temporal operators such as *eventually* ($\Diamond\varphi := \mathit{true}\, \mathcal{U}\, \varphi$), *once* ($\Diamond\!\!\!-\varphi := \mathit{true}\, \mathcal{S}\, \varphi$), *globally* ($\Box\varphi := \neg\Diamond\neg\varphi$), and *historically* ($\boxminus\varphi := \neg\Diamond\!\!\!-\neg\varphi$).

The semantics of Hyper^{2}LTL$_\mathrm{f}$ is defined with respect to a trace length $m \in \mathbb{N}$ and a set of traces $\mathbb{T} \subseteq \Sigma^m$. We restrict to traces of the same length to obtain a simple semantics. Without this restriction, we would have to deal with combinations of traces with different lengths, making the evaluation more difficult. In practice, we can either pad traces to the length of the longest trace or crop them to the length of the shortest trace, as is done in most HyperLTL monitoring tools [96, 95].

We, again, use a trace assignment $\Pi : V \rightharpoonup \Sigma^m$ mapping trace variables in V to finite traces of length m. Additionally, we maintain a second-order assignment $\Delta : \mathfrak{V} \rightharpoonup 2^{\Sigma^m}$ mapping second-order variables to sets of finite traces of length m. Note that, in contrast to previously defined semantics, the semantics now also explicitly specifies a time-point i since we are not only processing the suffixes of traces but also traversing backwards with past operators. The semantics are defined as follows:

$$\begin{aligned}
&\Pi, \Delta, i \vDash_{\mathbb{T}} a_\pi && \text{iff} \quad a \in \Pi(\pi)(i) \\
&\Pi, \Delta, i \vDash_{\mathbb{T}} \neg\varphi && \text{iff} \quad \Pi, \Delta, i \nvDash_{\mathbb{T}} \varphi \\
&\Pi, \Delta, i \vDash_{\mathbb{T}} \varphi_1 \wedge \varphi_2 && \text{iff} \quad \Pi, \Delta, i \vDash_{\mathbb{T}} \varphi_1 \text{ and } \Pi, \Delta, i \vDash_{\mathbb{T}} \varphi_2 \\
&\Pi, \Delta, i \vDash_{\mathbb{T}} \bigcirc\varphi && \text{iff} \quad i < m - 1 \text{ and } \Pi, \Delta, i + 1 \vDash_{\mathbb{T}} \varphi \\
&\Pi, \Delta, i \vDash_{\mathbb{T}} \ominus\varphi && \text{iff} \quad i > 0 \text{ and } \Pi, \Delta, i - 1 \vDash_{\mathbb{T}} \varphi \\
&\Pi, \Delta, i \vDash_{\mathbb{T}} \varphi_1\, \mathcal{U}\, \varphi_2 && \text{iff} \quad \exists j.\ i \leq j < m.\, \Pi, \Delta, j \vDash_{\mathbb{T}} \varphi_2 \text{ and}
\end{aligned}$$

$$\forall k.\ i \le k < j.\ \Pi, \Delta, k \vDash_{\mathbb{T}} \varphi_1$$

$\Pi, \Delta, i \vDash_{\mathbb{T}} \varphi_1 \mathcal{S} \varphi_2$ iff $\exists j.\ j \le i.\ \Pi, \Delta, j \vDash_{\mathbb{T}} \varphi_2$ and $\forall k.\ i \ge k > j.\ \Pi, \Delta, k \vDash_{\mathbb{T}} \varphi_1$

$\Pi, \Delta, i \vDash_{\mathbb{T}} \exists \pi \in X.\ \varphi$ iff $\exists t \in \Delta(X).\ \Pi[\pi \mapsto t], \Delta, i \vDash_{\mathbb{T}} \varphi$

$\Pi, \Delta, i \vDash_{\mathbb{T}} \forall \pi \in X.\ \varphi$ iff $\forall t \in \Delta(X).\ \Pi[\pi \mapsto t], \Delta, i \vDash_{\mathbb{T}} \varphi$

$\Pi, \Delta, i \vDash_{\mathbb{T}} \exists X.\ \varphi$ iff $\exists A \subseteq \mathbb{T}.\ \Pi, \Delta[X \mapsto A], i \vDash_{\mathbb{T}} \varphi$

$\Pi, \Delta, i \vDash_{\mathbb{T}} \forall X.\ \varphi$ iff $\forall A \subseteq \mathbb{T}.\ \Pi, \Delta[X \mapsto A], i \vDash_{\mathbb{T}} \varphi$

The temporal and Boolean operators are interpreted as usual. An atomic formula a_π holds at position i if the atomic proposition a is true at step i on the trace assigned to π. Quantification over a trace $\pi \in X$ ranges over all traces in the set currently assigned to X by Δ, while quantification over a set variable X considers all subsets of $\mathbb{T}$. A set of traces $\mathbb{T} \subseteq \Sigma^m$ satisfies φ, denoted $\mathbb{T} \vDash \varphi$, if $\emptyset, [\mathfrak{S} \mapsto \mathbb{T}], 0 \vDash_{\mathbb{T}} \varphi$. Here, the special second-order variable $\mathfrak{S}$ is bound to $\mathbb{T}$, so quantification over $\pi \in \mathfrak{S}$ corresponds to quantification over the traces in $\mathbb{T}$, as in HyperLTL.

Fixpoint Formulas. Full second-order quantification is challenging to handle algorithmically: Given n observed finite traces, there are 2^n possible subsets to consider as candidates, making exhaustive checking infeasible. However, most practical properties do not require unrestricted second-order quantification. Instead, the relevant second-order sets are typically characterized as monotone fixpoints. We extend the syntax of Hyper^{2}LTL$_\mathrm{f}$ with an explicit fixpoint operator, enabling concise definitions of such fixpoint-based trace sets. This also allows our monitoring algorithm to leverage the monotonicity inherent in fixpoint constructions. The extended syntax of Hyper^{2}LTL$_\mathrm{f}$ is then the following:

$$\begin{aligned}\varphi :=\ & a_\pi \mid \neg\varphi \mid \varphi \wedge \varphi \mid \bigcirc \varphi \mid \ominus \varphi \mid \varphi\, \mathcal{U}\, \varphi \mid \varphi\, \mathcal{S}\, \varphi \\ & \mid \exists \pi \in X.\ \varphi \mid \forall \pi \in X.\ \varphi \mid \exists X.\ \varphi \mid \forall X.\ \varphi \mid \mathit{fix}(X, \xi_1, \ldots, \xi_k).\ \varphi.\end{aligned}$$

The fixpoint operator $\mathit{fix}(X, \xi_1, \ldots, \xi_k).\ \varphi$ constructs a unique set of traces $X \in \mathfrak{V}$ that can be used in φ. This set X is uniquely defined by the fixpoint constraints $\xi_1, \ldots, \xi_k$ of the form

$$\forall \pi_1 \in X_1 \ldots \forall \pi_n \in X_n.\ \varphi_{step} \rightarrow \pi \in X,$$

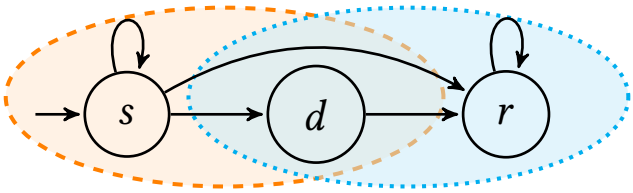


Figure 5.2 A transition system modeling a sender-receiver multi-agent system (MAS). Agent 1 observes only the atomic proposition s and therefore cannot distinguish between the states within the blue (dotted) region. Agent 2 observes only the atomic proposition r and thus cannot distinguish between the states within the orange (dashed) region.

where $X_1, \ldots, X_n \in \mathfrak{V}$, $\pi, \pi_1, \ldots, \pi_n \in V$, and φ_{step} is a quantifier-free formula. To be well-formed, (1) all trace variables used in φ_{step} must either be quantified outside or be one of $\pi_1, \ldots, \pi_n$; (2) all second-order variables $X_1, \ldots, X_n$ must be quantified outside or be equal to X, i.e., within the definition of X we can quantify over traces in X (as is usual for fixpoint definitions); and (3) π must be quantified outside or be one of $\pi_1, \ldots, \pi_n$. Intuitively, the formula states a requirement on traces that should be included in X. If we find traces $t_1 \in X_1, \ldots, t_n \in X_n$ that, together with the sets and traces quantified outside of $\mathit{fix}(X, \xi_1, \ldots, \xi_k).\, \varphi$, satisfy φ_{step}, then trace π should be added to X. In our semantics, we define

$$\Pi, \Delta, i \vDash_{\mathbb{T}} \mathit{fix}(X, \xi_1, \ldots, \xi_k).\, \varphi \text{ iff } \Pi, \Delta\big[X \mapsto \mathit{sol}(\Pi, \Delta, i, X, \xi_1, \ldots, \xi_k)\big], i \vDash_{\mathbb{T}} \varphi,$$

where $\mathit{sol}(\Pi, \Delta, i, X, \xi_1, \ldots, \xi_k)$ denotes the unique solution to the fixpoint definition of X. Formally, $\mathit{sol}(\Pi, \Delta, i, X, \xi_1, \ldots, \xi_k)$ is the *smallest* set of traces such that *for all* ξ_i it holds that

$$\Pi, \Delta[X \mapsto \mathit{sol}(\Pi, \Delta, i, X, \xi_1, \ldots, \xi_k)], i \vDash_{\mathbb{T}} \xi_i.$$

Note that the fixpoint solution $\mathit{sol}(\Pi, \Delta, i, X, \xi_1, \ldots, \xi_k)$ is uniquely determined. For given Π, Δ, i, define $g : 2^{\mathbb{T}} \to 2^{\mathbb{T}}$ as the function that, for a current candidate set X, returns X extended with all traces that must be included according to any of the constraints ξ_i. This function f is monotonic with respect to set inclusion. Each fixpoint of g corresponds to a set X satisfying all $\xi_1, \ldots, \xi_k$. By the Knaster-Tarski theorem [203], g has a unique least fixpoint, which is precisely $\mathit{sol}(\Pi, \Delta, i, X, \xi_1, \ldots, \xi_k)$.

Example 34. We demonstrate the use of our fixpoint construct in an example. Suppose we wish to specify not just that, on all traces where $\Diamond(r \wedge \bigcirc r)$ holds, agent 1 knows that $\Diamond r$ holds, but that it is actually *common knowledge* among agents 1 and 2 that $\Diamond r$ holds.

This requires iteratively constructing the set of all traces that are indistinguishable to some agent. While previous formulas specifying common knowledge required full second-order quantification, we observe here that the relevant set can be defined as a fixpoint:

$$\forall \pi \in \mathfrak{S}.\big(\diamondsuit(r_\pi \wedge \bigcirc r_\pi)\big) \rightarrow \diamondsuit\Big(\mathit{fix}(X, \xi_1, \xi_2).\, \forall \pi' \in X.\, \diamondsuit r_{\pi'}\Big),$$

where $\xi_1 := \mathit{true} \rightarrow \pi \in X$, and ξ_2 is defined as

$$\forall \pi_1 \in X. \forall \pi_2 \in \mathfrak{S}.\, \big(\pi_1 \sim_1 \pi_2 \vee \pi_1 \sim_2 \pi_2\big) \rightarrow \pi_2 \in X.$$

We define $\pi \sim_2 \pi' := \boxminus(r_\pi \leftrightarrow r_{\pi'})$. For each π, we define a set X via a fixpoint construction: X initially contains π (by ξ_1), and is closed under the formula ξ_2: If there exists $\pi_1 \in X$ and $\pi_2 \in \mathfrak{S}$ such that some agent cannot distinguish between π_1 and π_2, then π_2 is also added to X. We then require that every trace π' in X satisfies $\diamondsuit r_{\pi'}$. Common knowledge does not hold for the multi-agent system depicted in Figure 5.2. For example, consider the trace $\pi = sr^n$. It can be seen that

$$sr^n \sim_1 sdr^{n-1} \sim_2 s^2r^{n-1} \sim_1 s^2dr^{n-2} \sim_2 \ldots \sim_2 s^nr \sim_1 s^nd.$$

Consequently, the trace s^nd will be included in X, which is a counterexample to $\diamondsuit r$ being common knowledge on π.

5.2 Monitoring Second-Order HyperLTL

We now focus on *monitoring* a Hyper^{2}LTL$_\mathrm{f}$ formula, where we do not have complete access to the system but instead observe its executions. Based on the set of traces observed so far, we aim to determine whether we can already conclude satisfaction or violation of the property. However, this is not always possible: Certain properties are not *monitorable*, meaning the monitor may never be able to provide a definitive positive or negative answer. We begin by recalling the definitions of monitorability for trace properties, where a single trace is observed and, for every time step, whether a specification holds or is violated is computed.

5.2.1 Monitorability of Trace Properties

Monitorability of a trace property is defined via good prefixes and bad prefixes of infinite traces. We use the standard definition from the literature [67], where monitors observe

traces time-step-wise and compute, for every new time step, whether the property is satisfied, violated, or the evaluations are not yet conclusive. A visualization of a monitor operating on a trace is shown in Figure 5.3a. We are interested in whether a trace property is *monitorable*. Intuitively, a trace property is monitorable if, at some time point, a monitor can switch from inconclusive to *true* or *false*; This is not possible for all trace properties, e.g., for the LTL formula $\square\diamondsuit a$ the variable a has to occur infinitely often, and the monitor could never switch to *true* or to *false* at any time point. We formalize monitorability via good and bad prefixes.

Definition 46 (Good prefixes)**.** *Let $L \subseteq \Sigma^*$ be a trace property. The set of good prefixes is*

$$good(L) := \{u \in \Sigma^* \mid \forall v \in \Sigma^*. uv \in L\}$$

Intuitively, a finite prefix is a *good prefix* if all its extensions satisfy the specification. Similarly, we define the set of bad prefixes.

Definition 47 (Bad prefixes)**.** *Let $L \subseteq \Sigma^*$ be a trace property. The set of bad prefixes is*

$$bad(L) := \{u \in \Sigma^* \mid \forall v \in \Sigma^*. uv \notin L\}$$

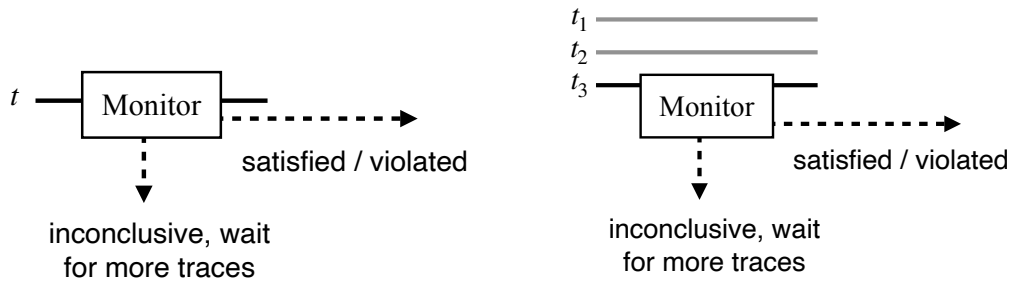
A finite prefix is a *bad prefix* if all its extensions violate the specification. A trace property is then *monitorable* if all prefixes can be extended to either a good prefix or a bad prefix. Most notably, this does not mean that every possibility of satisfying or violating the property must be monitorable, but it suffices if certain traces are monitorable.

Definition 48. *A trace property $L \subseteq \Sigma^*$ is monitorable if*

$$\forall u \in \Sigma^*. \exists v \in \Sigma^*. (uv \in good(L)) \vee (uv \in bad(L)).$$

Example 35. Consider the LTL property $\varphi = \square\diamondsuit a \vee \diamondsuit b$. The first part of the formula is not monitorable: It states that a must occur infinitely often on a trace, meaning the monitor can never conclude if the property is satisfied. However, the second part of φ is monitorable. For any prefix of a trace, once b is observed, the monitor can conclude that the property is satisfied. Regarding Definition 48, the suffix $v = \{b\}$ can be added to any prefix to make it a *good prefix* of φ.

For trace properties, effective algorithms that are based on this definition have been introduced to monitor systems against specifications [22]. In this thesis, however, we



(a) Trace property monitoring. (b) Monitoring hyperproperties sequentially.

Figure 5.3 A visualization for monitoring trace properties in Figure 5.3a and sequentially monitoring hyperproperties in Figure 5.3b. For trace properties, the monitor computes for one trace and every time step if the property is satisfied or violated. In the sequential model for hyperproperty monitoring, an unbounded number of finite traces sequentially arrive at the monitor. For every new trace, the monitor evaluates the formula for the set of traces already observed.

address the monitoring of hyperproperties rather than trace properties, meaning that multiple traces may need to be observed. The way these traces are provided to the monitor depends on the deployment setting, leading to different monitoring models. We consider two primary models: The *parallel model* and the *sequential model* [95]. In the parallel model, the number of traces is fixed in advance, and the monitor observes each time step across all traces simultaneously. In contrast, the sequential model presents traces to the monitor one after another, incrementally increasing the set of observed traces over time. In the following, we discuss both models and analyze their implications for the monitorability of Hyper^{2}LTL$_\mathrm{f}$ properties.

5.2.2 The Parallel Model

In the parallel model, the number of traces (executions) is predetermined and fixed to a bound $b \in \mathbb{N}$. At every time step, the monitor observes the next event from each of these b traces simultaneously. Figure 1.3a illustrates this setup. This approach closely resembles the monitoring of trace properties, where the monitor incrementally receives additional positions for a single, fixed trace.

In the parallel model, since the number of executions (traces) is fixed in advance, second-order quantification does not increase the expressive power of the logic. Intuitively, quantifying over sets of traces can be reduced to quantifying over a finite collection of at most b traces, which can be expressed using first-order trace quantification as in HyperLTL. Formally, for a given bound $b \in \mathbb{N}$ and a Hyper^{2}LTL$_\mathrm{f}$ formula φ, we can systematically rewrite φ into an equivalent formula without second-order

quantifiers, such that the two formulas agree on all sets of traces of size at most b. To do this, we use a partial function $M : \mathfrak{V} \rightharpoonup 2^V$ that maps second-order variables to sets of trace variables. The translation $[\![\varphi]\!]_{b,M}$ is then defined recursively as follows:

$$
\begin{aligned}
[\![a_\pi]\!]_{b,M} &:= a_\pi \\
[\![\circ\varphi]\!]_{b,M} &:= \circ[\![\varphi]\!]_{b,M} \quad \text{for } \circ \in \{\neg, \bigcirc, \ominus\} \\
[\![\varphi_1 \circ \varphi_2]\!]_{b,M} &:= [\![\varphi_1]\!]_{b,M} \circ [\![\varphi_2]\!]_{b,M} \quad \text{for } \circ \in \{\wedge, \mathcal{U}, \mathcal{S}\} \\
[\![\exists X.\, \varphi]\!]_{b,M} &:= \exists \pi_1, \ldots, \pi_b.\, [\![\varphi]\!]_{b,M[X \mapsto \{\pi_1,\ldots,\pi_b\}]} \\
[\![\forall X.\, \varphi]\!]_{b,M} &:= \forall \pi_1, \ldots, \pi_b.\, [\![\varphi]\!]_{b,M[X \mapsto \{\pi_1,\ldots,\pi_b\}]} \\
[\![\exists \pi \in X.\, \varphi]\!]_{b,M} &:= \bigvee_{\pi' \in M(X)} [\![\varphi]\!]_{b,M}[\pi' \leftarrow\!\prec \pi] \\
[\![\forall \pi \in X.\, \varphi]\!]_{b,M} &:= \bigwedge_{\pi' \in M(X)} [\![\varphi]\!]_{b,M}[\pi' \leftarrow\!\prec \pi]
\end{aligned}
$$

In this translation, $\varphi[\pi' \leftarrow\!\prec \pi]$ means that every free occurrence of π in φ is replaced by π'. For formulas without quantifiers, the structure remains unchanged. Because the number of traces is limited to b, second-order quantification over a set X can be rewritten as first-order quantification over b fresh trace variables $\pi_1, \ldots, \pi_b$. The mapping M keeps track of which first-order trace variables correspond to each second-order variable X. When handling first-order quantification $\forall \pi \in X$ (or $\exists \pi \in X$), we replace it with a conjunction (or disjunction) over all $\pi' \in M(X)$, substituting π with each π'. By induction, we obtain:

Proposition 6. *Let φ be any* Hyper^{2}LTL$_\mathrm{f}$ *formula and $b \in \mathbb{N}$. For any set of trace $\mathbb{T}$ with $|\mathbb{T}| \leq b$ we have $\mathbb{T} \vDash \varphi$ iff $\mathbb{T} \vDash [\![\varphi]\!]_{b,\emptyset}$.*

Thus, for any fixed bound $b \in \mathbb{N}$ (as in the parallel model), Proposition 6 shows that monitoring a Hyper^{2}LTL$_\mathrm{f}$ formula φ reduces to monitoring the first-order formula $[\![\varphi]\!]_{b,\emptyset}$. Since $[\![\varphi]\!]_{b,\emptyset}$ contains no second-order quantifiers, existing methods and tools for first-order logics such as HyperLTL [95] can be directly applied.

5.2.3 The Sequential Model

We now turn to the sequential model, which is the primary focus of this section. In this setting, finite traces are provided to the monitor one after another, and there is no fixed upper bound on the total number of traces. This setup is illustrated in Figure 5.3b. Our approach closely follows the sequential model described in [95] for HyperLTL, but differs in that we consider traces of finite length. This emphasis on finite traces makes

our framework and tool particularly suitable for multi-agent system scenarios, where executions are typically finite and correspond to repeated system runs.

Definition 49. *A hyperproperty* $H \subseteq 2^{\Sigma^*}$ *is monitorable in the sequential model if*

$$\forall U \subseteq \Sigma^*. \exists V \subseteq \Sigma^*. \big(\forall W \subseteq \Sigma^*. (U \cup V \cup W) \in H\big) \vee \big(\forall W \subseteq \Sigma^*. (U \cup V \cup W) \notin H\big).$$

In other words, for every set U, there exists a set V such that, after observing $U \cup V$, the monitor can provide a definitive answer: Either all further extensions of $U \cup V$ satisfy H (first disjunct), or all further extensions violate H (second disjunct).

Example 36. Consider the common knowledge formula in Figure 5.2. The formula is monitorable in the sequential model: We can always add traces to ensure that common knowledge does not hold, i.e., add some indistinguishable trace that does not satisfy $\Diamond r$. No matter what additional traces are observed, common knowledge remains violated.

The question of whether a $\text{Hyper}^2\text{LTL}_\text{f}$ formula is monitorable is undecidable:

Theorem 19. *Deciding if a* $\text{Hyper}^2\text{LTL}_\text{f}$ *formula is monitorable in the sequential model is undecidable.*

Proof. It was shown by Finkbeiner et al. [95] that monitorability is undecidable for HyperLTL with finite traces in the sequential model. Since HyperLTL with finite-trace semantics is strictly contained in $\text{Hyper}^2\text{LTL}_\text{f}$, it follows that monitorability for $\text{Hyper}^2\text{LTL}_\text{f}$ is also undecidable. □

It is, therefore, not immediately possible to infer a monitor for $\text{Hyper}^2\text{LTL}_\text{f}$. However, we observe that we can further strengthen the condition from monitorability to a new criterion called *monotonicity*, for which efficient, and more importantly, determined algorithms exist.

5.2.4 Monotonicity in $\text{Hyper}^2\text{LTL}_\text{f}$

Theorem 19 demonstrates that it is impossible to construct a monitoring algorithm for all $\text{Hyper}^2\text{LTL}_\text{f}$ formulas in the sequential model. Therefore, we restrict our attention to fragments of the logic where monitoring is feasible. Given a finite set of observed traces, our goal is to provide a monitoring verdict that remains valid regardless of any additional traces that may be observed in the future. To achieve this, the satisfaction of the formula must be preserved under the addition of new traces; in other words, the property must be *monotone*. We distinguish between $\oplus$-monotonicity and $\ominus$-monotonicity. A formula

is $\oplus$-monotone if, once it is satisfied by a set of traces, it remains satisfied as more traces are added. Conversely, a formula is $\ominus$-monotone if, once it is violated by a set of traces, it remains violated regardless of any further traces.

Definition 50. *A* Hyper^{2}LTL$_\text{f}$ *formula φ is $\oplus$-monotone (resp. $\ominus$-monotone) if for any set of traces $\mathbb{T}$ such that $\mathbb{T} \vDash \varphi$ (resp. $\mathbb{T} \nvDash \varphi$), for any larger set $\mathbb{T}' \supseteq \mathbb{T}$, we have $\mathbb{T}' \vDash \varphi$ (resp. $\mathbb{T}' \nvDash \varphi$).*

A formula is called monotone if it is either $\oplus$-monotone or $\ominus$-monotone. For such formulas, once a $\oplus$-monotone formula is satisfied (or an $\ominus$-monotone formula is violated) by the current set of observed traces, the monitor can immediately conclude that this verdict will not change, regardless of any additional traces that may be observed in the future. For example, the common knowledge formula in Example 34 is $\ominus$-monotone: If → p. 126 the current set of traces violates common knowledge, no additional traces can make the property hold. In many situations, monotonicity ensures monitorability:

Proposition 7. *Let φ be a* Hyper^{2}LTL$_\text{f}$ *formula. If one of the following holds, then φ is monitorable.*

1. *φ is $\oplus$-monotone and has at least one finite model (i.e., a finite set of traces $\mathbb{T}$ such that $\mathbb{T} \vDash \varphi$).*
2. *φ is $\ominus$-monotone and $\neg\varphi$ has at least one finite model.*

This result is an immediate consequence of the definitions of monotonicity (Definition 50) and monitorability (Definition 49). Note, however, that the converse does not hold, even for the first-order fragment of Hyper^{2}LTL$_\text{f}$.

Example 37. As an example, consider the formula

$$\psi := \forall\pi.\exists\pi'.(\pi \neq \pi' \wedge \square(a_\pi \leftrightarrow a_{\pi'})) \vee \exists\pi''.\diamondsuit(b_{\pi''})$$

and the trace sets $\mathbb{T} := \{\{a,c\}^m, \{a\}^m\}$, $\mathbb{T}' := \mathbb{T} \cup \{\{c\}^m\}$, and $\mathbb{T}'' := \mathbb{T}' \cup \{\{b\}^m\}$. Clearly, $\mathbb{T} \vDash \varphi$, $\mathbb{T}' \nvDash \varphi$, and $\mathbb{T}'' \vDash \varphi$. The formula φ can therefore not be monotone. However, φ is monitorable: For every set U, we can the trace set $V := \{\{b\}^m\}$, and every extension of $U \cup V$ satisfies φ (cf. Definition 49).

Our monitoring algorithm leverages monotonicity to deliver definitive verdicts. To statically check the monotonicity of (sub)formulas, we employ a deductive inference system reminiscent of a type system. Judgments are written as $\Gamma \vdash \varphi : \alpha$, where

$$\frac{}{\Gamma \vdash a_\pi : \oplus} \qquad \frac{}{\Gamma \vdash a_\pi : \ominus} \qquad \frac{\Gamma \vdash \varphi : \oplus}{\Gamma \vdash \neg\varphi : \ominus} \qquad \frac{\Gamma \vdash \varphi : \ominus}{\Gamma \vdash \neg\varphi : \oplus}$$

$$\frac{\circ \in \{\bigcirc, \ominus\} \qquad \Gamma \vdash \varphi : \alpha}{\Gamma \vdash \circ\varphi : \alpha} \qquad \frac{\circ \in \{\wedge, \mathcal{U}, \mathcal{S}\} \qquad \begin{array}{c}\Gamma \vdash \varphi_1 : \alpha \\ \Gamma \vdash \varphi_2 : \alpha\end{array}}{\Gamma \vdash \varphi_1 \circ \varphi_2 : \alpha}$$

$$\frac{X \in \Gamma \qquad \Gamma \vdash \varphi : \oplus}{\Gamma \vdash \exists \pi \in X.\, \varphi : \oplus} \qquad \frac{X \in \Gamma \qquad \Gamma \vdash \varphi : \ominus}{\Gamma \vdash \forall \pi \in X.\, \varphi : \ominus}$$

$$\frac{\Gamma \cup \{X\} \vdash \varphi : \alpha}{\Gamma \vdash \mathit{fix}(X, \xi_1, \ldots, \xi_k).\, \varphi : \alpha} \qquad \frac{\Gamma \vdash \varphi : \alpha}{\Gamma \vdash \mathbb{Q}X.\, \varphi : \alpha}$$

Figure 5.4 The inference system used in Algorithm 5 to infer the monotonicity of a Hyper^{2}LTL$_\mathrm{f}$ formula.

$\alpha \in \{\oplus, \ominus\}$ and $\Gamma = \{X_1, \ldots, X_n\}$ is a set of context assumptions. Here, $X \in \Gamma$ indicates that X has a unique model that can only grow as more traces are observed (i.e., its model is monotonically increasing). The judgment $\Gamma \vdash \varphi : \oplus$ (resp. $\Gamma \vdash \varphi : \ominus$) means that, under the assumptions in Γ, φ is $\oplus$-monotone (resp. $\ominus$-monotone). The inference rules for this system are shown in Figure 5.4. Most rules are straightforward: Atomic propositions are both $\oplus$- and $\ominus$-monotone, temporal operators preserve monotonicity, and negation inverts monotonicity.

The most interesting cases are the rules for quantification. A formula $\varphi = \exists \pi \in X.\, \varphi'$ is $\oplus$-monotone if φ' is $\oplus$-monotone and $X \in \Gamma$ (i.e., the interpretation of X can only increase): suppose φ holds for the current set of traces $\mathbb{T}$, meaning there is a trace $t \in X$ that witnesses the satisfaction of φ'. Since $X \in \Gamma$, the set of traces assigned to X can only grow as more traces are observed, so t remains a valid witness. Similarly, for fixpoint-defined second-order variables X, their interpretations only expand as new traces are added, so we include X in Γ. In contrast, for full second-order quantification, we cannot make assumptions about how the interpretation of variables changes with additional traces, so no assumptions are added to the context. In our inference system, we start with the context $\Gamma = \{\mathfrak{S}\}$, since the set of all traces (represented by the special variable $\mathfrak{S} \in \mathfrak{V}$) only increases. A straightforward induction yields:

Proposition 8. *Assume $\{\mathfrak{S}\} \vdash \varphi : \oplus$ (resp. $\{\mathfrak{S}\} \vdash \varphi : \ominus$), then φ is $\oplus$-monotone (resp. $\ominus$-monotone).*

Algorithm 5: Monitoring Hyper^{2}LTL$_\mathrm{f}$ formula φ

```
1 let monitor(φ) :=
2   let monMap = computeMonotonicity(φ)
3   let 𝕋 = ∅
4   while t = getNextTrace(): // Obtain current execution
5     𝕋 = 𝕋 ∪ {t}
6     let res = check(∅, ∅, 0, 𝕋, φ) // Check the current set of traces against φ
7     if res = true and ⊕ ∈ monMap(φ) then return SAT // True positive
8     if res = false and ⊖ ∈ monMap(φ) then return UNSAT // True negative
```

It is important to note that our inference system is not complete; that is, the converse of Proposition 8 does not necessarily hold. Nevertheless, the system enables us to verify, for example, that the common knowledge property in Example 34 is $\ominus$-monotone.

5.2.5 Monitoring Algorithm

Algorithm 5 presents our basic monitoring algorithm for a formula φ. Initially, we (1) compute a monotonicity map *monMap* using the inference system from Figure 5.4, where *monMap* assigns to each subformula φ' of φ a set $\mathit{monMap}(\varphi') \in 2^{\{\oplus,\ominus\}}$ indicating whether φ' is monotone in the current context Π, Δ, i; and (2) initialize the set of observed traces $\mathbb{T}$ as empty. As the monitor runs, each newly observed trace t is added to $\mathbb{T}$, and we check whether $\mathbb{T}$ satisfies φ by invoking the check function. As discussed in Section 5.2.4, the monitor can only return a definitive SAT verdict if φ is $\oplus$-monotone and the current traces satisfy φ, or UNSAT if φ is $\ominus$-monotone and the current traces violate φ. If neither case applies, the monitor waits for additional traces and repeats.

Incremental Model Checking. At the heart of our monitoring approach is the recursive model-checking function check, shown in Algorithm 6. Conceptually, the algorithm check implements the semantics of Hyper^{2}LTL$_\mathrm{f}$ as an executable procedure. For clarity, we temporarily disregard the code segments highlighted in gray, which pertain to hashing-based optimizations discussed later in Section 5.2.5. In Line 6, the function performs pattern matching on the structure of φ. We present only a subset of cases here, as the remaining ones follow analogously. When φ is an atomic proposition, its truth value can be determined directly (Line 7); for negation, conjunction, and temporal operators, the function proceeds via the expected recursive calls. For first-order quantification (Line 9), the function iterates over all traces assigned by Δ. In the case of full second-order quantification (Line 16), it considers all subsets of $\mathbb{T}$. For

Algorithm 6: Incremental model-checking

```
1  let check(Π,Δ,i,𝕋,φ):=
2   if ⊕ ∈ monMap(φ) and h_sat(Π, Δ, i, φ) = true then
3    return true // True positive hashed
4   if ⊖ ∈ monMap(φ) and h_sat(Π, Δ, i, φ) = false then
5    return false // True negative hashed
6   let res = match φ with  // Recursive checking of the formula
7    | a_π: if a ∈ Π(π)(i) then return true else return false
8    | ¬φ′: return (not (check(Π, Δ, i, 𝕋, φ′)))
9    | ∃π ∈ X. φ′: // Witness hashing
10    A = if h_wit(Π, Δ, i, φ) = t then order(t, Δ(X)) else Δ(X)
11    for t in A:  // Try to find witness trace
12     if check(Π[π ↦ t], Δ, i, 𝕋, φ′) then
13      h_wit(Π, Δ, i, φ) = t  // Witness found
14      return true
15    return false
16   | ∃X. φ′:
17    for A ⊆ 𝕋: // Try to find witness trace set
18     if check(Π, Δ[X ↦ A], i, 𝕋, φ′) then
19      return true
20    return false
21   | fix(X, ξ_1, ..., ξ_k). φ′:
22    A′ = if h_fix(Π, Δ, i, fix(X, ξ_1, ..., ξ_k)) = A″ then A″ else ∅ // Fixpoint hashed
23    A = computeFix(Π, Δ, i, 𝕋, fix(X, ξ_1, ..., ξ_k),A′) // Fixpoint computation
24    h_fix(Π, Δ, i, fix(X, ξ_1, ..., ξ_k)) = A  // Hash the fixpoint
25    return check(Π, Δ[X ↦ A],i,𝕋, φ′)
26  h_sat(Π, Δ, i, φ) = res
27  return res
```

fixpoint-based second-order quantification (Line 21), the function computes the unique fixpoint solution (Line 23) using computeFix, and then recursively checks the remainder of the formula (Line 25).

Fixpoint Computation. As discussed in Section 5.1.4, fixpoints offer sufficient expressiveness for most practical properties while being significantly more tractable algorithmically. Given a set $\mathbb{T}$ of n traces, the fixpoint can be computed in polynomial time using Knaster-Tarski fixpoint iteration [203], in contrast to the exponential 2^n possibilities required for full second-order quantification. Algorithm 7 describes the fixpoint computation procedure. The algorithm receives the fixpoint formula and the current candidate set A (initially $A = \emptyset$). The second-order assignment Δ is updated to map X to A (Line 2). The algorithm then checks whether A satisfies the fixpoint

→ p. 124

Algorithm 7: Fixpoint computation

```
1 let computeFix(Π, Δ, i, 𝕋, fix(X, ξ_1, ..., ξ_k),A):
2  let Δ' = Δ[X ↦ A]  // Initialization with previously computed fixpoint
3  for ∀π_1 ∈ X_1 ... ∀π_n ∈ X_n. φ_step → π ∈ X in {ξ_1, ..., ξ_k}:
4    for t_1 ∈ Δ'(X_1), ..., t_n ∈ Δ'(X_n): // All possible instantiations
5      let Π' = Π[π_1 ↦ t_1, ..., π_n ↦ t_n]
6      if Π'(π) ∉ A and check(Π', Δ', i, 𝕋, φ_step) then  // Find member of the fixpoint
7        return computeFix(Π, Δ, i, 𝕋, fix(X, ξ_1, ..., ξ_k), A ∪ {Π'(π)})  // Next fixpoint
8  return A
```

constraints $\xi_1, \ldots, \xi_k$: For each constraint, it searches for traces $t_1 \in \Delta(X_1), \ldots, t_n \in \Delta(X_n)$ that satisfy the step condition (checked recursively in Line 6). If such a trace is found, it is added to A, and the process repeats recursively (Line 7).

Optimizations. Without optimizations, Algorithm 6 evaluates the Hyper^{2}LTL$_\mathrm{f}$ formula on the entire current set of traces, repeating this process from scratch for each new observed trace. This approach quickly becomes impractical, as the verification time grows with the number of traces. To achieve efficient monitoring, it is crucial to reuse intermediate results from previous iterations: When a new trace t is added to the set $\mathbb{T}$, we aim to leverage as much of the prior computation on $\mathbb{T}$ as possible. We identify three main opportunities for such reuse: Caching verification results for subformulas, caching fixpoint computations, and caching witnesses.

SAT Hashing. Our first optimization is to cache the results of the check function for *subformulas* using hashing. Specifically, for any evaluation context (Π, Δ, i), if a $\oplus$-monotone (resp. $\ominus$-monotone) subformula φ has previously been determined to be satisfied (resp. violated), then this verdict remains unchanged for all future iterations under the same assignments. To exploit this, we introduce a hash function h_{sat} (initially empty) that records the outcome for each tuple $(\Pi, \Delta, i, \varphi)$, as shown in Algorithm 6, Lines 2 and 4. For example, $h_{sat}(\Pi,\Delta,i,\varphi) = \mathrm{true}$ indicates that $(\Pi, \Delta, i, \varphi)$ is stored in h_{sat} with value true. If no cached result is available, φ is evaluated and the result is stored in h_{sat} in Line 26.

Fixpoint Hashing. The second optimization involves caching fixpoint solutions. For any given context (Π, Δ, i), the fixpoint solution can only grow as new traces are added to $\mathbb{T}$. Therefore, when recomputing a fixpoint (Algorithm 6, Line 23), it is unnecessary to start from scratch. Instead, we use a hash function h_{fix} to store previously computed

fixpoint solutions. If a solution for $(\Pi, \Delta, i, \mathit{fix}(X, \xi_1, \ldots, \xi_k))$ is already cached, we use it as the initial set for the current computation; otherwise, we begin with the empty set (Line 22). After invoking computeFix, we update h_{fix} to map $(\Pi, \Delta, i, \mathit{fix}(X, \xi_1, \ldots, \xi_k))$ to the newly computed solution A (Line 24).

Witness Hashing. Our final hashing-based optimization targets the handling of first-order quantification. In Line 11, all traces in the current model must be considered. Although this cannot be avoided in the general case, we can improve efficiency by adjusting the order in which traces are explored. Specifically, when a witness trace t is found, it is cached using the h_{wit} hash function (Line 13). During subsequent iterations, we reorder $\Delta(X)$ so that the previously discovered witness t is checked first (Line 10). The rationale is that if t served as a witness for the satisfaction of a formula $\exists \pi.\varphi'$ in a prior iteration, it is likely to remain a suitable candidate as more traces are added to $\mathbb{T}$; prioritizing t can therefore lead to performance gains.

Prefix and Postfix Trees. Our final optimization enhances all the aforementioned hashing techniques. Rather than representing the set $\mathbb{T}$ of traces as a flat list, we organize it as a tree structure, grouping traces by their shared prefixes (or postfixes). This approach offers two main benefits: First, it reduces memory usage, especially when many traces have overlapping segments. Second, it enables more efficient hashing operations. For instance, when computing a fixpoint at time step i that involves only future temporal operators (i.e., without past modalities). For the purpose of fixpoint computation, all traces that share the same suffix from step i onward can be considered equivalent. This allows us to perform computations directly on nodes of the postfix tree, rather than on individual traces.

5.2.6 Experiments

We have developed an implementation of our monitoring algorithm, including the optimizations described in Section 5.2.5, in a tool named `MoSo`. In this section, we show

Table 5.1 We monitor common knowledge in Example 34 for varying trace lengths and trace numbers (**# traces**). We report `MoSo`'s average runtime in seconds (***t***).

length	20	30	40	50	60	70	80
# traces	35	55	75	95	115	135	155
t	0.51	1.51	5.98	18.40	48.83	111.52	230.71

# children	2	3	4	5	6	7	8	9
$\boldsymbol{t}$	0.1	0.2	0.2	0.2	0.3	0.4	0.4	0.4
$\boldsymbol{t}_{noFix}$	0.1	1.1	TO	TO	TO	TO	TO	TO

Table 5.2 We evaluate the monitoring of common knowledge in the muddy children puzzle. The table shows the average runtime of MoSo in seconds ($\boldsymbol{t}$). For comparison, we also include the runtime for the non-fixpoint-based encoding of common knowledge ($\boldsymbol{t}_{noFix}$). A timeout (TO) is set to 1 minute.

that MoSo is capable of monitoring complex second-order hyperproperties that are beyond the capabilities of existing monitoring and model-checking tools.

Running Example. We first evaluate the scalability of MoSo with respect to trace length. Specifically, we monitor the formula from Example 34 on traces generated from the example in Figure 5.2. For each instance, we generate traces of lengths 20, 30, 40, ..., 80 and record the time required to detect a violation for the initial trace $s^n r^n$, as shown in Table 5.1. In all cases, the monitor successfully detects that the common knowledge formula is violated.

→ p. 126

Common Knowledge in Multi-Agent Systems. We evaluate our monitor on the classic muddy children puzzle [81] as presented in Section 5.1.1. In this multi-agent system, n children participate in rounds of communication to establish common knowledge about which of them are muddy. Each child i is associated with an atomic proposition m_i (indicating if i is muddy) and d_i (indicating if i has declared being muddy). We formalize the property that, after $b \in \mathbb{N}$ rounds of communication (the *communication-bound*), the muddiness of all children is common knowledge using the following Hyper^{2}LTL$_\mathrm{f}$ formula φ_{mud}:

→ p. 113

$$\forall \pi \in \mathfrak{S}. \bigcirc^b \mathit{fix}\big(X, \xi_1, \xi_2\big). \forall \pi_1 \in X. \forall \pi_2 \in X. \bigwedge_{a \in \{m_1, \ldots, m_n\}} \big(a_{\pi_1} \leftrightarrow a_{\pi_2}\big),$$

where $\xi_1 := \mathit{true} \rightarrow \pi \in X$, and ξ_2 is defined as

$$\forall \pi_1 \in X. \forall \pi_2 \in \mathfrak{S}. \Big(\bigvee_{i \in \{1,\ldots,n\}} \boxminus \Big(\bigwedge_{\substack{a \in \{d_1,\ldots,d_n\} \cup \\ \{m_1,\ldots,m_{i-1}\} \cup \\ \{m_{i+1},\ldots,m_n\}}} a_{\pi_1} \leftrightarrow a_{\pi_2} \Big) \Big) \rightarrow \pi_2 \in X.$$

Intuitively, the fixpoint collects all traces that, for some child i, are observationally indistinguishable from π during the first b steps—that is, traces that match on all atomic propositions except possibly m_i. The formula φ_{mud} then asserts that all traces in X must agree on the muddiness status of every child. The muddy children system fails to satisfy φ_{mud} precisely when $b < n$.

Scalability. To evaluate scalability, we sample random traces for the game with n children and use MoSo to monitor common knowledge, setting the communication-bound to $b = \lceil \frac{n}{2} \rceil$ so that common knowledge does not hold. The runtimes are shown in Table 5.2. Even as the number of children increases, our monitor efficiently detects violations of common knowledge. In contrast, existing model-checking approaches for common knowledge [35] are currently limited to $n = 4$. This demonstrates that monitoring second-order hyperproperties provides a lightweight and scalable alternative in scenarios where full verification is impractical.

Fixpoints vs Second-Order Sets. The muddy children puzzle also demonstrates the practical advantage of fixpoint-based encodings over general second-order quantification. To evaluate this, we monitor the same common knowledge property, but encode it using unrestricted second-order quantifiers (as in Section 5.1.1). The runtimes for this non-fixpoint encoding are shown in Table 5.2 ($\boldsymbol{t}_{noFix}$). With general second-order quantification, MoSo already reaches a timeout for 4 children, due to the exponential blowup from enumerating all subsets of traces. This highlights that fixpoint encodings are essential for scalable monitoring of such properties.

Table 5.3 We depict the (average) number of traces the monitor processes before concluding a violation of common knowledge.

		communication-bound					
		0	**1**	**2**	**3**	**4**	**5**
# children	**2**	1.3	7.0	-	-	-	-
	3	2.1	4.5	15.0	-	-	-
	4	2.1	4.2	5.3	31.0	-	-
	5	2.9	3.9	5.7	33.1	63.0	-
	6	4.3	4.3	8.1	12.7	43.3	127.0

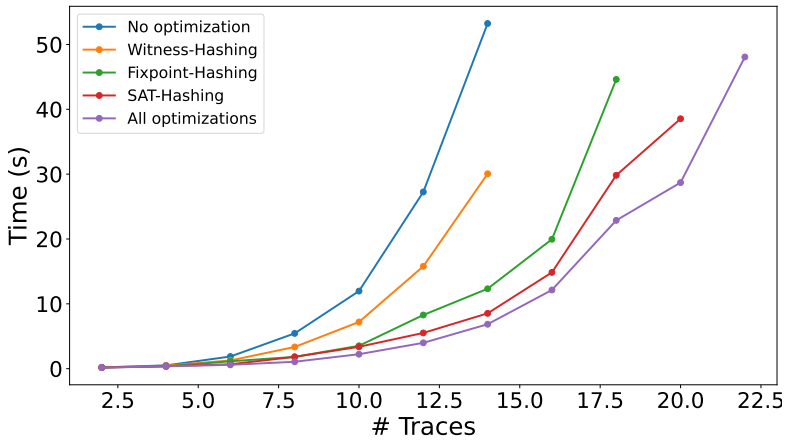


Figure 5.5 We compare the optimizations implemented in MoSo as a cactus plot. The timeout is set to 60 seconds.

Impact of Communication-bound. The ability of the monitor to detect violations of the common knowledge formula is strongly influenced by the communication-bound. As the communication-bound increases, each child observes more information, making it less likely to sample traces that violate common knowledge. Consequently, the monitor typically needs to observe more traces before it can conclude that common knowledge does not hold. To evaluate this effect, we empirically monitor the muddy children puzzle for different numbers of children and communication bounds, recording the average number of traces (over 10 runs) required for the monitor to detect a violation. The results are shown in Table 5.3. Clearly, a higher communication bound leads to an increase in the number of traces processed before a violation is reported.

Impact of Optimizations. Common knowledge also serves as a useful baseline to highlight the importance of our hashing-based optimizations. To demonstrate this, we check the following common knowledge formula

$$\exists \pi \in \mathfrak{S}. \Diamond \mathit{fix}\Big(X, \mathit{true} \rightarrow \pi \in X, \forall \pi_1 \in X. \forall \pi_2 \in \mathfrak{S}. \\ \big(\boxminus(a_{\pi_1} \leftrightarrow a_{\pi_2}) \vee \boxminus(b_{\pi_1} \leftrightarrow b_{\pi_2})\big) \rightarrow \pi_2 \in X\Big). \forall \pi' \in X. c_{\pi'},$$

which states that on *some* path π, c is eventually common knowledge, given that one agent observes AP a and one agent observes AP b. We use MoSo to check the above formula on randomly generated instances with a varying number of traces. We depict the results in Figure 5.5. We observe that all optimizations improve upon the baseline.

	size	55	70	85	100	115	130
t-sen.	***t***	1.66	4.08	5.44	38.1	35.52	88.19
	# traces	1.03	1.76	3.2	3.91	4.86	5.04
t-ins.	***t***	45.2	61.45	72.15	112.15	76.5	137.95
	# traces	52.15	77.35	98.25	83.95	125.75	104.55

Table 5.4 We analyze the detection of a path from source to target by observing random paths in randomly generated graphs. For each graph size, we generate 1000 random traces and report the average number of traces processed by MoSo to reach a verdict (**# traces**) as well as the average runtime in seconds (***t***).

Planning Analysis. In our final experiment, we apply our monitor to the problem of detecting reachability in a graph by observing random paths. For instance, if we observe the traces t_1 = abc and t_2 = bbd, we can infer the existence of the trace t = abd in the graph, even if it has not been directly observed. Note that each atomic proposition identifies a state in the graph. This inference can be naturally expressed using a fixpoint constraint. We distinguish between time-sensitive (**t-sen.**) and time-insensitive (**t-ins.**) paths. In the time-sensitive setting, two paths can be combined only if they visit the same state at the same time step. In contrast, the time-insensitive setting allows combining paths that visit the same state at possibly different time steps. Table 5.4 reports the average runtime and the number of traces required to detect a valid plan. As expected, the time-insensitive case typically requires fewer traces to reach a verdict, since more combinations are possible and the likelihood of inferring a path from source to target increases.

Chapter 6

Discussion

We conclude with a summary of the main contributions of this thesis and an outlook on both immediate and long-term directions for future research. This thesis demonstrated that incorporating reasoning about knowledge and information flow fundamentally improves the design of algorithms for correct-by-construction systems. Focusing on distributed systems and their necessary decomposition for scalability, we have presented several new solutions for reactive synthesis, one of the most challenging problems in formal methods, that explicitly leverage knowledge and information flow. The improvements in scalability show the potential of knowledge-based approaches for decomposing distributed systems. To extend our reasoning techniques and algorithms to more complex properties, particularly common knowledge, we have extended existing temporal hyperlogics to express second-order hyperproperties. The new logics and monitoring algorithm lay the foundation for further reasoning techniques and algorithms to achieve the ultimate verification goal: Formal guarantees for correct distributed systems.

6.1 Conclusion

Distributed Synthesis. The presented compositional synthesis algorithms introduce a new class of distributed synthesis methods, where assumptions are formulated in terms of information flow and expressed as hyperproperties. Unlike previous work, we concentrated on the necessary assumptions required for assume-guarantee synthesis, which turn out to be hyperproperties. We developed techniques for analyzing the necessary information flow between components based only on the desired specification. Based on the computed information flow, we showed how to construct effective information-flow assumptions for both liveness and safety specifications. For liveness

specifications, we assumed an arbitrary time-bound until which the information was received. For safety, we compute specification-defined deadlines after which the information will be present to the receiver. For both approaches, we designed compositional synthesis algorithms that leverage these hyperproperty assumptions during the local synthesis of system components. After the local synthesis, an additional composition and decomposition step refines the local solutions to overall solutions to the distributed synthesis problem.

Controller Synthesis. We introduced a novel controller synthesis method that abstracts away from the specific size of the plant by constructing a universal controller, one that is correct for any plant where the synthesis problem is realizable. Prophecies, which are knowledge-based assumptions about the plant's implementation, enable optimizations that are independent of the particular plant. When composing with the explicit plant, these optimizations significantly reduce the exploration of the plant's state space. Our results demonstrate that this approach is compositional, complete, and achieves improved execution times in our experiments.

Logics for Hyperproperties. Knowledge and information flow are hyperproperties that can be expressed by existing logics for hyperproperties, logics that quantify over the traces on which relational properties are required to hold, called first-order quantification. Motivated by the property of *common knowledge*, a shared agreement that is widely used in the verification of distributed protocols, this thesis extended hyperlogics beyond first-order quantification, like HyperLTL, by investigating second-order hyperproperties. Second-order hyperproperties require quantification over sets of traces rather than individual traces. We introduced Hyper^{2}LTL as a natural and expressive language for specifying second-order hyperproperties. Hyper^{2}LTL provides a general framework that captures a wide range of relevant hyperproperties, including common knowledge and asynchronous hyperproperties, which previously required specialized logics for specification.

Monitoring Second-Order Hyperproperties. We further explored algorithms for second-order hyperproperties by introducing a monitoring algorithm that verifies whether a system satisfies a specification during its execution. In a first step, we introduced Hyper^{2}LTL$_\text{f}$, the first logic capable of expressing second-order hyperproperties over finite traces. We categorized a subclass of monitorable second-order hyperproperties, for which we designed an algorithm that automatically verifies executions of a system. Our experimental results demonstrate that the proposed monitoring algorithm

scales to large problems. Consequently, monitoring complex and practically relevant second-order hyperproperties, especially in the multi-agent system domain, offers a promising compromise between rigorous guarantees and scalability.

6.2 Future Work

The key direction for future work is to expand the application of knowledge and information flow reasoning to a broader range of algorithms for verifying distributed systems. While this thesis has demonstrated the benefits of such reasoning in the synthesis and monitoring of distributed systems, similar reasoning principles can be leveraged to improve general verification techniques, enabling more scalable and expressive analysis of complex applications. By systematically integrating knowledge-based and information-flow reasoning into verification algorithms, we can address major challenges: Compositionality, scalability, and the specification of rich system properties. This has the potential to advance the state of the art in distributed system verification in many application domains.

6.2.1 Algorithms

We first outline possible extensions and open problems of the algorithms presented in this thesis.

Assumptions. While compositional synthesis benefits from the expressiveness of hyperproperties, synthesizing from hyperproperties is significantly more difficult than from trace properties. To mitigate this, Section 3.3 and Section 3.5 present more practical approaches that substitute the hyperproperties with approximating trace properties that can be synthesized more easily. However, this approach is currently limited to information flow assumptions involving only a finite amount of information in the liveness case and information flow assumptions that can be separated into a finite number of information classes in the safety case. In practice, it is often necessary to handle cases where an unbounded amount of information must be repeatedly transmitted. We believe that our method can be generalized to accommodate such scenarios, where the overarching challenge is the sound approximation of hyperproperty synthesis by trace property synthesis.

→ p. 52

Components. A second important direction is to extend our approach to distributed systems involving more than two processes. While the two-process scenario allows

each process's assumptions to be directly guaranteed by the other, and is a useful proof of concept for information-flow assumptions, the situation becomes more complex with additional processes. In such cases, localizing assumptions is more challenging or may even be infeasible, especially when multiple processes have (partial) access to the necessary information. A possible solution is *negotiation* algorithms, where processes refine assumptions in a turn-based manner. During negotiation, assumptions can be safely strengthened without introducing contradictions, and components can be assigned to provide certain information flow. Extending the algorithms in this thesis to negotiation algorithms is a promising next step for the synthesis of distributed systems with many components.

Prophecies. Prophecies, as introduced in Section 4.1.3, are a general concept of assumptions over plant behavior. We show that, for safety LTL formulas as goal properties, we can construct necessary and sufficient prophecies represented by tree automata. While preserving the completeness of the problem, this is a complex formalism with little tool support. Obvious next steps include finding prophecy representations that are similarly useful but easier to minimize and verify, e.g., the branching-time logic CTL. While CTL cannot satisfy the condition of completeness, CTL formulas are more succinct and easier to verify than tree automata. Another obvious next step is algorithms for constructing prophecies for liveness. Here, the main problem is to verify progress in the plant, meaning that the prophecy must include all plants that reach accepting states arbitrarily late. Finding an algorithm and proper representation for such prophecies remains an open problem. → p. 96

Logics. Hyperlogics are actively developed to further support properties in different system formalisms. Regarding Hyper^{2}LTL, the main question is whether there exist efficient fragments of the language that capture relevant properties and can be verified easily. Possible underlying systems can include cyber-physical, strategic, and asynchronous systems, where multiple-component systems add another layer of complexity. While Hyper^{2}LTL is most expressive and general, lifting the concept of second-order properties to large-scale systems, as in hardware and program verification, by adapting the logic, is interesting future work.

6.2.2 Applications

To conclude, we highlight promising applications of knowledge and information-flow reasoning for the practical development and verification of systems. These approaches

have the potential to enhance specification-driven analyses of system requirements and strengthen verification techniques, particularly in the context of security.

System Development. Our reasoning techniques can have a positive impact on the semi-automatic construction of correct system implementations. While our work has demonstrated the effectiveness of these techniques for synthesis, a fully automatic approach, more practical, application-oriented methods for generating correct solutions can also benefit from our algorithms: A promising application area is hardware-software contracts [124], which specify the interface between hardware and software and ensure certain guarantees. Such contracts have already been adapted to mitigate side-channel attacks, and information-flow analyses can further enhance their formalism. A similar perspective applies to programming languages for asynchronous distributed systems, where components operate independently rather than in lock-step. Widely used languages for distributed system design, such as P [71], could leverage information-flow and knowledge-based analyses to improve requirement specification and implementation evaluation prior to deployment.

Information Flow Security. Information flow security is a primary application area for hyperproperty specification and verification, and information flow itself was a key motivation for the development of hyperlogics. Reasoning about knowledge and information flow enables more nuanced correctness analyses for security properties: Rather than verifying a single property, one can check whether an attacker knows or does not know a particular fact, which has important implications for security analysis. Additionally, analyzing the required information flow for a given functional property provides a generalized way to determine whether certain information flows are permissible. Any additional information flow, such as that caused by timing side-channels, should be prevented. For example, in a password checker, the fact that a password is correct must necessarily be revealed when the correct password is entered; This represents necessary information flow, which is closely related to declassification [193]. A different potential application area is zero-knowledge proofs [85], distributed systems that communicate about a shared secret over a public channel, without revealing it to an observer. Our information-flow analysis is applicable as the components can be checked to distinguish the different secrets, whereas they should be indistinguishable for the observer. Lastly, the security of machine learning models that support decision-making algorithms is widely applied but can unintentionally reveal secret information about components, e.g., inferring salaries by observing personal attributes in hiring

algorithms. Monitoring the information flow and the knowledge of observers during the execution of the hiring procedure could shield this information from an attacker. Our algorithms can help to further enhance the security of systems both before and after deployment.

Bibliography

[1] Martín Abadi and Leslie Lamport. "The Existence of Refinement Mappings". In: *Theor. Comput. Sci.* 82.2 (1991), pp. 253–284. DOI: 10.1016/0304-3975(91)90224-P. URL: https://doi.org/10.1016/0304-3975(91)90224-P.

[2] Martín Abadi, Leslie Lamport, and Pierre Wolper. "Realizable and Unrealizable Specifications of Reactive Systems". In: *Automata, Languages and Programming, 16th International Colloquium, ICALP89, Stresa, Italy, July 11-15, 1989, Proceedings*. Ed. by Giorgio Ausiello, Mariangiola Dezani-Ciancaglini, and Simona Ronchi Della Rocca. Vol. 372. Lecture Notes in Computer Science. Springer, 1989, pp. 1–17. DOI: 10.1007/BFB0035748. URL: https://doi.org/10.1007/BFb0035748.

[3] S. Akshay, Eliyahu Basa, Supratik Chakraborty, and Dror Fried. "On Dependent Variables in Reactive Synthesis". In: *Tools and Algorithms for the Construction and Analysis of Systems - 30th International Conference, TACAS 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part I*. Ed. by Bernd Finkbeiner and Laura Kovács. Vol. 14570. Lecture Notes in Computer Science. Springer, 2024, pp. 123–143. DOI: 10.1007/978-3-031-57246-3_8. URL: https://doi.org/10.1007/978-3-031-57246-3%5C_8.

[4] Alexandre Duret-Lutz, Etienne Renault, Maximilien Colange, Florian Renkin, Alexandre Gbaguidi Aisse, Philipp Schlehuber-Caissier, Thomas Medioni, Antoine Martin, Jérôme Dubois, Clément Gillard, and Henrich Lauko. "From Spot 2.0 to Spot 2.10: What's New?" In: *Proceedings of the 34th International Conference on Computer Aided Verification (CAV'22)*. Vol. 13372. Lecture Notes in Computer Science. Springer, Aug. 2022, pp. 174–187. DOI: 10.1007/978-3-031-13188-2_9.

[5] Rajeev Alur, Salar Moarref, and Ufuk Topcu. "Compositional Synthesis of Reactive Controllers for Multi-agent Systems". In: *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*. Ed. by Swarat Chaudhuri and Azadeh Farzan. Vol. 9780. Lecture Notes in Computer Science. Springer, 2016, pp. 251–269. DOI: 10.1007/978-3-319-41540-6_14. URL: https://doi.org/10.1007/978-3-319-41540-6%5C_14.

[6] Rajeev Alur, Salar Moarref, and Ufuk Topcu. "Pattern-Based Refinement of Assume-Guarantee Specifications in Reactive Synthesis". In: *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference,*

TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings. Ed. by Christel Baier and Cesare Tinelli. Vol. 9035. Lecture Notes in Computer Science. Springer, 2015, pp. 501–516. DOI: 10.1007/978-3-662-46681-0_49. URL: https://doi.org/10.1007/978-3-662-46681-0%5C_49.

[7] Ashwani Anand, Kaushik Mallik, Satya Prakash Nayak, and Anne-Kathrin Schmuck. "Computing Adequately Permissive Assumptions for Synthesis". In: *Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Paris, France, April 22-27, 2023, Proceedings, Part II*. Ed. by Sriram Sankaranarayanan and Natasha Sharygina. Vol. 13994. Lecture Notes in Computer Science. Springer, 2023, pp. 211–228. DOI: 10.1007/978-3-031-30820-8_15. URL: https://doi.org/10.1007/978-3-031-30820-8%5C_15.

[8] Ashwani Anand, Satya Prakash Nayak, and Anne-Kathrin Schmuck. "Synthesizing Permissive Winning Strategy Templates for Parity Games". In: *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part I*. Ed. by Constantin Enea and Akash Lal. Vol. 13964. Lecture Notes in Computer Science. Springer, 2023, pp. 436–458. DOI: 10.1007/978-3-031-37706-8_22. URL: https://doi.org/10.1007/978-3-031-37706-8%5C_22.

[9] Roman Andriushchenko, Ezio Bartocci, Milan Ceska, Francesco Pontiggia, and Sarah Sallinger. "Deductive Controller Synthesis for Probabilistic Hyperproperties". In: *Quantitative Evaluation of Systems - 20th International Conference, QEST 2023, Antwerp, Belgium, September 20-22, 2023, Proceedings*. Ed. by Nils Jansen and Mirco Tribastone. Vol. 14287. Lecture Notes in Computer Science. Springer, 2023, pp. 288–306. DOI: 10.1007/978-3-031-43835-6_20. URL: https://doi.org/10.1007/978-3-031-43835-6%5C_20.

[10] Anagha Athavale, Ezio Bartocci, Maria Christakis, Matteo Maffei, Dejan Nickovic, and Georg Weissenbacher. "Verifying Global Two-Safety Properties in Neural Networks with Confidence". In: *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part II*. Ed. by Arie Gurfinkel and Vijay Ganesh. Vol. 14682. Lecture Notes in Computer Science. Springer, 2024, pp. 329–351. DOI: 10.1007/978-3-031-65630-9_17. URL: https://doi.org/10.1007/978-3-031-65630-9%5C_17.

[11] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008. ISBN: 978-0-262-02649-9.

[12] Jorge A. Baier and Sheila A. McIlraith. "Knowledge-based programs as building blocks for planning". In: *Artif. Intell.* 303 (2022), p. 103634. DOI: 10.1016/J.ARTINT.2021.103634. URL: https://doi.org/10.1016/j.artint.2021.103634.

[13] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. "SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft". In: *Integrated Formal Methods, 4th International Conference, IFM 2004, Canterbury, UK, April 4-7, 2004, Proceedings*. Ed. by Eerke A. Boiten, John Derrick, and Graeme Smith. Vol. 2999. Lecture Notes in Computer Science. Springer, 2004, pp. 1–20. DOI: 10.1007/978-3-540-24756-2_1. URL: https://doi.org/10.1007/978-3-540-24756-2%5C_1.

[14] Musard Balliu, Mads Dam, and Gurvan Le Guernic. "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*. Ed. by Aslan Askarov and Joshua D. Guttman. ACM, 2011, p. 6. DOI: 10.1145/2166956.2166962. URL: https://doi.org/10.1145/2166956.2166962.

[15] Suguman Bansal, Yong Li, Lucas M. Tabajara, Moshe Y. Vardi, and Andrew M. Wells. "Model Checking Strategies from Synthesis over Finite Traces". In: *International Symposium on Automated Technology for Verification and Analysis, ATVA 2023*. 2023. DOI: 10.1007/978-3-031-45329-8_11. URL: https://doi.org/10.1007/978-3-031-45329-8%5C_11.

[16] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. "Satisfiability Modulo Theories". In: *Handbook of Satisfiability - Second Edition*. Ed. by Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh. Vol. 336. Frontiers in Artificial Intelligence and Applications. IOS Press, 2021, pp. 1267–1329. DOI: 10.3233/FAIA201017. URL: https://doi.org/10.3233/FAIA201017.

[17] Howard Barringer, David E. Rydeheard, and Klaus Havelund. "Rule Systems for Run-time Monitoring: from Eagle to RuleR". In: *J. Log. Comput.* 20.3 (2010), pp. 675–706. DOI: 10.1093/LOGCOM/EXN076. URL: https://doi.org/10.1093/logcom/exn076.

[18] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. "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, 2004, pp. 100–114. DOI: 10.1109/CSFW.2004.17. URL: https://doi.ieeecomputersociety.org/10.1109/CSFW.2004.17.

[19] Keith A. Bartlett, Roger A. Scantlebury, and Peter T. Wilkinson. "A note on reliable full-duplex transmission over half-duplex links". In: *Commun. ACM* 12.5 (1969), pp. 260–261. DOI: 10.1145/362946.362970. URL: https://doi.org/10.1145/362946.362970.

[20] Nicolas Basset, Jean-François Raskin, and Ocan Sankur. "Admissible Strategies in Timed Games". In: *Models, Algorithms, Logics and Tools - Essays Dedicated to Kim Guldstrand Larsen on the Occasion of His 60th Birthday*. Ed. by Luca Aceto, Giorgio Bacci, Giovanni Bacci, Anna Ingólfsdóttir, Axel Legay, and Radu Mardare. Vol. 10460. Lecture Notes in Computer Science. Springer, 2017, pp. 403–

425. DOI: 10.1007/978-3-319-63121-9_20. URL: https://doi.org/10.1007/978-3-319-63121-9%5C_20.

[21] Andreas Bauer, Martin Leucker, and Christian Schallhart. "Runtime Verification for LTL and TLTL". In: *ACM Trans. Softw. Eng. Methodol.* (2011). DOI: 10.1145/2000799.2000800. URL: https://doi.org/10.1145/2000799.2000800.

[22] Andreas Bauer, Martin Leucker, and Christian Schallhart. "Runtime Verification for LTL and TLTL". In: *ACM Trans. Softw. Eng. Methodol.* 20.4 (2011), 14:1–14:64. DOI: 10.1145/2000799.2000800. URL: https://doi.org/10.1145/2000799.2000800.

[23] Jan Baumeister, Norine Coenen, Borzoo Bonakdarpour, Bernd Finkbeiner, and César Sánchez. "A Temporal Logic for Asynchronous Hyperproperties". In: *International Conference on Computer Aided Verification, CAV 2021*. Vol. 12759. Lecture Notes in Computer Science. Springer, 2021. DOI: 10.1007/978-3-030-81685-8_33.

[24] Jan Eric Baumeister. "Encodings of Bounded Synthesis of Distributed Systems". B.Sc. Thesis, Saarland University. 2017.

[25] Bernhard Beckert and Mattias Ulbrich. "Trends in Relational Program Verification". In: *Principled Software Development - Essays Dedicated to Arnd Poetzsch-Heffter on the Occasion of his 60th Birthday*. Ed. by Peter Müller and Ina Schaefer. Springer, 2018, pp. 41–58. DOI: 10.1007/978-3-319-98047-8_3. URL: https://doi.org/10.1007/978-3-319-98047-8%5C_3.

[26] Calin Belta, Boyan Yordanov, and Ebru Aydin Gol. *Formal methods for discrete-time dynamical systems*. Vol. 89. Springer, 2017.

[27] Julien Bernet, David Janin, and Igor Walukiewicz. "Permissive strategies: from parity games to safety games". In: *RAIRO Theor. Informatics Appl.* 36.3 (2002), pp. 261–275. DOI: 10.1051/ita:2002013.

[28] Dietmar Berwanger. "Admissibility in Infinite Games". In: *STACS 2007, 24th Annual Symposium on Theoretical Aspects of Computer Science, Aachen, Germany, February 22-24, 2007, Proceedings*. Ed. by Wolfgang Thomas and Pascal Weil. Vol. 4393. Lecture Notes in Computer Science. Springer, 2007, pp. 188–199. DOI: 10.1007/978-3-540-70918-3_17. URL: https://doi.org/10.1007/978-3-540-70918-3%5C_17.

[29] Raven Beutner and Bernd Finkbeiner. "A Temporal Logic for Strategic Hyperproperties". In: *International Conference on Concurrency Theory, CONCUR 2021*. Vol. 203. LIPIcs. Schloss Dagstuhl, 2021. DOI: 10.4230/LIPIcs.CONCUR.2021.24.

[30] Raven Beutner and Bernd Finkbeiner. "HyperATL*: A Logic for Hyperproperties in Multi-Agent Systems". In: *Log. Methods Comput. Sci.* (2023).

[31] Raven Beutner and Bernd Finkbeiner. "Model Checking Omega-Regular Hyperproperties with AutoHyperQ". In: *LPAR 2023: Proceedings of 24th International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Manizales, Colombia, 4-9th June 2023*. Ed. by Ruzica Piskac and Andrei Voronkov. Vol. 94. EPiC Series in Computing. EasyChair, 2023, pp. 23–35. DOI: 10.29007/1XJT. URL: https://doi.org/10.29007/1xjt.

[32] Raven Beutner and Bernd Finkbeiner. "Prophecy Variables for Hyperproperty Verification". In: *35th IEEE Computer Security Foundations Symposium, CSF 2022, Haifa, Israel, August 7-10, 2022*. IEEE, 2022, pp. 471–485. DOI: 10.1109/CSF54842.2022.9919658. URL: https://doi.org/10.1109/CSF54842.2022.9919658.

[33] Raven Beutner and Bernd Finkbeiner. "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*. Ed. by Sharon Shoham and Yakir Vizel. Vol. 13371. Lecture Notes in Computer Science. Springer, 2022, pp. 341–362. DOI: 10.1007/978-3-031-13185-1_17. URL: https://doi.org/10.1007/978-3-031-13185-1%5C_17.

[34] Raven Beutner, Bernd Finkbeiner, Hadar Frenkel, and Niklas Metzger. "Monitoring Second-Order Hyperproperties". In: *Proceedings of the 23rd International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2024, Auckland, New Zealand, May 6-10, 2024*. Ed. by Mehdi Dastani, Jaime Simão Sichman, Natasha Alechina, and Virginia Dignum. International Foundation for Autonomous Agents and Multiagent Systems / ACM, 2024, pp. 180–188. DOI: 10.5555/3635637.3662865. URL: https://dl.acm.org/doi/10.5555/3635637.3662865.

[35] Raven Beutner, Bernd Finkbeiner, Hadar Frenkel, and Niklas Metzger. "Second-Order Hyperproperties". In: *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part II*. Ed. by Constantin Enea and Akash Lal. Vol. 13965. Lecture Notes in Computer Science. Springer, 2023, pp. 309–332. DOI: 10.1007/978-3-031-37703-7_15. URL: https://doi.org/10.1007/978-3-031-37703-7%5C_15.

[36] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, eds. *Handbook of Satisfiability*. Vol. 185. Frontiers in Artificial Intelligence and Applications. IOS Press, 2009. ISBN: 978-1-58603-929-5.

[37] Roderick Bloem, Krishnendu Chatterjee, Swen Jacobs, and Robert Könighofer. "Assume-Guarantee Synthesis for Concurrent Reactive Programs with Partial Information". In: *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*. Ed. by Christel Baier and Cesare Tinelli. Vol. 9035. Lecture Notes in Computer Science. Springer, 2015, pp. 517–532. DOI: 10.1007/978-3-662-46681-0_50. URL: https://doi.org/10.1007/978-3-662-46681-0%5C_50.

[38] Roderick Bloem, Krishnendu Chatterjee, and Barbara Jobstmann. "Graph Games and Reactive Synthesis". In: *Handbook of Model Checking*. Ed. by Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem. Springer, 2018, pp. 921–962. DOI: 10.1007/978-3-319-10575-8_27. URL: https://doi.org/10.1007/978-3-319-10575-8%5C_27.

[39] Microsoft Research Blog. *TLA+ Foundation aims to bring math-based software modeling to the mainstream*. 2023. URL: https://www.microsoft.com/en-us/research/blog/tla-foundation-aims-to-bring-math-based-software-modeling-to-the-mainstream/ (visited on 06/05/2025).

[40] Borzoo Bonakdarpour and Bernd Finkbeiner. "Controller Synthesis for Hyperproperties". In: *33rd IEEE Computer Security Foundations Symposium, CSF 2020, Boston, MA, USA, June 22-26, 2020*. IEEE, 2020, pp. 366–379. DOI: 10.1109/CSF49147.2020.00033. URL: https://doi.org/10.1109/CSF49147.2020.00033.

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

[42] Borzoo Bonakdarpour, César Sánchez, and Gerardo Schneider. "Monitoring Hyperproperties by Combining Static Analysis and Runtime Verification". In: *Leveraging Applications of Formal Methods, Verification and Validation. Verification - 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part II*. Ed. by Tiziana Margaria and Bernhard Steffen. Vol. 11245. Lecture Notes in Computer Science. Springer, 2018, pp. 8–27. DOI: 10.1007/978-3-030-03421-4_2. URL: https://doi.org/10.1007/978-3-030-03421-4%5C_2.

[43] Borzoo Bonakdarpour and Sarai Sheinvald. "Finite-Word Hyperlanguages". In: *Language and Automata Theory and Applications - 15th International Conference, LATA 2021, Milan, Italy, March 1-5, 2021, Proceedings*. Ed. by Alberto Leporati, Carlos Martín-Vide, Dana Shapira, and Claudio Zandron. Vol. 12638. Lecture Notes in Computer Science. Springer, 2021, pp. 173–186. DOI: 10.1007/978-3-030-68195-1_17. URL: https://doi.org/10.1007/978-3-030-68195-1%5C_17.

[44] Patricia Bouyer, Nicolas Markey, Jörg Olschewski, and Michael Ummels. "Measuring Permissiveness in Parity Games: Mean-Payoff Parity Games Revisited". In: *Automated Technology for Verification and Analysis, 9th International Symposium, ATVA 2011, Taipei, Taiwan, October 11-14, 2011. Proceedings*. Ed. by Tevfik Bultan and Pao-Ann Hsiung. Vol. 6996. Lecture Notes in Computer Science. Springer, 2011, pp. 135–149. DOI: 10.1007/978-3-642-24372-1_11.

[45] Rodica Bozianu, Catalin Dima, and Emmanuel Filiot. "Safraless Synthesis for Epistemic Temporal Specifications". 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. Ed. by Armin Biere and Roderick Bloem. Vol. 8559. Lecture Notes in Computer Science. Springer, 2014, pp. 441–456. DOI: 10.1007/978-3-319-08867-9_29. URL: https://doi.org/10.1007/978-3-319-08867-9%5C_29.

[46] Laura Bozzelli, Bastien Maubert, and Sophie Pinchinat. "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*. Ed. by Andrew M. Pitts. Vol. 9034. Lecture Notes in Computer Science. Springer, 2015, pp. 167–182. DOI: 10.1007/978-3-662-46678-0_11. URL: https://doi.org/10.1007/978-3-662-46678-0%5C_11.

[47] Laura Bozzelli, Adriano Peron, and César Sánchez. "Asynchronous Extensions of HyperLTL". In: *Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021*. IEEE, 2021. DOI: 10.1109/LICS52264.2021.9470583.

[48] Laura Bozzelli, Adriano Peron, and César Sánchez. "Expressiveness and Decidability of Temporal Logics for Asynchronous Hyperproperties". In: *33rd International Conference on Concurrency Theory, CONCUR 2022, September 12-16, 2022, Warsaw, Poland*. Ed. by Bartek Klin, Slawomir Lasota, and Anca Muscholl. Vol. 243. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, 27:1–27:16. DOI: 10.4230/LIPICS.CONCUR.2022.27. URL: https://doi.org/10.4230/LIPIcs.CONCUR.2022.27.

[49] Romain Brenguier, Jean-François Raskin, and Ocan Sankur. "Assume-admissible synthesis". In: *Acta Informatica* 54.1 (2017), pp. 41–83. DOI: 10.1007/S00236-016-0273-2. URL: https://doi.org/10.1007/s00236-016-0273-2.

[50] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. "Symbolic Model Checking: 10^{20} States and Beyond". In: *Proceedings of the Fifth Annual Symposium on Logic in Computer Science (LICS '90), Philadelphia, Pennsylvania, USA, June 4-7, 1990*. IEEE Computer Society, 1990, pp. 428–439. DOI: 10.1109/LICS.1990.113767. URL: https://doi.org/10.1109/LICS.1990.113767.

[51] Alberto Camacho, Jorge A. Baier, Christian J. Muise, and Sheila A. McIlraith. "Finite LTL Synthesis as Planning". In: *International Conference on Automated Planning and Scheduling, ICAPS 2018*. 2018.

[52] Christos G Cassandras and Stéphane Lafortune. *Introduction to Discrete Event Systems*. Springer Nature, 2021.

[53] Marek Chalupa and Thomas A. Henzinger. "Monitoring Hyperproperties with Prefix Transducers". In: *Runtime Verification - 23rd International Conference, RV 2023, Thessaloniki, Greece, October 3-6, 2023, Proceedings*. Ed. by Panagiotis Katsaros and Laura Nenzi. Vol. 14245. Lecture Notes in Computer Science. Springer, 2023, pp. 168–190. DOI: 10.1007/978-3-031-44267-4_9. URL: https://doi.org/10.1007/978-3-031-44267-4%5C_9.

[54] Krishnendu Chatterjee and Thomas A. Henzinger. "Assume-Guarantee Synthesis". In: *Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24 - April 1, 2007, Proceedings*. Ed. by Orna Grumberg and Michael Huth. Vol. 4424. Lecture Notes in Computer Science. Springer, 2007, pp. 261–275. DOI: 10.1007/978-3-540-71209-1_21. URL: https://doi.org/10.1007/978-3-540-71209-1%5C_21.

[55] A. Church. "Applications of recursive arithmetic to the problem of circuit synthesis". In: *Summaries of the Summer Institute of Symbolic Logic*. Vol. 1. Cornell Univ., Ithaca, NY, 1957, pp. 3–50.

[56] Alonzo Church. "Logic, Arithmetic, and Automata". In: *Journal of Symbolic Logic* 29.4 (1964), pp. 210–210. DOI: 10.2307/2270398.

[57] Edmund M. Clarke. "The Birth of Model Checking". In: *25 Years of Model Checking - History, Achievements, Perspectives*. Ed. by Orna Grumberg and Helmut Veith. Vol. 5000. Lecture Notes in Computer Science. Springer, 2008, pp. 1–26. DOI: 10.1007/978-3-540-69850-0_1. URL: https://doi.org/10.1007/978-3-540-69850-0%5C_1.

[58] Edmund M. Clarke, Orna Grumberg, Daniel Kroening, Doron A. Peled, and Helmut Veith. *Model checking, 2nd Edition*. MIT Press, 2018. ISBN: 978-0-262-03883-6. URL: https://mitpress.mit.edu/books/model-checking-second-edition.

[59] Michael R. Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K. Micinski, Markus N. Rabe, and César Sánchez. "Temporal Logics for Hyperproperties". In: *Principles of Security and Trust - Third International Conference, POST 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*. Ed. by Martín Abadi and Steve Kremer. Vol. 8414. Lecture Notes in Computer Science. Springer, 2014, pp. 265–284. DOI: 10.1007/978-3-642-54792-8_15. URL: https://doi.org/10.1007/978-3-642-54792-8%5C_15.

[60] Michael R. Clarkson and Fred B. Schneider. "Hyperproperties". In: *Journal of Computer Security* 18.6 (2010), pp. 1157–1210.

[61] Norine Coenen, Raimund Dachselt, Bernd Finkbeiner, Hadar Frenkel, Christopher Hahn, Tom Horak, Niklas Metzger, and Julian Siber. "Explaining Hyperproperty Violations". In: *International Conference on Computer Aided Verification, CAV 2022*. Vol. 13371. Lecture Notes in Computer Science. Springer, 2022. DOI: 10.1007/978-3-031-13185-1_20.

[62] Norine Coenen, Bernd Finkbeiner, Hadar Frenkel, Christopher Hahn, Niklas Metzger, and Julian Siber. "Temporal Causality in Reactive Systems". In: *International Symposium on Automated Technology for Verification and Analysis, ATVA 2022*. Vol. 13505. Lecture Notes in Computer Science. Springer, 2022. DOI: 10.1007/978-3-031-19992-9_13.

[63] Norine Coenen, Bernd Finkbeiner, Christopher Hahn, and Jana Hofmann. "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, 2019, pp. 1–13. DOI: 10.1109/LICS.2019.8785713. URL: https://doi.org/10.1109/LICS.2019.8785713.

[64] Norine Coenen, Bernd Finkbeiner, Christopher Hahn, Jana Hofmann, and Yannick Schillo. "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*. Ed. by Zhe Hou and Vijay Ganesh. Vol. 12971. Lecture Notes in Computer Science. Springer, 2021, pp. 283–299. DOI: 10.1007/978-3-030-88885-5_19. URL: https://doi.org/10.1007/978-3-030-88885-5%5C_19.

[65] Norine Coenen, Bernd Finkbeiner, Jana Hofmann, and Julia J. Tillman. "Smart Contract Synthesis Modulo Hyperproperties". In: *36th IEEE Computer Security Foundations Symposium, CSF 2023, Dubrovnik, Croatia, July 10-14, 2023*. IEEE, 2023, pp. 276–291. DOI: 10.1109/CSF57540.2023.00006. URL: https://doi.org/10.1109/CSF57540.2023.00006.

[66] Norine Coenen, Bernd Finkbeiner, César Sánchez, and Leander Tentrup. "Verifying Hyperliveness". In: *International Conference on Computer Aided Verification, CAV 2019*. Vol. 11561. Lecture Notes in Computer Science. Springer, 2019. DOI: 10.1007/978-3-030-25540-4_7.

[67] Marcelo d'Amorim and Grigore Rosu. "Efficient Monitoring of omega-Languages". In: *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*. Ed. by Kousha Etessami and Sriram K. Rajamani. Vol. 3576. Lecture Notes in Computer Science. Springer, 2005, pp. 364–378. DOI: 10.1007/11513988_36. URL: https://doi.org/10.1007/11513988%5C_36.

[68] Werner Damm and Bernd Finkbeiner. "Automatic Compositional Synthesis of Distributed Systems". In: *FM 2014: Formal Methods - 19th International Symposium, Singapore, May 12-16, 2014. Proceedings*. Ed. by Cliff B. Jones, Pekka Pihlajasaari, and Jun Sun. Vol. 8442. Lecture Notes in Computer Science. Springer, 2014, pp. 179–193. DOI: 10.1007/978-3-319-06410-9_13.

[69] Werner Damm and Bernd Finkbeiner. "Automatic Compositional Synthesis of Distributed Systems". In: *FM 2014: Formal Methods - 19th International Symposium, Singapore, May 12-16, 2014. Proceedings*. Ed. by Cliff B. Jones, Pekka Pihlajasaari, and Jun Sun. Vol. 8442. Lecture Notes in Computer Science. Springer, 2014, pp. 179–193. DOI: 10.1007/978-3-319-06410-9_13. URL: https://doi.org/10.1007/978-3-319-06410-9%5C_13.

[70] Ádám Darvas, Reiner Hähnle, and David Sands. "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. Ed. by Dieter Hutter and Markus Ullmann. Vol. 3450. Lecture Notes in Computer Science. Springer, 2005, pp. 193–209. DOI: 10.1007/978-3-540-32004-3_20. URL: https://doi.org/10.1007/978-3-540-32004-3%5C_20.

[71] Ankush Desai, Amar Phanishayee, Shaz Qadeer, and Sanjit A. Seshia. "Compositional programming and testing of dynamic distributed systems". In: *Proc. ACM Program. Lang.* 2.OOPSLA (2018), 159:1–159:30. DOI: 10.1145/3276529. URL: https://doi.org/10.1145/3276529.

[72] Tom van Dijk. "Oink: An Implementation and Evaluation of Modern Parity Game Solvers". In: *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part I*. Ed. by Dirk Beyer and Marieke Huisman. Vol. 10805. Lecture Notes in Computer Science. Springer, 2018, pp. 291–308. DOI: 10.1007/978-3-319-89960-2_16. URL: https://doi.org/10.1007/978-3-319-89960-2%5C_16.

[73] Rayna Dimitrova, Bernd Finkbeiner, Máté Kovács, Markus N. Rabe, and Helmut Seidl. "Model Checking Information Flow in Reactive Systems". In: *Verification, Model Checking, and Abstract Interpretation - 13th International Conference, VMCAI 2012, Philadelphia, PA, USA, January 22-24, 2012. Proceedings*. Ed. by Viktor Kuncak and Andrey Rybalchenko. Vol. 7148. Lecture Notes in Computer Science. Springer, 2012, pp. 169–185. DOI: 10.1007/978-3-642-27940-9_12. URL: https://doi.org/10.1007/978-3-642-27940-9%5C_12.

[74] Rayna Dimitrova, Bernd Finkbeiner, and Hazem Torfah. "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*. Ed. by Dang Van Hung and Oleg Sokolsky. Vol. 12302. Lecture Notes in Computer Science. Springer, 2020, pp. 484–500. DOI: 10.1007/978-3-030-59152-6_27. URL: https://doi.org/10.1007/978-3-030-59152-6%5C_27.

[75] Ali Dorri, Salil S. Kanhere, and Raja Jurdak. "Multi-Agent Systems: A Survey". In: *IEEE Access* 6 (2018), pp. 28573–28593. DOI: 10.1109/ACCESS.2018.2831228. URL: https://doi.org/10.1109/ACCESS.2018.2831228.

[76] Alexandre Duret-Lutz, Etienne Renault, Maximilien Colange, Florian Renkin, Alexandre Gbaguidi Aisse, Philipp Schlehuber-Caissier, Thomas Medioni, Antoine Martin, Jérôme Dubois, Clément Gillard, and Henrich Lauko. "From Spot 2.0 to Spot 2.10: What's New?" In: *International Conference on Computer Aided Verification, CAV 2022*. Vol. 13372. Lecture Notes in Computer Science. Springer, 2022. DOI: 10.1007/978-3-031-13188-2_9.

[77] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam D. Smith. "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. Ed. by Shai Halevi and Tal Rabin. Vol. 3876. Lecture Notes in Computer Science. Springer, 2006, pp. 265–284. DOI: 10.1007/11681878_14. URL: https://doi.org/10.1007/11681878%5C_14.

[78] Rüdiger Ehlers, Stéphane Lafortune, Stavros Tripakis, and Moshe Y. Vardi. "Supervisory control and reactive synthesis: a comparative introduction". In: *Discret. Event Dyn. Syst.* 27.2 (2017), pp. 209–260. DOI: 10.1007/S10626-015-0223-0. URL: https://doi.org/10.1007/s10626-015-0223-0.

[79] Ronald Fagin, Joseph Y. Halpern, Yoram Moses, and Moshe Y. Vardi. "Knowledge-Based Programs". In: *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing, Ottawa, Ontario, Canada, August 20-23, 1995*. Ed. by James H. Anderson. ACM, 1995, pp. 153–163. DOI: 10.1145/224964.224982. URL: https://doi.org/10.1145/224964.224982.

[80] Ronald Fagin, Joseph Y. Halpern, Yoram Moses, and Moshe Y. Vardi. "Knowledge-Based Programs". In: *Distributed Comput.* 10.4 (1997), pp. 199–225. DOI: 10.1007/S004460050038. URL: https://doi.org/10.1007/s004460050038.

[81] Ronald Fagin, Joseph Y. Halpern, Yoram Moses, and Moshe Y. Vardi. *Reasoning About Knowledge*. MIT Press, 1995. DOI: 10.7551/mitpress/5803.001.0001.

[82] Azadeh Farzan and Anthony Vandikas. "Automated Hypersafety Verification". In: *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*. Ed. by Isil Dillig and Serdar Tasiran. Vol. 11561. Lecture Notes in Computer Science. Springer, 2019, pp. 200–218. DOI: 10.1007/978-3-030-25540-4_11. URL: https://doi.org/10.1007/978-3-030-25540-4%5C_11.

[83] Peter Faymonville, Bernd Finkbeiner, Markus N. Rabe, and Leander Tentrup. "Encodings of Bounded Synthesis". In: *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I*. Ed. by Axel Legay and Tiziana Margaria. Vol. 10205. Lecture Notes in Computer Science. 2017, pp. 354–370. DOI: 10.1007/978-3-662-54577-5_20. URL: https://doi.org/10.1007/978-3-662-54577-5%5C_20.

[84] Peter Faymonville, Bernd Finkbeiner, and Leander Tentrup. "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*. Ed. by Rupak Majumdar and Viktor Kuncak. Vol. 10427. Lecture Notes in Computer Science. Springer, 2017, pp. 325–332. DOI: 10.1007/978-3-319-63390-9_17. URL: https://doi.org/10.1007/978-3-319-63390-9%5C_17.

[85] Uriel Feige, Amos Fiat, and Adi Shamir. "Zero-Knowledge Proofs of Identity". In: *J. Cryptol.* 1.2 (1988), pp. 77–94. DOI: 10.1007/BF02351717. URL: https://doi.org/10.1007/BF02351717.

[86] Paolo Felli, Marco Montali, Fabio Patrizi, and Sarah Winkler. "Monitoring Arithmetic Temporal Properties on Finite Traces". In: *Conference on Artificial Intelligence, AAAI 2023*. 2023. DOI: 10.1609/aaai.v37i5.25781.

[87] Emmanuel Filiot, Naiyong Jin, and Jean-François Raskin. "Compositional Algorithms for LTL Synthesis". In: *Automated Technology for Verification and Analysis - 8th International Symposium, ATVA 2010, Singapore, September 21-24, 2010. Proceedings*. Ed. by Ahmed Bouajjani and Wei-Ngan Chin. Vol. 6252. Lecture Notes in Computer Science. Springer, 2010, pp. 112–127. DOI: 10.1007/978-3-642-15643-4_10.

[88] Emmanuel Filiot, Naiyong Jin, and Jean-François Raskin. "Compositional Algorithms for LTL Synthesis". In: *Automated Technology for Verification and Analysis - 8th International Symposium, ATVA 2010, Singapore, September 21-24, 2010. Proceedings*. Ed. by Ahmed Bouajjani and Wei-Ngan Chin. Vol. 6252. Lecture Notes in Computer Science. Springer, 2010, pp. 112–127. DOI: 10.1007/978-3-642-15643-4_10. URL: https://doi.org/10.1007/978-3-642-15643-4%5C_10.

[89] B. Finkbeiner and S. Schewe. "Uniform Distributed Synthesis". In: *Proceedings of the 20th ACM/IEEE Symposium on Logic in Computer Science (LICS)*. 2005, pp. 321–330.

[90] Bernd Finkbeiner. "Bounded Synthesis for Petri Games". In: *Correct System Design - Symposium in Honor of Ernst-Rüdiger Olderog on the Occasion of His 60th Birthday, Oldenburg, Germany, September 8-9, 2015. Proceedings*. Ed. by Roland Meyer, André Platzer, and Heike Wehrheim. Vol. 9360. Lecture Notes in Computer Science. Springer, 2015, pp. 223–237. DOI: 10.1007/978-3-319-23506-6_15. URL: https://doi.org/10.1007/978-3-319-23506-6%5C_15.

[91] Bernd Finkbeiner. "Synthesis of Reactive Systems". In: *Dependable Software Systems Engineering*. Ed. by Javier Esparza, Orna Grumberg, and Salomon Sickert. Vol. 45. NATO Science for Peace and Security Series - D: Information and Communication Security. IOS Press, 2016, pp. 72–98. DOI: 10.3233/978-1-61499-627-9-72. URL: https://doi.org/10.3233/978-1-61499-627-9-72.

[92] Bernd Finkbeiner, Hadar Frenkel, Jana Hofmann, and Janine Lohse. "Automata-Based Software Model Checking of Hyperproperties". In: *NASA Formal Methods - 15th International Symposium, NFM 2023, Houston, TX, USA, May 16-18, 2023, Proceedings*. Ed. by Kristin Yvonne Rozier and Swarat Chaudhuri. Vol. 13903. Lecture Notes in Computer Science. Springer, 2023, pp. 361–379. DOI: 10.1007/978-3-031-33170-1_22. URL: https://doi.org/10.1007/978-3-031-33170-1%5C_22.

[93] Bernd Finkbeiner, Christopher Hahn, Philip Lukert, Marvin Stenger, and Leander Tentrup. "Synthesis from hyperproperties". In: *Acta Informatica* 57.1-2 (2020), pp. 137–163. DOI: 10.1007/S00236-019-00358-2. URL: https://doi.org/10.1007/s00236-019-00358-2.

[94] Bernd Finkbeiner, Christopher Hahn, Philip Lukert, Marvin Stenger, and Leander Tentrup. "Synthesizing Reactive Systems from Hyperproperties". In: *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*. Ed. by Hana Chockler and Georg Weissenbacher. Vol. 10981. Lecture Notes in Computer Science. Springer, 2018, pp. 289–306. DOI: 10.1007/978-3-319-96145-3_16. URL: https://doi.org/10.1007/978-3-319-96145-3%5C_16.

[95] Bernd Finkbeiner, Christopher Hahn, Marvin Stenger, and Leander Tentrup. "Monitoring hyperproperties". In: *Formal Methods Syst. Des.* (2019). DOI: 10.1007/s10703-019-00334-z. URL: https://doi.org/10.1007/s10703-019-00334-z.

[96] Bernd Finkbeiner, Christopher Hahn, Marvin Stenger, and Leander Tentrup. "RVHyper: A Runtime Verification Tool for Temporal Hyperproperties". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2018*. 2018. DOI: 10.1007/978-3-319-89963-3_11. URL: https://doi.org/10.1007/978-3-319-89963-3%5C_11.

[97] Bernd Finkbeiner, Niklas Metzger, and Yoram Moses. "Information Flow Guided Synthesis". In: *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part II*. Ed. by Sharon Shoham and Yakir Vizel. Vol. 13372. Lecture Notes in Computer Science. Springer, 2022, pp. 505–525. DOI: 10.1007/978-3-031-13188-2_25. URL: https://doi.org/10.1007/978-3-031-13188-2%5C_25.

[98] Bernd Finkbeiner, Niklas Metzger, and Yoram Moses. "Information Flow Guided Synthesis with Unbounded Communication". In: *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part III*. Ed. by Arie Gurfinkel and Vijay Ganesh. Vol. 14683. Lecture Notes in Computer Science. Springer, 2024, pp. 64–86. DOI: 10.1007/978-3-031-65633-0_4. URL: https://doi.org/10.1007/978-3-031-65633-0%5C_4.

[99] Bernd Finkbeiner, Niklas Metzger, Satya Prakash Nayak, and Anne-Kathrin Schmuck. "Synthesis of Universal Safety Controllers". In: *Tools and Algorithms for the Construction and Analysis of Systems - 31st International Conference, TACAS 2025, Held as Part of the International Joint Conferences on Theory and Practice of Software, ETAPS 2025, Hamilton, ON, Canada, May 3-8, 2025, Proceedings, Part II*. Ed. by Arie Gurfinkel and Marijn Heule. Vol. 15697. Lecture Notes in Computer Science. Springer, 2025, pp. 177–197. DOI: 10.1007/978-3-031-90653-4_9. URL: https://doi.org/10.1007/978-3-031-90653-4%5C_9.

[100] Bernd Finkbeiner and Ernst-Rüdiger Olderog. "Petri games: Synthesis of distributed systems with causal memory". In: *Inf. Comput.* 253 (2017), pp. 181–203. DOI: 10.1016/J.IC.2016.07.006. URL: https://doi.org/10.1016/j.ic.2016.07.006.

[101] Bernd Finkbeiner and Noemi Passing. "Compositional Synthesis of Modular Systems". In: *Automated Technology for Verification and Analysis - 19th International Symposium, ATVA 2021, Gold Coast, QLD, Australia, October 18-22, 2021, Proceed-*

ings. Ed. by Zhe Hou and Vijay Ganesh. Vol. 12971. Lecture Notes in Computer Science. Springer, 2021, pp. 303–319. DOI: 10.1007/978-3-030-88885-5_20. URL: https://doi.org/10.1007/978-3-030-88885-5%5C_20.

[102] Bernd Finkbeiner and Noemi Passing. "Dependency-Based Compositional Synthesis". In: *Automated Technology for Verification and Analysis - 18th International Symposium, ATVA 2020, Hanoi, Vietnam, October 19-23, 2020, Proceedings*. Ed. by Dang Van Hung and Oleg Sokolsky. Vol. 12302. Lecture Notes in Computer Science. Springer, 2020, pp. 447–463. DOI: 10.1007/978-3-030-59152-6_25.

[103] Bernd Finkbeiner and Noemi Passing. "Dependency-Based Compositional Synthesis". In: *Automated Technology for Verification and Analysis - 18th International Symposium, ATVA 2020, Hanoi, Vietnam, October 19-23, 2020, Proceedings*. Ed. by Dang Van Hung and Oleg Sokolsky. Vol. 12302. Lecture Notes in Computer Science. Springer, 2020, pp. 447–463. DOI: 10.1007/978-3-030-59152-6_25. URL: https://doi.org/10.1007/978-3-030-59152-6%5C_25.

[104] Bernd Finkbeiner and Noemi Passing. "Synthesizing Dominant Strategies for Liveness". In: *42nd IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2022, December 18-20, 2022, IIT Madras, Chennai, India*. Ed. by Anuj Dawar and Venkatesan Guruswami. Vol. 250. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, 37:1–37:19. DOI: 10.4230/LIPICS.FSTTCS.2022.37. URL: https://doi.org/10.4230/LIPIcs.FSTTCS.2022.37.

[105] Bernd Finkbeiner, Markus N. Rabe, and César Sánchez. "Algorithms for Model Checking HyperLTL and HyperCTL $^{\hat{*}}$". In: *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*. Ed. by Daniel Kroening and Corina S. Pasareanu. Vol. 9206. Lecture Notes in Computer Science. Springer, 2015, pp. 30–48. DOI: 10.1007/978-3-319-21690-4_3. URL: https://doi.org/10.1007/978-3-319-21690-4%5C_3.

[106] Bernd Finkbeiner and Sven Schewe. "Uniform Distributed Synthesis". In: *20th IEEE Symposium on Logic in Computer Science (LICS 2005), 26-29 June 2005, Chicago, IL, USA, Proceedings*. IEEE Computer Society, 2005, pp. 321–330. DOI: 10.1109/LICS.2005.53. URL: https://doi.org/10.1109/LICS.2005.53.

[107] Bernd Finkbeiner, Anne-Kathrin Schmuck, Satya Prakash Nayak, and Niklas Metzger. *Artifact for Synthesis of Universal Safety Controllers at TACAS 2025*. Jan. 2025. DOI: 10.5281/zenodo.14627151. URL: https://doi.org/10.5281/zenodo.14627151.

[108] Bernd Finkbeiner and Henny Sipma. "Checking Finite Traces using Alternating Automata". In: *Workshop on Runtime Verification, RV 2001*. 2001. DOI: 10.1016/S1571-0661(04)00250-6. URL: https://doi.org/10.1016/S1571-0661(04)00250-6.

[109] Bernd Finkbeiner and Martin Zimmermann. "The First-Order Logic of Hyperproperties". In: *Symposium on Theoretical Aspects of Computer Science, STACS 2017*. Vol. 66. LIPIcs. Schloss Dagstuhl, 2017. DOI: 10.4230/LIPIcs.STACS.2017.30.

[110] Valeria Fionda and Gianluigi Greco. "The Complexity of LTL on Finite Traces: Hard and Easy Fragments". In: *Conference on Artificial Intelligence, AAAI 2016*. 2016. DOI: 10.1609/aaai.v30i1.10104.

[111] Limor Fix. "Fifteen Years of Formal Property Verification in Intel". In: *25 Years of Model Checking - History, Achievements, Perspectives*. Ed. by Orna Grumberg and Helmut Veith. Vol. 5000. Lecture Notes in Computer Science. Springer, 2008, pp. 139–144. DOI: 10.1007/978-3-540-69850-0_8. URL: https://doi.org/10.1007/978-3-540-69850-0%5C_8.

[112] Daniel J. Fremont and Sanjit A. Seshia. "Reactive Control Improvisation". 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*. Ed. by Hana Chockler and Georg Weissenbacher. Vol. 10981. Lecture Notes in Computer Science. Springer, 2018, pp. 307–326. DOI: 10.1007/978-3-319-96145-3_17. URL: https://doi.org/10.1007/978-3-319-96145-3%5C_17.

[113] Hadar Frenkel and Sarai Sheinvald. "Realizable and Context-Free Hyperlanguages". In: *Proceedings of the 13th International Symposium on Games, Automata, Logics and Formal Verification, GandALF 2022, Madrid, Spain, September 21-23, 2022*. Ed. by Pierre Ganty and Dario Della Monica. Vol. 370. EPTCS. 2022, pp. 114–130. DOI: 10.4204/EPTCS.370.8. URL: https://doi.org/10.4204/EPTCS.370.8.

[114] *FsOmegaLib*. https://github.com/ravenbeutner/FsOmegaLib. Accessed: 2025-05-27.

[115] Dov M. Gabbay, Amir Pnueli, Saharon Shelah, and Jonathan Stavi. "On the Temporal Analysis of Fairness". In: *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages, Las Vegas, Nevada, USA, January 1980*. Ed. by Paul W. Abrahams, Richard J. Lipton, and Stephen R. Bourne. ACM Press, 1980, pp. 163–173. DOI: 10.1145/567446.567462. URL: https://doi.org/10.1145/567446.567462.

[116] Peter Gammie and Ron van der Meyden. "MCK: Model Checking the Logic of Knowledge". In: *International Conference on Computer Aided Verification, CAV 2004*. 2004. DOI: 10.1007/978-3-540-27813-9_41.

[117] Paul Gastin, Nathalie Sznajder, and Marc Zeitoun. "Distributed synthesis for well-connected architectures". In: *Formal Methods in System Design* 34.3 (2009), pp. 215–237.

[118] Giuseppe De Giacomo, Paolo Felli, Marco Montali, and Giuseppe Perelli. "HyperLDLf: a Logic for Checking Properties of Finite Traces Process Logs". In: *International Joint Conference on Artificial Intelligence, IJCAI 2021*. 2021. DOI: 10.24963/IJCAI.2021/256.

[119] Giuseppe De Giacomo and Moshe Y. Vardi. "Linear Temporal Logic and Linear Dynamic Logic on Finite Traces". In: *International Joint Conference on Artificial Intelligence, IJCAI 2013*. 2013.

[120] Giuseppe De Giacomo and Moshe Y. Vardi. "Synthesis for LTL and LDL on Finite Traces". In: *International Joint Conference on Artificial Intelligence, IJCAI 2015*. 2015.

[121] Adwait Godbole, Yatin A. Manerkar, and Sanjit A. Seshia. "SemPat: From Hyperproperties to Attack Patterns for Scalable Analysis of Microarchitectural Security". In: *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security, CCS 2024, Salt Lake City, UT, USA, October 14-18, 2024*. Ed. by Bo Luo, Xiaojing Liao, Jun Xu, Engin Kirda, and David Lie. ACM, 2024, pp. 2756–2770. DOI: 10.1145/3658644.3690214. URL: https://doi.org/10.1145/3658644.3690214.

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

[123] Ohad Goudsmid, Orna Grumberg, and Sarai Sheinvald. "Compositional Model Checking for Multi-properties". In: *Verification, Model Checking, and Abstract Interpretation - 22nd International Conference, VMCAI 2021, Copenhagen, Denmark, January 17-19, 2021, Proceedings*. Ed. by Fritz Henglein, Sharon Shoham, and Yakir Vizel. Vol. 12597. Lecture Notes in Computer Science. Springer, 2021, pp. 55–80. DOI: 10.1007/978-3-030-67067-2_4. URL: https://doi.org/10.1007/978-3-030-67067-2%5C_4.

[124] Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. "Hardware-Software Contracts for Secure Speculation". In: *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 2021, pp. 1868–1883. DOI: 10.1109/SP40001.2021.00036. URL: https://doi.org/10.1109/SP40001.2021.00036.

[125] Sankar Gurumurthy, Orna Kupferman, Fabio Somenzi, and Moshe Y. Vardi. "On Complementing Nondeterministic Büchi Automata". In: *Correct Hardware Design and Verification Methods, 12th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2003, L'Aquila, Italy, October 21-24, 2003, Proceedings*. Ed. by Daniel Geist and Enrico Tronci. Vol. 2860. Lecture Notes in Computer Science. Springer, 2003, pp. 96–110. DOI: 10.1007/978-3-540-39724-3_10. URL: https://doi.org/10.1007/978-3-540-39724-3%5C_10.

[126] Julian Gutierrez, Giuseppe Perelli, and Michael J. Wooldridge. "Multi-player games with LDL goals over finite traces". In: *Inf. Comput.* (2021). DOI: 10.1016/J.IC.2020.104555.

[127] Jens Oliver Gutsfeld, Markus Müller-Olm, and Christoph Ohrem. "Automata and fixpoints for asynchronous hyperproperties". In: *Proc. ACM Program. Lang.* 5.POPL (2021). DOI: 10.1145/3434319.

[128] Jens Oliver Gutsfeld, Markus Müller-Olm, and Christoph Ohrem. "Propositional Dynamic Logic for Hyperproperties". In: *31st International Conference on Concurrency Theory, CONCUR 2020, September 1-4, 2020, Vienna, Austria (Virtual Conference)*. Ed. by Igor Konnov and Laura Kovács. Vol. 171. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 50:1–50:22. DOI: 10.4230/LIPICS.CONCUR.2020.50. URL: https://doi.org/10.4230/LIPIcs.CONCUR.2020.50.

[129] Joseph Y. Halpern, Ron van der Meyden, and Moshe Y. Vardi. "Complete Axiomatizations for Reasoning about Knowledge and Time". In: *SIAM J. Comput.* 33.3 (2004), pp. 674–703. DOI: 10.1137/S0097539797320906. URL: https://doi.org/10.1137/S0097539797320906.

[130] Joseph Y. Halpern and Yoram Moses. "Knowledge and Common Knowledge in a Distributed Environment". In: *J. ACM* 37.3 (1990), pp. 549–587. DOI: 10.1145/79147.79161. URL: https://doi.org/10.1145/79147.79161.

[131] Joseph Y. Halpern and Kevin R. O'Neill. "Secrecy in Multiagent Systems". In: *ACM Trans. Inf. Syst. Secur.* 12.1 (2008), 5:1–5:47. DOI: 10.1145/1410234.1410239. URL: https://doi.org/10.1145/1410234.1410239.

[132] David Harel and Amir Pnueli. "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*. Ed. by Krzysztof R. Apt. Vol. 13. NATO ASI Series. Springer, 1984, pp. 477–498. DOI: 10.1007/978-3-642-82453-1_17. URL: https://doi.org/10.1007/978-3-642-82453-1%5C_17.

[133] Klaus Havelund and Grigore Rosu. "Efficient monitoring of safety properties". In: *Int. J. Softw. Tools Technol. Transf.* (2004). DOI: 10.1007/s10009-003-0117-6.

[134] Jesko Hecking-Harbusch and Niklas O. Metzger. "Efficient Trace Encodings of Bounded Synthesis for Asynchronous Distributed Systems". In: *Automated Technology for Verification and Analysis - 17th International Symposium, ATVA 2019, Taipei, Taiwan, October 28-31, 2019, Proceedings*. Ed. by Yu-Fang Chen, Chih-Hong Cheng, and Javier Esparza. Vol. 11781. Lecture Notes in Computer Science. Springer, 2019, pp. 369–386. DOI: 10.1007/978-3-030-31784-3_22. URL: https://doi.org/10.1007/978-3-030-31784-3%5C_22.

[135] Kaarlo Jaakko Juhani Hintikka. *Knowledge and Belief: An Introduction to the Logic of the Two Notions*. Ithaca, NY, USA: Cornell University Press, 1962.

[136] Wiebe van der Hoek and Michael J. Wooldridge. "Model Checking Knowledge and Time". In: *International Workshop on Model Checking of Software, SPIN 2002*. Vol. 2318. Lecture Notes in Computer Science. Springer, 2002. DOI: 10.1007/3-540-46017-9_9.

[137] Xiaowei Huang and Marta Kwiatkowska. "Model Checking Probabilistic Knowledge: A PSPACE Case". In: *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA*. Ed. by Dale Schuurmans and Michael P. Wellman. AAAI Press, 2016, pp. 2516–2522. DOI: 10.1609/AAAI.V30I1.10122. URL: https://doi.org/10.1609/aaai.v30i1.10122.

[138] Xiaowei Huang, Cheng Luo, and Ron van der Meyden. "Symbolic model checking of probabilistic knowledge". In: *Proceedings of the 13th Conference on Theoretical Aspects of Rationality and Knowledge (TARK-2011), Groningen, The Netherlands, July 12-14, 2011*. Ed. by Krzysztof R. Apt. ACM, 2011, pp. 177–186. DOI: 10.1145/2000378.2000399. URL: https://doi.org/10.1145/2000378.2000399.

[139] Xiaowei Huang and Ron van der Meyden. "Symbolic Model Checking Epistemic Strategy Logic". In: *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada*. Ed. by Carla E. Brodley and Peter Stone. AAAI Press, 2014, pp. 1426–1432. DOI: 10.1609/AAAI.V28I1.8894. URL: https://doi.org/10.1609/aaai.v28i1.8894.

[140] Xiaowei Huang and Ron van der Meyden. "Symbolic Synthesis for Epistemic Specifications with Observational Semantics". In: *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*. Ed. by Erika Ábrahám and Klaus Havelund. Vol. 8413. Lecture Notes in Computer Science. Springer, 2014, pp. 455–469. DOI: 10.1007/978-3-642-54862-8_39. URL: https://doi.org/10.1007/978-3-642-54862-8%5C_39.

[141] Swen Jacobs, Guillermo A. Pérez, Remco Abraham, Véronique Bruyère, Michaël Cadilhac, Maximilien Colange, Charly Delfosse, Tom van Dijk, Alexandre Duret-Lutz, Peter Faymonville, Bernd Finkbeiner, Ayrat Khalimov, Felix Klein, Michael Luttenberger, Klara J. Meyer, Thibaud Michaud, Adrien Pommellet, Florian Renkin, Philipp Schlehuber-Caissier, Mouhammad Sakr, Salomon Sickert, Gaëtan Staquet, Clément Tamines, Leander Tentrup, and Adam Walker. "The Reactive Synthesis Competition (SYNTCOMP): 2018-2021". In: *CoRR* abs/2206.00251 (2022). DOI: 10.48550/ARXIV.2206.00251. arXiv: 2206.00251. URL: https://doi.org/10.48550/arXiv.2206.00251.

[142] Barbara Jobstmann and Roderick Bloem. "Optimizations for LTL Synthesis". In: *Formal Methods in Computer-Aided Design, 6th International Conference, FMCAD 2006, San Jose, California, USA, November 12-16, 2006, Proceedings*. IEEE Computer Society, 2006, pp. 117–124. DOI: 10.1109/FMCAD.2006.22. URL: https://doi.org/10.1109/FMCAD.2006.22.

[143] Barbara Jobstmann, Stefan J. Galler, Martin Weiglhofer, and Roderick Bloem. "Anzu: A Tool for Property Synthesis". In: *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*. Ed. by Werner Damm and Holger Hermanns. Vol. 4590. Lecture Notes in Computer Science. Springer, 2007, pp. 258–262. DOI: 10.1007/978-3-540-73368-3_29. URL: https://doi.org/10.1007/978-3-540-73368-3%5C_29.

[144] Daniel Kirsten. "Alternating Tree Automata and Parity Games". In: *Automata, Logics, and Infinite Games: A Guide to Current Research [outcome of a Dagstuhl seminar, February 2001]*. Ed. by Erich Grädel, Wolfgang Thomas, and Thomas Wilke. Vol. 2500. Lecture Notes in Computer Science. Springer, 2001, pp. 153–

167. DOI: 10.1007/3-540-36387-4_9. URL: https://doi.org/10.1007/3-540-36387-4%5C_9.

[145] Joachim Klein, Christel Baier, and Sascha Klüppelholz. "Compositional construction of most general controllers". In: *Acta Informatica* 52.4-5 (2015), pp. 443–482. DOI: 10.1007/s00236-015-0239-9.

[146] Alexander Knapp, Heribert Mühlberger, and Bernhard Reus. "Interpreting Knowledge-based Programs". In: *Programming Languages and Systems - 32nd European Symposium on Programming, ESOP 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22-27, 2023, Proceedings*. Ed. by Thomas Wies. Vol. 13990. Lecture Notes in Computer Science. Springer, 2023, pp. 253–280. DOI: 10.1007/978-3-031-30044-8_10. URL: https://doi.org/10.1007/978-3-031-30044-8%5C_10.

[147] 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. "Spectre Attacks: Exploiting Speculative Execution". In: *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 2019, pp. 1–19. DOI: 10.1109/SP.2019.00002. URL: https://doi.org/10.1109/SP.2019.00002.

[148] 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. "Spectre attacks: exploiting speculative execution". In: *Commun. ACM* 63.7 (2020), pp. 93–101. DOI: 10.1145/3399742. URL: https://doi.org/10.1145/3399742.

[149] Jeremy Kong and Alessio Lomuscio. "Model Checking Multi-Agent Systems against LDLK Specifications". In: *International Joint Conference on Artificial Intelligence, IJCAI 2017*. 2017. DOI: 10.24963/ijcai.2017/158.

[150] Jeremy Kong and Alessio Lomuscio. "Model Checking Multi-Agent Systems against LDLK Specifications on Finite Traces". In: *International Conference on Autonomous Agents and MultiAgent Systems, AAMAS 2018*. 2018.

[151] Elisavet Kozyri, Stephen Chong, and Andrew C. Myers. "Expressing Information Flow Properties". In: *Found. Trends Priv. Secur.* 3.1 (2022), pp. 1–102. DOI: 10.1561/3300000008. URL: https://doi.org/10.1561/3300000008.

[152] Hadas Kress-Gazit, Morteza Lahijanian, and Vasumathi Raman. "Synthesis for Robots: Guarantees and Feedback for Robot Behavior". In: *Annu. Rev. Control. Robotics Auton. Syst.* 1 (2018), pp. 211–236. DOI: 10.1146/ANNUREV-CONTROL-060117-104838. URL: https://doi.org/10.1146/annurev-control-060117-104838.

[153] Hillel Kugler and Itai Segall. "Compositional Synthesis of Reactive Systems from Live Sequence Chart Specifications". In: *Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*. Ed. by Stefan Kowalewski and

Anna Philippou. Vol. 5505. Lecture Notes in Computer Science. Springer, 2009, pp. 77–91. DOI: 10.1007/978-3-642-00768-2_9.

[154] O. Kupferman and M. Y. Vardi. "Synthesizing distributed Systems". In: *Logic in Computer Science (LICS)*. 2001.

[155] Orna Kupferman, Nir Piterman, and Moshe Y. Vardi. "Safraless Compositional Synthesis". In: *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*. Ed. by Thomas Ball and Robert B. Jones. Vol. 4144. Lecture Notes in Computer Science. Springer, 2006, pp. 31–44. DOI: 10.1007/11817963_6.

[156] Orna Kupferman, Nir Piterman, and Moshe Y. Vardi. "Safraless Compositional Synthesis". In: *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*. Ed. by Thomas Ball and Robert B. Jones. Vol. 4144. Lecture Notes in Computer Science. Springer, 2006, pp. 31–44. DOI: 10.1007/11817963_6. URL: https://doi.org/10.1007/11817963%5C_6.

[157] Orna Kupferman and Moshe Y. Vardi. "Model Checking of Safety Properties". In: *Computer Aided Verification, 11th International Conference, CAV '99, Trento, Italy, July 6-10, 1999, Proceedings*. Ed. by Nicolas Halbwachs and Doron A. Peled. Vol. 1633. Lecture Notes in Computer Science. Springer, 1999, pp. 172–183. DOI: 10.1007/3-540-48683-6_17. URL: https://doi.org/10.1007/3-540-48683-6%5C_17.

[158] Orna Kupferman and Moshe Y. Vardi. "Synthesizing Distributed Systems". In: *16th Annual IEEE Symposium on Logic in Computer Science, Boston, Massachusetts, USA, June 16-19, 2001, Proceedings*. IEEE Computer Society, 2001, pp. 389–398. DOI: 10.1109/LICS.2001.932514. URL: https://doi.org/10.1109/LICS.2001.932514.

[159] Leslie Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002. ISBN: 0-3211-4306-X. URL: http://research.microsoft.com/users/lamport/tla/book.html.

[160] Jérôme Lang and Bruno Zanuttini. "Probabilistic Knowledge-Based Programs". In: *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*. Ed. by Qiang Yang and Michael J. Wooldridge. AAAI Press, 2015, pp. 1594–1600. URL: http://ijcai.org/Abstract/15/228.

[161] Timo Latvala. "Efficient Model Checking of Safety Properties". In: *Model Checking Software, 10th International SPIN Workshop. Portland, OR, USA, May 9-10, 2003, Proceedings*. Ed. by Thomas Ball and Sriram K. Rajamani. Vol. 2648. Lecture Notes in Computer Science. Springer, 2003, pp. 74–88. DOI: 10.1007/3-540-44829-2_5. URL: https://doi.org/10.1007/3-540-44829-2%5C_5.

[162] C.I. Lewis and C.H. Langford. *Symbolic Logic*. Dover Books on Advanced Mathematics. Dover Publications, 1959. ISBN: 9780486601700. URL: https://books.google.de/books?id=2KSuAAAAIAAJ.

[163] Clarence Irving Lewis. *Symbolic Logic*. Ed. by Cooper Harold Langford. [New York]: Dover Publications, 1959.

[164] 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. "Meltdown: Reading Kernel Memory from User Space". In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018.

[165] 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. "Meltdown: Reading Kernel Memory from User Space". In: *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*. Ed. by William Enck and Adrienne Porter Felt. USENIX Association, 2018, pp. 973–990. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/lipp.

[166] Alessio Lomuscio, Hongyang Qu, and Franco Raimondi. "MCMAS: an open-source model checker for the verification of multi-agent systems". In: *Int. J. Softw. Tools Technol. Transf.* (2017). DOI: 10.1007/s10009-015-0378-x.

[167] P. Madhusudan and P. S. Thiagarajan. "A Decidable Class of Asynchronous Distributed Controllers". In: *CONCUR 2002 - Concurrency Theory, 13th International Conference, Brno, Czech Republic, August 20-23, 2002, Proceedings*. Ed. by Lubos Brim, Petr Jancar, Mojmír Kretínský, and Antonín Kucera. Vol. 2421. Lecture Notes in Computer Science. Springer, 2002, pp. 145–160. DOI: 10.1007/3-540-45694-5_11. URL: https://doi.org/10.1007/3-540-45694-5%5C_11.

[168] P. Madhusudan and P. S. Thiagarajan. "Distributed Controller Synthesis for Local Specifications". In: *Automata, Languages and Programming, 28th International Colloquium, ICALP 2001, Crete, Greece, July 8-12, 2001, Proceedings*. Ed. by Fernando Orejas, Paul G. Spirakis, and Jan van Leeuwen. Vol. 2076. Lecture Notes in Computer Science. Springer, 2001, pp. 396–407. DOI: 10.1007/3-540-48224-5_33. URL: https://doi.org/10.1007/3-540-48224-5%5C_33.

[169] Rupak Majumdar, Kaushik Mallik, Anne-Kathrin Schmuck, and Damien Zufferey. "Assume-Guarantee Distributed Synthesis". In: *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 39.11 (2020), pp. 3215–3226. DOI: 10.1109/TCAD.2020.3012641. URL: https://doi.org/10.1109/TCAD.2020.3012641.

[170] Zohar Manna and Amir Pnueli. "A Hierarchy of Temporal Properties". In: *Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing, Quebec City, Quebec, Canada, August 22-24, 1990*. Ed. by Cynthia Dwork. ACM, 1990, pp. 377–410. DOI: 10.1145/93385.93442. URL: https://doi.org/10.1145/93385.93442.

[171] Zohar Manna and Pierre Wolper. "Synthesis of Communicating Processes from Temporal Logic Specifications". In: *ACM Trans. Program. Lang. Syst.* 6.1 (1984), pp. 68–93. DOI: 10.1145/357233.357237. URL: https://doi.org/10.1145/357233.357237.

[172] Nicolas Markey. "Temporal logic with past is exponentially more succinct, Concurrency Column". In: *Bull. EATCS* (2003).

[173] Michael C. Martin, V. Benjamin Livshits, and Monica S. Lam. "Finding application errors and security flaws using PQL: a program query language". In: *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*. Ed. by Ralph E. Johnson and Richard P. Gabriel. ACM, 2005, pp. 365–383. DOI: 10.1145/1094811.1094840. URL: https://doi.org/10.1145/1094811.1094840.

[174] Ron van der Meyden. "Common Knowledge and Update in Finite Environments". In: *Inf. Comput.* 140.2 (1998). DOI: 10.1006/inco.1997.2679.

[175] Ron van der Meyden. "Finite State Implementations of Knowledge-Based Programs". In: *Foundations of Software Technology and Theoretical Computer Science, 16th Conference, Hyderabad, India, December 18-20, 1996, Proceedings*. Ed. by Vijay Chandru and V. Vinay. Vol. 1180. Lecture Notes in Computer Science. Springer, 1996, pp. 262–273. DOI: 10.1007/3-540-62034-6_55. URL: https://doi.org/10.1007/3-540-62034-6%5C_55.

[176] Ron van der Meyden and Nikolay V. Shilov. "Model Checking Knowledge and Time in Systems with Perfect Recall (Extended Abstract)". In: *Foundations of Software Technology and Theoretical Computer Science, 19th Conference, Chennai, India, December 13-15, 1999, Proceedings*. Ed. by C. Pandu Rangan, Venkatesh Raman, and Ramaswamy Ramanujam. Vol. 1738. Lecture Notes in Computer Science. Springer, 1999, pp. 432–445. DOI: 10.1007/3-540-46691-6_35. URL: https://doi.org/10.1007/3-540-46691-6%5C_35.

[177] Ron van der Meyden and Moshe Y. Vardi. "Synthesis from Knowledge-Based Specifications (Extended Abstract)". In: *CONCUR '98: Concurrency Theory, 9th International Conference, Nice, France, September 8-11, 1998, Proceedings*. Ed. by Davide Sangiorgi and Robert de Simone. Vol. 1466. Lecture Notes in Computer Science. Springer, 1998, pp. 34–49. DOI: 10.1007/BFB0055614. URL: https://doi.org/10.1007/BFb0055614.

[178] Ron van der Meyden and Thomas Wilke. "Synthesis of Distributed Systems from Knowledge-Based Specifications". In: *CONCUR 2005 - Concurrency Theory, 16th International Conference, CONCUR 2005, San Francisco, CA, USA, August 23-26, 2005, Proceedings*. Ed. by Martín Abadi and Luca de Alfaro. Vol. 3653. Lecture Notes in Computer Science. Springer, 2005, pp. 562–576. DOI: 10.1007/11539452_42. URL: https://doi.org/10.1007/11539452%5C_42.

[179] Philipp J. Meyer, Salomon Sickert, and Michael Luttenberger. "Strix: Explicit Reactive Synthesis Strikes Back!" In: *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*. Ed. by Hana Chockler and Georg Weissenbacher. Vol. 10981. Lecture Notes in Computer Science.

Springer, 2018, pp. 578–586. DOI: 10.1007/978-3-319-96145-3_31. URL: https://doi.org/10.1007/978-3-319-96145-3%5C_31.

[180] Edward F. Moore. "Gedanken-Experiments on Sequential Machines". In: *Automata Studies*. Ed. by C. E. Shannon and J. McCarthy. Princeton: Princeton University Press, 1956, pp. 129–154. ISBN: 9781400882618. DOI: doi:10.1515/9781400882618-006. URL: https://doi.org/10.1515/9781400882618-006.

[181] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. "How Amazon web services uses formal methods". In: *Commun. ACM* 58.4 (2015), pp. 66–73. DOI: 10.1145/2699417. URL: https://doi.org/10.1145/2699417.

[182] Luan Viet Nguyen, James Kapinski, Xiaoqing Jin, Jyotirmoy V. Deshmukh, and Taylor T. Johnson. "Hyperproperties of real-valued signals". In: *Proceedings of the 15th ACM-IEEE International Conference on Formal Methods and Models for System Design, MEMOCODE 2017, Vienna, Austria, September 29 - October 02, 2017*. Ed. by Jean-Pierre Talpin, Patricia Derler, and Klaus Schneider. ACM, 2017, pp. 104–113. DOI: 10.1145/3127041.3127058. URL: https://doi.org/10.1145/3127041.3127058.

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

[184] Amir Pnueli and Roni Rosner. "Distributed Reactive Systems Are Hard to Synthesize". In: *31st Annual Symposium on Foundations of Computer Science, St. Louis, Missouri, USA, October 22-24, 1990, Volume II*. IEEE Computer Society, 1990, pp. 746–757. DOI: 10.1109/FSCS.1990.89597.

[185] Amir Pnueli and Roni Rosner. "Distributed Reactive Systems Are Hard to Synthesize". In: *31st Annual Symposium on Foundations of Computer Science, St. Louis, Missouri, USA, October 22-24, 1990, Volume II*. IEEE Computer Society, 1990, pp. 746–757. DOI: 10.1109/FSCS.1990.89597. URL: https://doi.org/10.1109/FSCS.1990.89597.

[186] Amir Pnueli and Roni Rosner. "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, 1989, pp. 179–190. DOI: 10.1145/75277.75293. URL: https://doi.org/10.1145/75277.75293.

[187] Arthur N. Prior. *Time and Modality*. London: Oxford University Press, 1957.

[188] Markus N. Rabe. "A temporal logic approach to information-flow control". PhD thesis. Saarland University, 2016.

[189] Senthil Rajasekaran and Moshe Y. Vardi. "Verification and Realizability in Finite-Horizon Multiagent Systems". In: *International Conference on Principles of Knowledge Representation and Reasoning, KR 2022*. 2022.

[190] P. J. Ramadge and W. M. Wonham. "Supervisory Control of a Class of Discrete Event Processes". In: *SIAM Journal on Control and Optimization* 25.1 (1987), pp. 206–230. DOI: 10.1137/0325013. eprint: https://doi.org/10.1137/0325013. URL: https://doi.org/10.1137/0325013.

[191] Florian Renkin, Philipp Schlehuber-Caissier, Alexandre Duret-Lutz, and Adrien Pommellet. "Dissecting ltlsynt". In: *Formal Methods Syst. Des.* 61.2 (2022), pp. 248–289. DOI: 10.1007/S10703-022-00407-6. URL: https://doi.org/10.1007/s10703-022-00407-6.

[192] Andrei Sabelfeld and Andrew C. Myers. "Language-based information-flow security". In: *IEEE J. Sel. Areas Commun.* 21.1 (2003), pp. 5–19. DOI: 10.1109/JSAC.2002.806121. URL: https://doi.org/10.1109/JSAC.2002.806121.

[193] Andrei Sabelfeld and David Sands. "Declassification: Dimensions and principles". In: *J. Comput. Secur.* 17.5 (2009), pp. 517–548. DOI: 10.3233/JCS-2009-0352. URL: https://doi.org/10.3233/JCS-2009-0352.

[194] Michael G. Safonov. "A course in robust control theory: a convex approach [Book Reviews]". In: *IEEE Trans. Autom. Control.* 46.9 (2001), pp. 1510–1511. DOI: 10.1109/TAC.2001.948489. URL: https://doi.org/10.1109/TAC.2001.948489.

[195] Shmuel Safra. "On the Complexity of omega-Automata". In: *29th Annual Symposium on Foundations of Computer Science, White Plains, New York, USA, 24-26 October 1988*. IEEE Computer Society, 1988, pp. 319–327. DOI: 10.1109/SFCS.1988.21948. URL: https://doi.org/10.1109/SFCS.1988.21948.

[196] Sven Schewe and Bernd Finkbeiner. "Semi-automatic Distributed Synthesis". In: *Int. J. Found. Comput. Sci.* 18.1 (2007), pp. 113–138.

[197] Anne-Kathrin Schmuck, Thomas Moor, and Rupak Majumdar. "On the relation between reactive synthesis and supervisory control of non-terminating processes". In: *Discret. Event Dyn. Syst.* 30.1 (2020), pp. 81–124. DOI: 10.1007/S10626-019-00299-5. URL: https://doi.org/10.1007/s10626-019-00299-5.

[198] A. Segall. "Distributed network protocols". In: *IEEE Transactions on Information Theory* 29.1 (1983), pp. 23–35. DOI: 10.1109/TIT.1983.1056620.

[199] Koushik Sen and Grigore Rosu. "Generating Optimal Monitors for Extended Regular Expressions". In: *Third Workshop on Run-time Verification, RV@CAV 2003, Boulder, Colorado, USA, July 14, 2003*. Ed. by Oleg Sokolsky and Mahesh Viswanathan. Vol. 89. Electronic Notes in Theoretical Computer Science 2. Elsevier, 2003, pp. 226–245. DOI: 10.1016/S1571-0661(04)81051-X. URL: https://doi.org/10.1016/S1571-0661(04)81051-X.

[200] Nikolay V. Shilov and Natalya Olegovna Garanina. "Model checking knowledge and fixpoints". In: *Fixed Points in Computer Science, FICS 2002, Copenhagen, Denmark, 20-21 July 2002, Preliminary Proceedings*. Ed. by Zoltán Ésik and Anna Ingólfsdóttir. Vol. NS-02-2. BRICS Notes Series. University of Aarhus, 2002, pp. 25–39.

[201] Aravinda Prasad Sistla. "Theoretical issues in the design and verification of distributed systems". PhD thesis. Harvard University, 1983.

[202] Paulo Tabuada. *Verification and control of hybrid systems: a symbolic approach*. Springer Science & Business Media, 2009.

[203] Alfred Tarski. "A lattice-theoretical fixpoint theorem and its applications." In: (1955).

[204] Leander Tentrup. "Symbolic reactive synthesis". PhD thesis. Saarland University, Saarbrücken, Germany, 2019. URL: https://publikationen.sulb.uni-saarland.de/handle/20.500.11880/28497.

[205] Moshe Y. Vardi. "From Philosophical to Industrial Logics". In: *Logic and Its Applications, Third Indian Conference, ICLA 2009, Chennai, India, January 7-11, 2009. Proceedings*. Ed. by Ramaswamy Ramanujam and Sundar Sarukkai. Vol. 5378. Lecture Notes in Computer Science. Springer, 2009, pp. 89–115. DOI: 10.1007/978-3-540-92701-3_7. URL: https://doi.org/10.1007/978-3-540-92701-3%5C_7.

[206] Moshe Y. Vardi. "Implementing Knowledge-Based Programs". In: *Proceedings of the Sixth Conference on Theoretical Aspects of Rationality and Knowledge, De Zeeuwse Stromen, The Netherlands, March 17-20 1996*. Ed. by Yoav Shoham. Morgan Kaufmann, 1996, pp. 15–30.

[207] Moshe Y. Vardi and Pierre Wolper. "Reasoning About Infinite Computations". In: *Inf. Comput.* 115.1 (1994), pp. 1–37. DOI: 10.1006/INCO.1994.1092. URL: https://doi.org/10.1006/inco.1994.1092.

[208] Pierre Wolper, Moshe Y. Vardi, and A. Prasad Sistla. "Reasoning about Infinite Computation Paths (Extended Abstract)". In: *24th Annual Symposium on Foundations of Computer Science, Tucson, Arizona, USA, 7-9 November 1983*. IEEE Computer Society, 1983, pp. 185–194. DOI: 10.1109/SFCS.1983.51. URL: https://doi.org/10.1109/SFCS.1983.51.

[209] WM Wonham, Kai Cai, and Karen Rudie. "Supervisory control of discrete-event systems: A brief history". In: *Annual Reviews in Control* 45 (2018), pp. 250–256.

[210] Bozena Wozna, Alessio Lomuscio, and Wojciech Penczek. "Bounded model checking for knowledge and real time". In: *4th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2005), July 25-29, 2005, Utrecht, The Netherlands*. Ed. by Frank Dignum, Virginia Dignum, Sven Koenig, Sarit Kraus, Munindar P. Singh, and Michael J. Wooldridge. ACM, 2005, pp. 165–172. DOI: 10.1145/1082473.1082498. URL: https://doi.org/10.1145/1082473.1082498.

[211] Georg Henrik von Wright. *An Essay in Modal Logic*. Amsterdam, Netherlands, 1951.

[212] Xiang Yin, Bingzhao Gao, and Xiao Yu. "Formal synthesis of controllers for safety-critical autonomous systems: Developments and challenges". In: *Annu. Rev. Control.* 57 (2024), p. 100940. DOI: 10.1016/J.ARCONTROL.2024.100940. URL: https://doi.org/10.1016/j.arcontrol.2024.100940.

[213] Steve Zdancewic and Andrew C. Myers. "Observational Determinism for Concurrent Program Security". In: *16th IEEE Computer Security Foundations Workshop (CSFW-16 2003), 30 June - 2 July 2003, Pacific Grove, CA, USA*. IEEE Computer Society, 2003, p. 29. DOI: 10.1109/CSFW.2003.1212703. URL: https://doi.org/10.1109/CSFW.2003.1212703.