VERIFICATION OF PROBABILISTIC HYPERPROPERTIES ON MARKOV MODELS

By

Oyendrila Dobe

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

Computer Science - Doctor of Philosophy

2024

ABSTRACT

Formal verification encapsulates the process of ensuring the correctness of systems with respect to user-specified requirements, expressed as formal properties. This dissertation explores the different aspects of the verification of systems involving uncertainty, specified at an abstract level as Markov models, against system-level specifications, expressed as hyperproperties.

In this context, we refer to models as frameworks used to capture the behaviour of systems while abstracting away the exact details of its inner workings. Such system models are rarely deterministic e.g., we may choose to simplify or remove elements that are irrelevant to the verification, we might not be aware of exact details of certain components in the system, or we might not be able to observe every aspect of the system. To this end, we chose to represent systems as Markov models due to their flexibility in representing uncertainty (in terms of nondeterminism, randomization, and partial observability), and its simplicity in using the current state to determine the future evolution of the system.

The current ubiquitous nature of software demands verification of not only its isolated behaviours, but also ones relating to security, privacy, robustness, and efficiency. These additional requirements are more challenging to verify as they require simultaneous reasoning across multiple system behaviours. Such requirements can be formally specified as hyperproperties. Prominent examples include noninterference of secret inputs on publicly observable outputs, observational determinism of public outputs, optimal path planning in robotics, individual fairness in models, side-channel timing attacks, conformance of different system versions, etc.

Given this combination of model and properties, we explore the following:

- We explored the parameter synthesis problem and developed an SMT-based solution for synthesizing values for unknown or missing parameters of the system, under the assumption of satisfaction of a given specification.

- We generalize the probabilistic hyperlogic HyperPCTL by extending it to formally specify

requirements involving nondeterministic choices and rewards in models. We discuss the expressive power of this extended logic, and how it makes the model checking problem for this logic undecidable, in general.

- To provide tractable solutions to the model checking problem, we focused on a decidable fragment of HyperPCTL with a single existential quantifier and proposed an SMT-encoding-based model checking algorithm.

- We identify two different fragments of HyperPCTL based on their ability to express the reachability requirement in most of our applications. For these two different fragments, we provide a randomized convex addition of a scheduler and a distribution-based approach respectively.

- We have developed prototypical implementations for each of the proposed algorithms. Specifically, to model check the nondeterministic and reward property extensions, we have built a tool HyperProb, which implements our SMT-based algorithm to symbolically model check restricted fragments of HyperPCTL.

- To tackle the challenges of scalability, we explored statistical model checking as a possible solution to model check a fragment of HyperPLTL and extended the tool PLASMA to evaluate our theory.

I dedicate this dissertation to my parents, Juthika and Swapan Dobe,
for always letting me pursue my (often eccentric) ideas
&
to my younger self
for choosing to take on this life-changing journey.

ACKNOWLEDGEMENTS

As I run the final lap of this marathon of obtaining this degree, I cannot help but feel grateful for all the amazing people, memories, and lessons I have encountered on my way. This journey has humbled me and helped me understand myself better beyond any limit I had ever imagined. For everything I have accomplished, I have done so by standing on the shoulders of giants who have made this path a little easier for me at every step. My only hope is to be able to pay it forward in life. I want to use this space to give credit to a few notable people who walked along with me on this journey and helped me pace my way to the end line.

Adhering to traditions, I would first like to extend my gratitude towards my advisor and mentor, Borzoo Bonakdarpour. I am thankful to him for taking me under his umbrella at a confusing point in my graduate studies and introducing me to the world of hyperproperties. His guidance all along the way has helped me grow both as a researcher and a person. Despite our differences and several tough conversations, his unwavering support has, time and again, kept me on track to finish this marathon. At its core, this advisor-advisee relationship has been fuelled by cooked-to-perfection barbecues, bar-night discussions, late-night paper submissions (delayed by my silly mistakes), and a shared trait of stubbornness. It is my honour to have a life-long conflict of interest with him for future paper submissions.

I am thankful to my committee members Prof. Betty Cheng, Prof. Sandeep Kulkarni, and Prof. Yiming Deng, for their time and feedback on my research, and for accommodating my requests during my comprehensive exam and final defense. I am especially thankful to Prof. Cheng for her detailed feedback on my dissertation. I would also like to extend my thanks to all my collaborators, especially Prof. Erika Abraham and Prof. Ezio Bartocci. It has been a great experience working alongside them and learning from them during our collaborations. I have been fortunate to have crossed paths with Stefan Schuup and Lina Gerlach, collaborators that I now call friends.

Taking a step back, I want to thank Prof. Amiya Halder, my beloved HoD *ma'am*,

B.N. Ray *uncle*, and Chatterjee *uncle* for encouraging me to pursue higher education. I am sincerely grateful to Sinha *jethu* for all his help and mentorship during the application process.

Next in my gratitude queue would be my fellow lab-mates of TART: Ritam Ganguly, Anik Momtaz, Tzu-Han Hsu, Eshita Zaman, and Arshia Rafieioskouei. Thank you for making the lab a fun workplace. It was a pleasure discussing formal methods, politics, food, frustrations, and prank ideas with you. I have learnt and soaked different attributes of your work ethics and cherish our time spent together.

Among my fellows runners, running their own marathons, I thank- *Debrudra Mitra* and *Arna Ganguly* for being there at one of my lowest points in this journey; I will forever be indebted to you for lending me your shoulders, *Soham Vanage* and *Ritam Ganguly* for being the initiators of crazy ideas that Debrudra da and I jumped in to pursue without complaints, *Ali Saffary* for always lending me a listening pair of ears, *David Ackley* for bringing weird humour to our conversations, *Akash Dutta* for being my partner-in-crime, *Umesh Chinalachi* for his emotionless honest advice, *Tiana Zefreen* and *Anik Momtaz* for accepting my weirdness, *Saurabh Mishra* for being the life of our parties, *Arya Gupta* for our stimulating discussions, and *Emma*, *Sabrina* for making me feel at home in the final year. I also thank *Nishan Chatterjee* and *Vyomdhaval Bhatt* for being my comfort places.

I am grateful for the existence of the MSU Badminton Club and its members, especially Jayci Simon; the club helped me escape the daily grind, find my strength, and express myself authentically. As part of the Spartan Girls Who Code community, I have learned so much about mentoring, leadership, and preparing the next generation of steminists. I am thankful to Teresa Vandersloot and Prof. Laura Dillon for accepting me as a part of this amazing community run by strong fellow Spartans. I hope to continue to support this community going forward.

A very important part of my graduate school experience was my internships which convinced me of my future career choice. I am indebted to Nafi Diallo for believing in me and

selecting me for my first internship opportunity. I enjoyed working under brilliant managers like Temesghen Kahsai and Aaron Tomb who gave me interesting problems, the freedom to pursue them independently, and their guidance. I thank my mentors Byron Cook and Rustan Leino for inspiring me by sharing their passions and visions. And finally, I am extremely grateful to have found a mentor in Nadia Labai who made my experiences enjoyable.

I am grateful to the CSE department at MSU for providing the space and financial support for my research. I want to thank our non-teaching staff, especially, Vincent Mattison and Brenda Hodge for making most of my department-related issues vanish in no time.

My research has been supported by generous funds from the National Science Foundation and I am grateful for their continued support towards pushing the boundaries of science.

My final words are for the most important people in my life, my parents. Ma and Bapi thank you for being the source of my strength, inspiration, and clarity. It has been a long journey and you have sacrificed a lot for it. I hope to make it all worth it. Thank you for being patient with me.

TABLE OF CONTENTS

LIST OF TABLES . . . . . **x**

LIST OF FIGURES . . . . . **xi**

Chapter 1 Introduction **1**
1.1 Modelling of Systems 2
1.2 Specifications 4
1.3 Model Checking for Hyperproperties 7
1.4 Motivation 9
1.5 Thesis statement 13
1.6 Contributions 13
1.7 Organization 15

Chapter 2 Preliminaries **18**
2.1 Discrete-Time Markov models 18
2.2 Discrete-Time Markov Models with Rewards 21
2.3 Probabilistic Hyperproperties 25

Chapter 3 Parameter Synthesis for Probabilistic Hyperproperties **29**
3.1 Introduction 29
3.2 Parameter Synthesis Algorithm for ReachHyperPCTL 35
3.3 Case Studies and Evaluation 41
3.4 Summary 50

Chapter 4 Probabilistic Hyperproperties with Nondeterminism **51**
4.1 Introduction 51
4.2 HyperPCTL for MDPs 54
4.3 The Expressiveness Power of HyperPCTL 57
4.4 Applications of HyperPCTL on MDPs 62
4.5 Summary 64

Chapter 5 Decidable Fragment of Probabilistic Hyperproperties with Nondeterminism **66**
5.1 Introduction 66
5.2 Proof of decidability of the restricted fragment 66
5.3 Model Checking for Non-probabilistic Memoryless Schedulers 70
5.4 Evaluation 78
5.5 Summary 80

Chapter 6 Probabilistic Hyperproperties with Rewards **82**
6.1 Introduction 82
6.2 HyperPCTL with Rewards 85
6.3 Applications of HyperPCTL with Rewards 91
6.4 Model Checking Algorithm for Reward Operators 94

6.5 Evaluation . . . 97
6.6 Summary . . . 101

Chapter 7 HyperProb: A Model Checker for Probabilistic Hyperproperties . . . 102
7.1 Introduction . . . 102
7.2 Input to the Tool . . . 103
7.3 Tool Structure and Usage . . . 105
7.4 Evaluation . . . 108
7.5 Summary . . . 113

Chapter 8 Efficient Probabilistic Model Checking for Relational Reachability . . . 114
8.1 Preliminaries . . . 115
8.2 Model checking algorithm for $(2\sigma 2s)$ fragment . . . 116
8.3 Distribution-Based Approach for the $(1\sigma 1s)$ Fragment . . . 120
8.4 Experimental Evaluation . . . 126
8.5 Summary . . . 133

Chapter 9 Lightweight Verification of Hyperproperties . . . 134
9.1 Introduction . . . 134
9.2 Preliminaries . . . 137
9.3 Problem Formulation . . . 139
9.4 Approach . . . 141
9.5 Case Studies . . . 147
9.6 Experimentation/Evaluation . . . 153
9.7 Summary . . . 159

Chapter 10 Related Work . . . 160
10.1 Discrete and Continuous Time Logics . . . 160
10.2 Hyperproperties . . . 162
10.3 Parameter Synthesis . . . 163
10.4 Model Checking of Probabilistic Hyperproperties . . . 165
10.5 Statistical Model Checking . . . 166

Chapter 11 Conclusion and Future Work . . . 168
11.1 Summary . . . 168
11.2 Future Work . . . 170

BIBLIOGRAPHY . . . 174

LIST OF TABLES

Table 1.1: Details of publications. 15

Table 3.1: Experimental results for thread scheduling. 46

Table 5.1: Experimental results. **TA:** Timing attack. **PW:** Password leakage. **TS:** Thread scheduling. **PC:** Probabilistic conformance. 80

Table 6.1: Semantics of $\varphi = \mathbb{P}(\varphi_1 \mathcal{U} \varphi_2)$, partly depending on $p = \sum_{s' \in \mathcal{S}^\sigma} P(s, s') \cdot [\![\varphi]\!]_{\mathcal{M},\sigma,s'} \in [0,1] \cup \{\bot\}$. Here, $[\![.]\!]$ is short for $[\![.]\!]_{\mathcal{M},\sigma,s}$. 89

Table 6.2: Experimental results. **VR:** Verification result. **TA**: Timing attack. **PC**: Probabilistic conformance. **RO**: Robotics example. **HS**: Herman's algorithm. **IJ**: Israeli-Jaflon's algorithm. ✓: the result is true. ×: the result is false. 99

Table 7.1: Experimental results and comparison. **TA:** Timing attack. **PW:** Password leakage. **TS:** Thread scheduling. **PC:** Probabilistic conformance. **TO:** Timeout. **N:** Prototype presented in [ÁBBD20b]. **O:** HyperProb,. **SE:** SMT encoding. **SS:** SMT solving. #op: Formula size (number of operators). #st: Number of states. #tr: Number of transitions. 112

Table 8.1: Parameters used for scheduling three tasks τ_0, τ_1, τ_2. 129

Table 8.2: Experimental results for scheduler generation using Alg. 17. **#st**: number of states in the model, **#tran**: number of transitions in the model, **#act**: number of actions in the model. 131

Table 8.3: Experimental results for scheduler generation using Alg. 18. **#st**: number of states in the model, **#tran**: number of transitions in the model, **#act**: number of actions in the model. 132

Table 9.1: Model details of grey-box case studies. 155

Table 9.2: Data from experimentation. #sch: number of scheduler-tuples sampled, #tr: number of trace tuples sampled per scheduler tuple, k: length of traces sampled. $\alpha = \beta = 0.01$. 156

LIST OF FIGURES

Figure 1.1: Models to represent a roll of a dice. 3

Figure 1.2: Traces considered in different types of logic. 5

Figure 1.3: Implementation of dice by coin tosses using Knuth-Yao protocol. 7

Figure 1.4: Overall idea of model checking. 8

Figure 2.1: Example DTMC. 19

Figure 2.2: MDP showing the actions and probabilistic distributions. 20

Figure 2.3: Example Markov models with rewards. 22

Figure 3.1: The randomized response protocol. 30

Figure 3.2: Semantics example. 35

Figure 3.3: A PDTMC (left) and the result of eliminating s_1 in [DJJ$^+$15] (mid) and in Alg. 4 (right). 39

Figure 3.4: Synthesized probability distribution for the randomized response protocol. 43

Figure 3.5: Parametric die-coin using Knuth-Yao protocol. 45

Figure 3.6: Parametric DTMC for the probabilistic noninterference example program with two threads th_1 and th_2. 46

Figure 3.7: Synthesized probability distribution for a randomized scheduler. 47

Figure 3.8: A parametric DTMC model for the dining cryptographers protocol with three cryptographers and three biased coins. 49

Figure 4.1: Example DTMC. 52

Figure 4.2: MDPs that require different types of schedulers. 52

Figure 4.3: Modular exponentiation. 62

Figure 4.4: String comparison. 63

Figure 5.1: Example of mapping SAT to HyperPCTL model checking. 68

Figure 6.1: Example Markov models. 83

Figure 6.2: HyperPCTL syntax. 85

Figure 6.3: Existing Semantics rules for HyperPCTL. 87

Figure 6.4: Semantics for reward operators in HyperPCTL. 88

Figure 6.5: Modular exponentiation in RSA. 91

Figure 6.6: The maze on the left satisfies φ_{target}, while on the right it violates φ_{target}. 93

Figure 6.7: Herman's algorithm [Her90] for process i and example for three processes. 94

Figure 7.1: PRISM model generating the MDP in Fig.7.2 103

Figure 7.2: MDP of the PRISM program in Fig 7.1. 103

Figure 7.3: Grammar defining HyperPCTL inputs to HyperProb, where the NUM token is a decimal number and NAME is a non-empty string. 104

Figure 7.4: Overview of the docker container with the tool and its dependencies. . . . 106

Figure 7.5: Dataflow inside the tool. 107

Figure 7.6: Modular exponentiation. 109

Figure 7.7: String comparison. 110

Figure 8.1: (Partial) MDP modelling information leakage via side-channel. 117

Figure 8.2: DTMCs induced by different schedulers on MDP in Fig. 8.1. 119

Figure 8.3: Modular exponentiation. 127

Figure 8.4: String comparison. 128

Figure 9.1: Two robots attempting to reach the same goal. 149

Figure 9.2: Grid divided into regions to ensure opacity. 150

Figure 9.3: DC with $n = 4$ ($Pr \geq 0.1 \pm 0.01$). 157

Figure 9.4: Plots showing the change in ratio based on sampling across schedulers. . 158

Chapter 1

Introduction

A system is described as a group of interrelated components that work together to accomplish a single goal. Everything around us can be considered a system - mobile phones, traffic management, robotic arms, spacecraft, stock prices, patient flow in hospitals, etc. When building these systems our aim should be to ensure their robustness, security, and conformance to specific expectations. This problem is comparatively simpler if the systems are deterministic. In reality, most systems are rarely deterministic. There is always a degree of uncertainty involved in these systems contributed by noise disturbances, random time delays, stochastic inputs, human errors, unknown inputs from environments, randomness in algorithms to break deadlocks, assigning probability in output distributions in ML models, etc. Hence, we are interested in reasoning about systems under uncertainty. In the rest of this material, we specifically use the term *randomness* for cases where we can quantify the uncertainty of the system and we use the term *uncertainty* to only describe cases where the uncertainty is not quantifiable and has to be represented nondeterministically.

Formal verification is described as the process of using mathematical reasoning to prove or disprove the correctness of a system against a set of specifications. For our purposes, we use Markov models to represent the systems we study. When verifying systems, we need to consider multiple observations of the system simultaneously to ensure its correctness and security. Such specifications are referred to as *hyperproperties*. The lack of organized theory and verification tools for handling such properties has motivated the range of research

described here.

1.1 Modelling of Systems

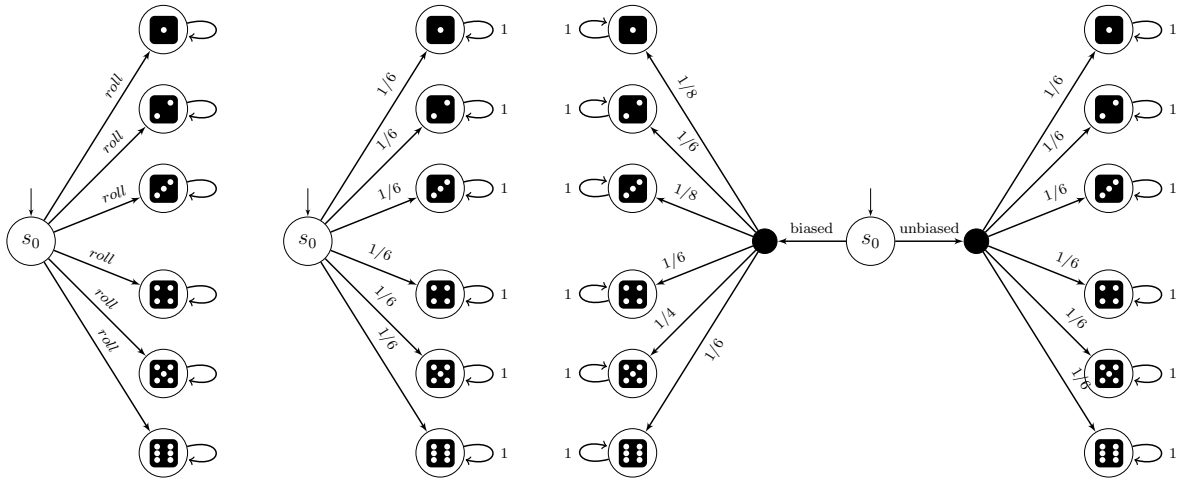
Models — A system exhibits observable behaviours due to their interactions between its sub-components as well as its environment. In verification, we study the extent to which these behaviours are analogous to the expectations for the same. However, when verifying a particular aspect of the system, we might not need all the intricate details of its working and can abstract the components irrelevant to the current context. However, these models should be mathematically precise and unambiguous, and obtaining these models is no easy feat. We can find incompleteness, ambiguities, and inconsistencies even while just accurately modeling a complex system [BK08].

Historically, such system models have been generated in the form of *guarded commands* [Dij75], *Petri nets* [Rei85], *process algebra* [DN11], *Kripke structures* (Sec.1.1), *timed automata* [RD94], etc.

Transition Systems — These are directed graphs where states of the system are abstracted as nodes, and the changes in the states are modeled as edges of the graph, known as transitions [BK08]. *Kripke structures* [Kri63] is a variation of transition systems where each state is labeled by *atomic propositions* to express simple facts about it. Each transition may be labeled by the *action* that causes the corresponding evolution of the system.

Example 1. Let us consider the simple example of modeling a dice. We describe this using a Kripke structure in Fig. 1.1a. We use s_0 to denote the initial state with the incoming transition from nowhere. Using six actions, labeled *roll*, we show the possible transitions to the six faces of the dice, written as $die_i, i \in \{1, \ldots, 6\}$.

Although Kripke structures are enough to represent the states of a system along with how the system evolves, they cannot give any details on how likely a transition is compared



(a) Kripke structure (b) DTMC (c) Markov Decision Process

Figure 1.1: Models to represent a roll of a dice.

to the others. Hence, we need a more efficient modeling structure to capture the probabilistic distributions of the transitions from one state to another.

Markov Processes — They capture the stochastic nature of systems. They abide by the Markov property which requires the transition from a state to its next to depend only on the current state and none of the previously visited states. In our work, we only consider discrete-time systems. Discrete-Time Markov Chains (DTMC) [Chu67] ensure each transition occurs at discrete moments in time as opposed to continuous time systems where the evolution of the system is described by the rate of transition at each state.

Example 2. In Fig. 1.1b, we describe the roll of the dice with the additional information of how probable each of the transitions is. In this case, each of the transitions is equally likely, hence they have a value of 1/6. Although not every system has equally likely transitions, the sum of the probabilities of all its transitions should always add up to one.

In order to represent our systems as DTMCs, we need to be aware of the probabilistic distributions it entails. However, in reality, when modeling systems that interact with their environment, it might not be possible to have the exact details of the inputs the environment

provides. In such cases, Markov Decision Processes (MDPs) [Put90] allow us to represent the actions of the environment using non-determinism.

Example 3. In Fig. 1.1c, we describe the rolling of two possible die. If we choose the unbiased dice, we will get transitions similar to Fig. 1.1b. But if we choose a biased dice, we can get skewed transitions. In this biased case, the outcome of getting a face with five is more likely than the others. Notice how we have not specified how likely it is to get a biased die compared to an unbiased one. This choice of which die to use can be non-deterministic.

In the sections and chapters that follow, we specifically consider these two types of models and describe them formally.

We use DTMCs to model purely probabilistic systems and MDPs to model probabilistic systems with non-deterministic actions and evolving in discrete time.

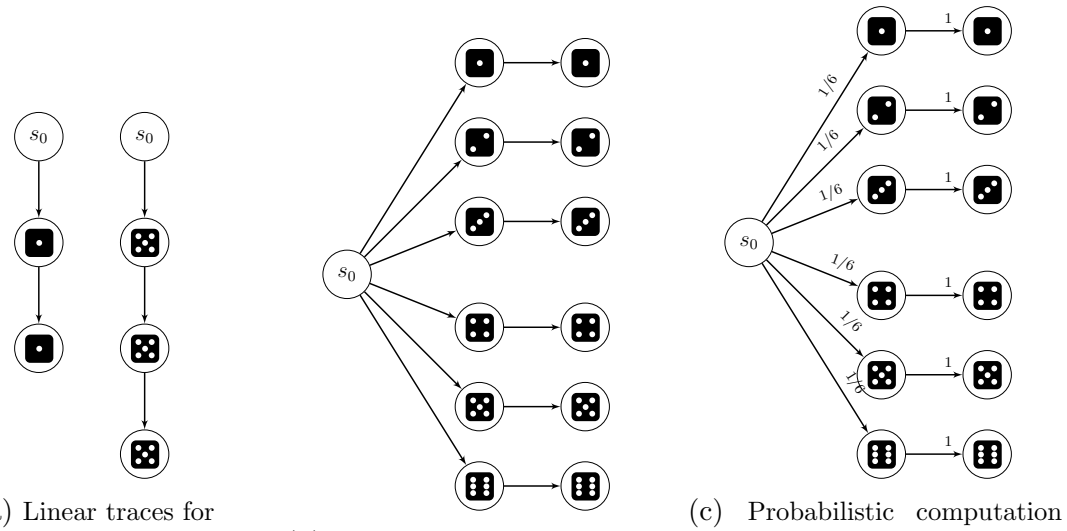
1.2 Specifications

Each state in our model is a snapshot of the system at that point in time. It reflects the values of variables of the system in that time instance. In verification, our aim is to check the conformance of these values in states against expected values. We express our requirements as *specifications*. We can use propositional logic to simply verify if certain values are true in a specific state. However, the states transition into each other with time. In discrete-time systems, we assume states evolve at each unit time step. In such situations, propositional logic is not enough to reason about the timely evolution of the system. *Temporal logics* introduces the concept of time into logic. This term was coined by *Arthur Prior* [Uck12]. In program verification, Amir Pnueli [Pnu77] presented the basis for temporal reasoning.

Example 1. Let us assume *Drink(coffee)* is a Boolean function that returns *true* if I am drinking coffee and false otherwise. We can express '*I am drinking coffee*' in propositional logic as *Drink*(*coffee*). To express '*I will drink coffee*', we will need temporal logic. $\Diamond$ *Drink*(*coffee*) means that either at the current moment or at some time in the future, I will be drinking coffee.

1.2.1 Trace-based Specifications

Traces — This forms the basis of our reasoning over systems. Each trace represents an execution of the system we have modeled. In verification, we essentially reason about the truth of our specifications on these traces. We can imagine them as a sequence of states as shown in Fig. 1.2a. The model in Fig. 1.1a can produce these traces.



(a) Linear traces for LTL

(b) Computation trees for CTL

(c) Probabilistic computation trees for PCTL

Figure 1.2: Traces considered in different types of logic.

We can use *Linear Temporal Logic*(LTL) [Pnu77] to reason about these individual traces. Using this logic, we can reason if something happens across all traces or not. However, we often also need to reason simultaneously about all paths that are possible from a given state in a possibilistic manner as shown in Fig. 1.2b for the model in Fig.1.1a. This concept was introduced in [BAMP81] as *Computational Tree Logic* (CTL). Using this logic, we argue if something happens at least once or always across the tree. An obvious extension of this

form of logic would be to argue about the probability of something happening instead of just its possibility. This can be done using Probabilistic Computation Tree Logic (PCTL) first introduced in [HJ94]. There have been several extensions of these logics to increase its expressive power. The specifications we define using these logics are called *trace properties* as each property corresponds to a set of traces that satisfy the property. So, a system satisfies the given specification if the set of traces generated by the system is a *subset* of the traces accepted by this property.

Example 2. In LTL, we can specify $\Diamond die_i$ to check if we can reach a state labeled die_i now or in the future. In PCTL, we can specify $P_{=0.16}(\Diamond die_1)$ to check that the sum of probability of reaching states labeled die_1 is equal to 0.16. Notice that we might not be able to reach such a state in every execution. Hence we cannot use LTL to reason about it, instead, we have to consider a branching time logic.

1.2.2 System Specifications

Several policies, however, cannot be verified by examining just the set of traces of a system. An easy example can be *conformance*. During the development of a system, we can design it according to our needs. In this phase, we usually abstract away the details of implementation. Once built, we would then need to check if the implementation *conforms* with the design modeled, based on some criteria. In Fig. 1.3 on the left we show the DTMC for a die roll and on the left we show how the same idea can be implemented using multiple coin tosses according to the Knuth Yao protocol [KY76]. Conformance, in this case, can aim to verify if the probabilistic distribution of getting the different faces of the dice is the same in both cases. To verify such a property we will need to compare the computation trees of the two models simultaneously. Such specifications are called *hyperproperties*.

Hyperproperties — These are used to express relations between executions of a system. Each hyperproperty is a set of sets of traces, $\mathcal{H}$, that satisfy it. Hence, these are also known as *system properties*. A system is said to satisfy a hyperproperty if it generates

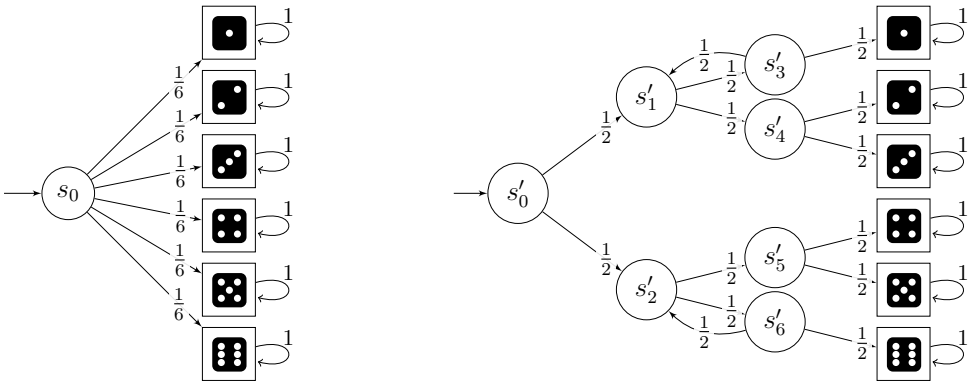


Figure 1.3: Implementation of dice by coin tosses using Knuth-Yao protocol.

a set of traces that belongs to the set $\mathcal{H}$. Several security policies like *non-interference* and *observational determinism*, system properties like average response time, and average up-time, are defined as hyperproperties. It allows us to reason and compare states across multiple traces, which may or may not belong to the same model. Recent formalization of the general concept of hyperproperties [CS08] has inspired extensions of previously mentioned logics to its hyper counterparts. PCTL, specifically, has been extended into HyperPCTL allowing quantification over initial states of a model [ÁB18].

Example 3. Compared to e.g. 1.2, to express '*I drink coffee everyday at the same time*', we will need hyperproperties. The property shown below says that if I am drinking coffee at some time during one day (denoted by s), I'll always drink coffee at the same time on any other day (denoted by s').

$$\forall s. \forall s'. \Box \left(\mathit{Drink}(\mathit{coffee})_s \leftrightarrow \mathit{Drink}(\mathit{coffee})_{s'} \right)$$

1.3 Model Checking for Hyperproperties

Model checking, shown in Fig. 1.4, is a technique of formal verification that involves reasoning along all possible behaviours of the model to prove or disprove a specification. This is a fully automated ('push-button') technique. A strong assumption in this process is that the model we use, to represent the system, and the specification used, to represent the

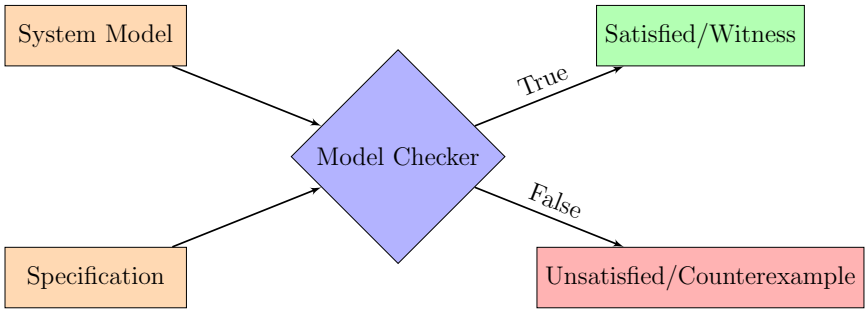


Figure 1.4: Overall idea of model checking.

requirement, is accurate. Note that in model checking we can pose our specification either as a *forall* case (commonly referred to as *verification*) or an *exists* case (commonly referred to as *synthesis*). For verification, we intend to check the property across all combinations of traces (depending on the number of quantifiers in the specification). Hence, the specification can either be *satisfied* or we can provide *counterexample*. For synthesis, we intend to search for a combination of traces. Hence, a satisfaction would yield a *witness* and a failure would result in an *unsatisfied* result.

Since model checking aims to check all possible cases in a system, it is rigorous. It has been extensively applied for safety-critical systems instead of testing. One of the main features is that it returns a *counterexample* when a specification fails in the system. This helps to find the location of the failure. In case of hyperproperties, the model is usually a composition of several copies of the same system or multiple systems. The number of models in the composition is determined by the number of quantifiers in the hyperproperty, which essentially is the number of traces we are comparing simultaneously.

The challenge lies in finding optimized model checking algorithms that scale well for large state spaces. This is handled by finding a balance between the accuracy of the model checker and the extent of model checking involved. On one hand, we can exhaustively model check the whole system which gives the most accurate result but is time and memory-consuming. On the other hand, we can use bounded model checking (where we limit the size of traces) or statistical model checking (where we model check a sample of the traces and infer the

result with some confidence) which are less accurate but faster and scales better. We have to make our choice depending on the system we are model checking - exhaustive for safety-critical systems and less accurate model checking for cases where have tight time and memory budget.

In all our subsequent work we explore different aspects and challenges of model checking probabilistic hyperproperties on DTMCs and MDPs.

1.4 Motivation

Markov models have been extensively used to represent systems due to their transitions being independent of their history. Although keeping a history does have its advantages, Markov models provide a simpler insight into the workings of the system, thus, assisting in its verification. They have been widely used in modeling randomized distributed systems to represent self-stabilizing algorithms, consensus protocols, mutual exclusion, conformance, etc. In communication protocols, they have been used to represent broadcasting protocols, device discovery, IP configuration protocols, etc. In security, they have been used to model contract signing, message exchange, cryptography, non-repudiation protocols, etc. In terms of path planning synthesis, Markov models have been widely used to design grids and movement of UAVs, robots, and vehicles. Note that the original works introducing the above ideas might not have suggested the use of Markov chains for their representation, but different model checking software has modeled the above problems as Markov Chains to verify the systems.

A recent study [CS08] has shown that several important policies related to security, privacy, and robustness are hyperproperties. Below are a few of the concepts that cannot be represented or verified as *trace properties*.

- **Secure information flow** — When verifying systems, one of the main aspects is

the interaction between systems and its users. Each user has an authorization level assigned to them (access to high/secret or low/public information). When interacting with the system, they provide inputs to the system and observe its outputs. Information leak refers to the unwanted exposure of sensitive information to unauthorized users, endangering the privacy of the information owners. Information leaks cannot be detected by observing individual outputs. In such cases, we look for similarity of output among traces possessing different secret inputs. Security policies regarding information flow enforce restrictions on how the system generates its outputs and ensure there is no direct flow of information from secret channels to publicly observable channels. Probabilistic information flow argues that the probabilistic distribution of the public channel outputs does not differ drastically based on different secret inputs. *Non-interference* [GM82] enforces that for every trace t in the system, there should be another trace that has none of the high inputs in t but produces the same low observable output as t. The observable outputs can be execution time, power consumption, value of variables, etc. The essence of this concept is to prevent the user from knowing that a specific trace is caused by specific secret values only. *Observational determinism* [ZM03] enforces the equivalence between corresponding states of two traces if they both begin in states that are *observably* equivalent. This will prevent an intruder from understanding how secret values change the course of the execution of a system. Both these policies require simultaneous observation of multiple traces, and hence cannot be expressed by existing trace properties.

Service level agreements — When designing systems, we aim to ensure their performance satisfies certain statistics specified in *service level agreements*. One such statistic is the average response time of a system, i.e., the average time that elapses between the reception of a request and the execution of its response. We can verify if each trace has an individual response time within the bound provided, but that will be a stronger property than necessary. We can certainly have a trace that has a much higher

response time, or one with a significantly lower response time, yet the average across all traces can still be within the specified bound. Thus we need to consider multiple traces simultaneously, to find the average value, requiring the use of hyperproperties.

Differential privacy — When working with anonymous data, one of the major policies is to ensure that we do not leak any information or indication about the owner of the information, either directly or via some statistic. *Differential privacy* [DR14] ensures that the output of a query on a dataset is similar irrespective of whether our data is contributing to the dataset. This is of immense importance in the current world. For example, we all receive targeted advertisements but we don't want them to reveal private details of our lives (say, the skin problems we suffer from) through the ads that appear in our internet browser. Since we need to argue about both cases where the specific owner is and is not a part of the database simultaneously, we will need hyperproperties to express this policy.

Probabilistic causation — This focuses on arguing about how likely an event is to cause another. The relation is bidirectional in the sense that the cause leads to the event and without the cause the event is significantly less probable [Hit21]. The concept of finding general causes of events can be visualized as trace properties where we find all possible causes (say, smoking, genetics, pollution) for an event (say, lung cancer). However, when we want to argue about the cause of a specific effect we will need to consider all the probable causes simultaneously and find the most likely cause. This is referred to as token causation. This will require hyperproperties to express the search for such a cause.

Probabilistic conformance — System designs are used to guide their implementations. Conformance answers questions like "Was this a correct optimization?" or "Was this a safe refactoring?" [Way20]. Extending this idea to probabilistic systems, we want to ensure that no matter how the system has been implemented or changed, the

probabilistic distribution of the outcomes does not change. This is one way of determining if the design ideas have been successfully translated into the implementation or if the changes in our implementation still satisfy the requirements of the original system. Fig. 1.3 shows one such example.

Fairness — This is a growing concern among artificially intelligent systems. Since a lot of the societal decisions about us are being assisted by pre-trained AI systems, we want to ensure the system itself is fair to the whole population. Fairness can be divided into *group fairness* and *individual fairness*. Group fairness checks if a group of minority population is less likely to have access to a service or opportunity specifically due to their minority status. Individual fairness refers to the fact that two similar individuals should be treated similarly and has been recognized as a hyperproperty [ADDN17].

Robotics path planning — Non-hyperlogics are helpful in synthesizing paths in grids. However, if we want to argue about optimality (a robot using a specific strategy is always able to reach a goal faster), performance (which strategy has less energy consumption), or robustness (the robot can always reach the goal under any disturbance), we need to use hyperproperties [WZBP19].

Distributed algorithms — We often use randomization to break symmetry in order to tackle impossibility results. Although one can reason about the expected performance of a randomized distributed algorithm by the traditional reward models, from a design perspective, it is desirable to determine and mitigate states from where convergence to the objective of the algorithm takes much longer than others. This would require the involvement of hyperproperties to argue about reachability from multiple states.

This list is by no means a complete view but provides an insight into the range of applications of probabilistic hyperproperties. This has motivated us to pursue research on logic to represent them more efficiently and techniques to verify them on discrete-time

systems.

1.5 Thesis statement

For any verification problem, the two most important choices are the structure of the model and the type of property considered. This dissertation focuses on one such specific combination of these choices:

- When considering real-world systems, the presence of noise and errors cannot be eradicated completely. This gives rise to additional complexities. Discrete-time Markovian models help in capturing these concepts while keeping the model simple enough for feasible computation. Hence, our choice of models was that of DTMCs and MDPs.
- We focus on quantitative hyperproperties for probabilistic systems. The high expressivity of the logic for this purpose (HyperPCTL and a fragment of HyperPLTL) makes it challenging to create efficient model checking algorithms. With the majority of its applications being in expressing security and privacy policies, these properties cannot be merely 'tested' to ensure correctness. Hence, we focus on an in-depth exploration of the complexity of these properties and attempt to devise different verification algorithms for fragments of these logics.

Given this context, this dissertation proposes and aims to defend the following statement:

> HyperPCTL can be used to concisely express quantitative system requirements as *probabilistic hyperproperties*. Despite their high complexity, we can utilize exhaustive and approximate approaches to model check specific fragments of this logic that can express a range of prominent applications.

1.6 Contributions

To validate my thesis statement, I have conducted research on the following three fronts:

- **Logic**: We have focused on proposing a generalized logic HyperPCTL (extended the idea of HyperPCTL proposed in [ÁB18]) to express probabilistic hyperproperties. This involved non-trivial extensions of the logic to allow the expression of properties on nondeterministic systems and incorporated connections to reward models. We also allow arguments about the truth of the property over schedulers which is used to resolve non-determinism in such systems.

- **Algorithm**: We explore the complexity involved in model checking the general logic. We have proved that an exhaustive model checking solution to this problem is accurate but undecidable, in general, and becomes decidable if we limit the type of schedulers to a memoryless, non-probabilistic type. We prove that this restriction makes the problem NP-complete for an existential scheduler quantifier (co-NP complete for a universal quantifier). Based on these results, we have explored different approaches to handle the problem:

  - Exhaustive: We devised an algorithm to encode the model checking problem into an SMT-solving problem. It involves the automatic generation of Z3-based constraints combining the information of both the model and the hyperproperty.
  - Statistical: We focused on an interesting fragment of HyperPLTL and devised an approximate sound but incomplete method that scales efficiently on models with large state space.
  - Fragment specific: We focused on specific fragments of the logic in terms of a number of schedulers and initial states to device algorithms that synthesize randomized and/or memoryful schedulers based on scheduler combination or distribution transformers.

  Additionally, we propose an SMT-based algorithm to solve the parameter synthesis problem for a fragment of HyperPCTL for synthesizing missing transition probabilities given that the model should satisfy a given hyperproperty.

- **Implementation**: We have built prototypical implementations of all our algorithms and evaluated them on relevant case studies to focus on their strengths and weaknesses.

  - For our exhaustive algorithm, we have built a tool HyperProb. The tool takes a PRISM program and a probabilistic hyperproperty in HyperPCTL as a string, parses them to undergo automated generation of Z3 constraints, uses Z3 to solve the constraint set and returns the satisfaction result along with a witness (for satisfaction of existential quantifier) and a counterexample (for unsatisfaction of universal quantifier).

  - For our statistical approach we focused on an existing tool PLASMA to utilize its strength in handling scheduler synthesis and extending it to handle universally quantified probabilistic hyperproperties in HyperPLTL.

  - We have separately implemented prototypes of our fragment-specific algorithms and intend to incorporate them into HyperProb as additional features.

1.7 Organization

Publication	**Focus Area**	**Model**	**Logic**	**Associated with**
Parameter Synthesis for Probabilistic Hyperproperties	*Algorithm*, *Implementation*	DTMC	Non-nested HyperPCTL	LPAR'20
Probabilistic Hyperproperties with Nondeterminism	*Logic*	MDP	HyperPCTL	ATVA'20
Model Checking Hyperproperties for Markov Decision Processes	*Logic* *Algorithm*	MDP	HyperPCTL (1σ)	Information and Computation'22
Probabilistic Hyperproperties with Rewards	*Logic*, *Algorithm*, *Implementation*	MDP	HyperPCTL	NFM'22
HyperProb: A Model Checker for Probabilistic Hyperproperties	*Implementation*	MDP	HyperPCTL	FM'21
Efficient Probabilistic Model Checking for Relational Reachability	*Algorithm*, *Implementation*	MDP	HyperPCTL ($1\sigma 1s$), ($1\sigma 2s$)	CSF'24 (Submitted)
Lightweight Verification of Hyperproperties	*Algorithm*, *Implementation*	MDP	HyperPLTL	ATVA'23

Table 1.1: Details of publications.

Table 1.1 elaborates on the publications associated with this dissertation. The rest of the chapters are organized as follows:

- In Chapter 2, we formally describe the preliminary concepts on which our current work is built. It describes the basics of the structure of our models as well as the logic.

- In Chapter 3, we describe our work [ÁBBD20a] on the synthesis of values for parameters in our models based on specifications they must satisfy, expressed as probabilistic hyperproperties.

- In Chapter 4, we describe our work [ÁBBD20b] to extend its expressibility to include non-deterministic aspect of models by allowing quantification over schedulers and provide undecidability results for the model checking problem of the general logic.

- In Chapter 5, we describe our work [DÁBB22] that focuses on the decidable fragment of HyperPCTL along with proofs, and our SMT-based model checking algorithm.

- In Chapter 6, we describe our work of extending the expression of HyperPCTL to reason over reward models [DWÁ+22].

- In Chapter 7, we describe the working of the model checker HyperProb [DABB21] that involves all of the above extensions of HyperPCTL. It elaborates on the inner structure and execution details of the tool along with detailed evaluation.

- In Chapter 8, we describe our scalable model checking algorithms for two specific fragments of HyperPCTL. We synthesize randomized schedulers using the convex addition of schedulers and memoryful, deterministic schedulers using distribution transformation-based search.

- In Chapter 9, we describe our scalable model checking algorithm [DSB+23] for a specific fragment of HyperPLTL by extending PLASMA along with thorough evaluation.

- In Chapter 10, we describe related works that have either formed the basis of our work or motivated our research.
- In Chapter 11 we provide concluding discussions on extensions and applications of this line of work.

Chapter 2

Preliminaries

This chapter formally defines the frameworks we use to describe our models and specifications in the following chapters. We define Markov chains and their variations which are used to model our systems. We further provide the syntax and semantics of the logic HyperPCTL which we extend in the following chapters. Commonly used in the rest of the chapters, we use $\mathbb{R}$, $\mathbb{Q}$, and $\mathbb{N}$ to denote real, rational, and natural (including zero) numbers, respectively. We use $\underline{n} = \{1, \ldots, n\}$ for $n \in \mathbb{N}$.

2.1 Discrete-Time Markov models

Definition 2.1.1. A *discrete-time Markov chain (DTMC)* is a tuple $\mathcal{M}{=}(\mathcal{S}, \mathbb{P}, \mathsf{AP}, L)$ with the following components:

- $\mathcal{S}$ is a non-empty finite set of *states*;
- $\mathbb{P} : \mathcal{S} \times \mathcal{S} \to [0, 1]$ is a *transition probability function* with $\sum_{s' \in \mathcal{S}} \mathbb{P}(s, s') = 1$, for all $s \in \mathcal{S}$;
- AP is a finite set of *atomic propositions*, and
- $L : \mathcal{S} \to 2^{\mathsf{AP}}$ is a *labelling function.* ■

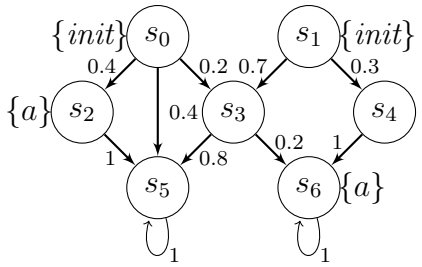


Figure 2.1: Example DTMC.

Fig 2.1 shows a simple DTMC. An (*infinite*) *path of* $\mathcal{M}$ is an infinite sequence $\pi = s_0 s_1 s_2 \ldots \in \mathcal{S}^\omega$ of states with $\mathbb{P}(s_i, s_{i+1}) > 0$, for all $i \geq 0$; we write $\pi[i]$ for s_i. Let $Paths^s(\mathcal{M})$ denote the set of all (infinite) paths of $\mathcal{M}$ starting in s, and $Paths^s_{fin}(\mathcal{M})$ denote the set of all non-empty finite prefixes of paths from $Paths^s(\mathcal{M})$, which we call *finite paths*. For a finite path $\pi = s_0 \ldots s_k \in Paths^{s_0}_{fin}(\mathcal{M})$, $k \geq 0$, we define $|\pi| = k$. We will also use the notations $Paths(\mathcal{M}) = \cup_{s \in \mathcal{S}} Paths^s(\mathcal{M})$ and $Paths_{fin}(\mathcal{M}) = \cup_{s \in \mathcal{S}} Paths^s_{fin}(\mathcal{M})$. A state $t \in \mathcal{S}$ is *reachable* from a state $s \in \mathcal{S}$ in $\mathcal{M}$ if there exists a finite path in $Paths^s_{fin}(\mathcal{M})$ with last state t; we use $Paths^{s,T}_{fin}(\mathcal{M})$ to denote the set of all finite paths from $Paths^s_{fin}(\mathcal{M})$ with last state in $T \subseteq S$. A state $s \in \mathcal{S}$ is *absorbing* if $\mathbb{P}(s, s) = 1$.

The *cylinder set* $Cyl^{\mathcal{M}}(\pi)$ of a finite path $\pi \in Paths^s_{fin}(\mathcal{M})$ is the set of all infinite paths of $\mathcal{M}$ with prefix π. The *probability space for* $\mathcal{M}$ *and state* $s \in \mathcal{S}$ is $(Paths^s(\mathcal{M}), \{\cup_{\pi \in R} Cyl^{\mathcal{M}}(\pi) \mid R \subseteq Paths^s_{fin}(\mathcal{M})\}, Pr^{\mathcal{M}}_s)$, where the *probability* of the cylinder set of $\pi \in Paths^s_{fin}(\mathcal{M})$ is $Pr^{\mathcal{M}}_s(Cyl^{\mathcal{M}}(\pi)) = \Pi_{i=1}^{|\pi|} \mathbb{P}(\pi[i-1], \pi[i])$.

Note that the cylinder sets of two finite paths starting in the same state are either disjoint or one is contained in the other. According to the definition of the probability spaces, the total probability for a set of cylinder sets defined by the finite paths $R \subseteq Paths^s_{fin}(\mathcal{M})$ is $Pr^{\mathcal{M}}(R) = \sum_{\pi \in R'} Pr^{\mathcal{M}}_s(\pi)$ with $R' = \{\pi \in R \mid no\ \pi' \in R \setminus \{\pi\}\ is\ a\ prefix\ of\ \pi\}$. To improve readability, we sometimes omit the DTMC index $\mathcal{M}$ in the notations when it is clear from the context.

Definition 2.1.2. The *parallel composition* of two DTMCs $\mathcal{M}_i = (\mathcal{S}_i, \mathbb{P}_i, \mathsf{AP}_i, L_i)$, $i = 1, 2$, is the DTMC $\mathcal{M}_1 \times \mathcal{M}_2 = (\mathcal{S}, \mathbb{P}, \mathsf{AP}, L)$ with the following components:

- $\mathcal{S} = \mathcal{S}_1 \times \mathcal{S}_2$;
- $\mathbb{P} : \mathcal{S} \times \mathcal{S} \to [0,1]$ with $\mathbb{P}((s_1, s_2), (s_1', s_2')) = \mathbb{P}_1(s_1, s_1') \cdot \mathbb{P}_2(s_2, s_2')$, for all states $(s_1, s_2), (s_1', s_2') \in \mathcal{S}$;
- $\mathsf{AP} = \mathsf{AP}_1 \cup \mathsf{AP}_2$, and
- $L : \mathcal{S} \to 2^{\mathsf{AP}}$ with $L((s_1, s_2)) = L_1(s_1) \cup L_2(s_2)$. ■

Definition 2.1.3. A *Markov decision process* (*MDP*) is a tuple $\mathcal{M} = (\mathcal{S}, Act, \mathbb{P}, \mathsf{AP}, L)$ with the following components:

- $\mathcal{S}$ is a non-empty finite set of *states*;
- Act is a non-empty finite set of *actions*;
- $\mathbb{P} : \mathcal{S} \times Act \times \mathcal{S} \to [0,1]$ is a *transition probability function* such that for all $s \in \mathcal{S}$ the *set of enabled actions* in s $Act(s) = \{\alpha \in Act \mid \sum_{s' \in \mathcal{S}} \mathbb{P}(s, \alpha, s') = 1\}$ is not empty and $\sum_{s' \in \mathcal{S}} \mathbb{P}(s, \alpha, s') = 0$ for all $\alpha \in Act \setminus Act(s)$;
- AP is a finite set of *atomic propositions*, and
- $L : \mathcal{S} \to 2^{\mathsf{AP}}$ is a *labeling function*. ■

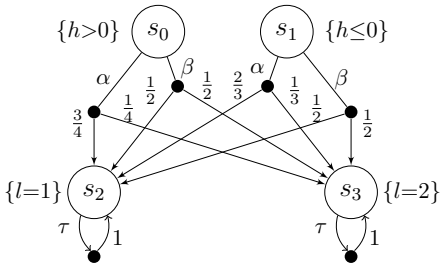


Figure 2.2: MDP showing the actions and probabilistic distributions.

Fig. 2.2 shows a simple MDP. Schedulers can be used to eliminate the non-determinism in MDPs, inducing DTMCs with well-defined probability spaces.

Definition 2.1.4. A *scheduler* for an MDP $\mathcal{M} = (\mathcal{S}, Act, \mathbb{P}, \mathsf{AP}, L)$ is a tuple $\sigma = (Q, act, mode, init)$, where

- Q is a countable set of *modes*;
- $act : Q \times \mathcal{S} \times Act \to [0,1]$ is a function for which $\sum_{\alpha \in Act(s)} act(q, s, \alpha) = 1$ and $\sum_{\alpha \in Act \setminus Act(s)} act(q, s, \alpha) = 0$ for all $s \in \mathcal{S}$ and $q \in Q$;

- $mode : Q \times \mathcal{S} \to Q$ is a *mode transition* function, and
- $init : \mathcal{S} \to Q$ is a function selecting a starting mode for each state of $\mathcal{M}$. ■

Let $\Sigma^{\mathcal{M}}$ denote the set of all schedulers for the MDP $\mathcal{M}$. A scheduler is *finite-memory* if Q is finite, *memoryless* if $|Q| = 1$, and *non-probabilistic* if $act(q, s, \alpha) \in \{0, 1\}$ for all $q \in Q$, $s \in \mathcal{S}$ and $\alpha \in Act$.

Definition 2.1.5. Assume an MDP $\mathcal{M} = (\mathcal{S}, Act, \mathbb{P}, \mathsf{AP}, L)$ and a scheduler $\sigma = (Q, act, mode, init) \in \Sigma^{\mathcal{M}}$ for $\mathcal{M}$. The *DTMC induced by* $\mathcal{M}$ *and* σ is defined as $\mathcal{M}^\sigma = (\mathcal{S}^\sigma, \mathbb{P}^\sigma, \mathsf{AP}, L^\sigma)$ with $\mathcal{S}^\sigma = Q \times \mathcal{S}$,

$$\mathbb{P}^\sigma((q, s), (q', s')) = \begin{cases} \sum_{\alpha \in Act(s)} act(q, s, \alpha) \cdot \mathbb{P}(s, \alpha, s') & \text{if } q' = mode(q, s) \\ 0 & \text{otherwise} \end{cases}$$

and $L^\sigma(q, s) = L(s)$ for all $s, s' \in \mathcal{S}$ and all $q, q' \in Q$. ■

A state s' is *reachable* from $s \in \mathcal{S}$ in MDP $\mathcal{M}$ is there exists a scheduler σ for $\mathcal{M}$ such that s' is reachable from s in $\mathcal{M}^\sigma$. A state $s \in \mathcal{S}$ is *absorbing* in $\mathcal{M}$ if s is absorbing in $\mathcal{M}^\sigma$ for all schedulers σ for $\mathcal{M}$. We sometimes omit the MDP index $\mathcal{M}$ in the notations when it is clear from the context.

2.2 Discrete-Time Markov Models with Rewards

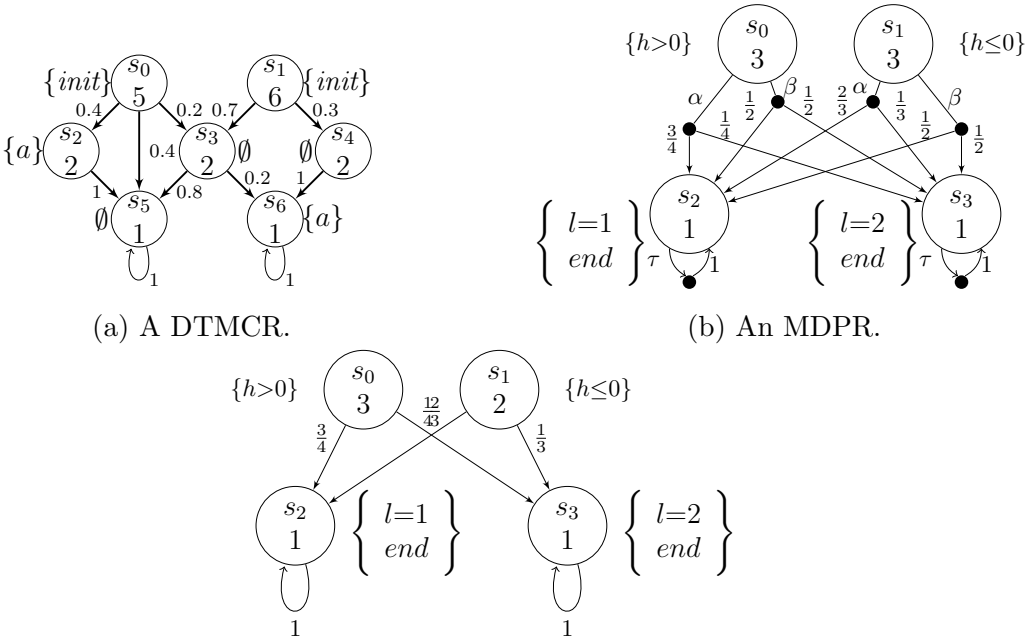
For any domain D and any $v = (v_0, \ldots, v_{n-1}) \in D^n$, we define $v[i] = v_i$ for $i \in \{0, \ldots, n-1\}$. The concepts below have been adapted from [BK08] and extended to work for hyperlogics.

When defining costs or rewards for Markov models, we can assign rewards to states or transitions. In this work we limit to the assignment of *non-negative* rewards to *states* and support multi-dimensional reward *vectors*.

Definition 2.2.1. A *Discrete Time Markov Chain with (k-ary) rewards (DTMCR)* is a tuple $\mathcal{M} = (\mathcal{S}, P, \mathsf{AP}, L, rew)$ with

- a non-empty set of states $\mathcal{S}$,

- a transition function $P : \mathcal{S} \times \mathcal{S} \to [0, 1] \subseteq \mathbb{R}$ with $\sum_{s' \in \mathcal{S}} P(s, s') = 1$ for all $s \in \mathcal{S}$,
- a finite set of atomic propositions AP,
- a labeling function $L : \mathcal{S} \to 2^{\mathsf{AP}}$ and
- a reward function *rew*: $\mathcal{S} \to \mathbb{R}^k_{\geq 0}$.



(a) A DTMCR.

(b) An MDPR.

(c) An induced DTMCR.

Figure 2.3: Example Markov models with rewards.

Fig. 2.3a shows an example DTMCR with unary rewards. Assume a DTMCR $\mathcal{M} = (\mathcal{S}, P, \mathsf{AP}, L, \mathit{rew})$. An *infinite path* is a sequence of states $\pi = s_0 s_1 \ldots \in \mathcal{S}^\omega$ with $P(s_i, s_{i+1}) > 0$ for all $i \in \mathbb{N}$. A non-empty prefix of an infinite path is a *finite path* $\pi = s_0 \ldots s_{n-1} \in \mathcal{S}^+$ of length $|\pi| = n \in \mathbb{N} \setminus \{0\}$. Let $\mathit{Paths}^s(\mathcal{M})$ ($\mathit{Paths}^s_{\mathit{fin}}(\mathcal{M})$) be the set of all infinite (finite) paths starting in $s \in \mathcal{S}$. A state $t \in \mathcal{S}$ is *reachable* from $s \in \mathcal{S}$ if there exists a path in $\mathit{Paths}^s_{\mathit{fin}}(\mathcal{M})$ ending in t. A state $s \in \mathcal{S}$ is *absorbing* iff $P(s, s) = 1$.

For a finite path $\pi \in \mathit{Paths}^s_{\mathit{fin}}(\mathcal{M})$, we define its *cylinder set* $\mathit{Cyl}^{\mathcal{M}}(\pi)$ as the set of all infinite paths with π as a prefix. The probability of the cylinder set of $\pi \in \mathit{Paths}^s_{\mathit{fin}}(\mathcal{M})$ is defined as $\mathit{Pr}^{\mathcal{M}}_s(\mathit{Cyl}^{\mathcal{M}}(\pi)) = \Pi_{i=0}^{|\pi|-1} P(s_i, s_{i+1})$. For sets $R \subseteq \mathit{Paths}^s_{\mathit{fin}}(\mathcal{M})$ we have $\mathit{Pr}^{\mathcal{M}}_s(R) = \sum_{\pi \in R'} \mathit{Pr}^{\mathcal{M}}_s(\pi)$, where R' contains all finite paths from R that have no extensions in R.

These notions induce for each $s \in \mathcal{S}$ the *probability space*,

$$\Big(Paths^s(\mathcal{M}), \big\{ \bigcup_{\pi \in R} Cyl^{\mathcal{M}}(\pi) \mid R \subseteq Paths^s_{fin}(\mathcal{M}) \big\}, Pr^{\mathcal{M}}_s \Big).$$

Note that the cylinder sets of two finite paths starting in the same state are either disjoint or one is contained in the other.

For a reward function $rew\colon \mathcal{S} \to \mathbb{R}^k_{\geq 0}$ and $i \in \{0, \ldots, k-1\}$ we define $rew_i\colon \mathcal{S} \to \mathbb{R}_{\geq 0}$ to assign the *ith state reward* $rew_i(s) = rew(s)[i]$ to all $s \in \mathcal{S}$. The *ith cumulative reward* for a finite path, $\pi = s_0 s_1 \ldots s_{n-1}$ is defined as $rew_i(\pi) = \sum_{j=0}^{n-1} rew_i(s_j)$. Note that non-negative rewards assure monotonic increase of cumulative rewards with path extensions.

To argue about simultaneous runs across two DTMCRs, we define their parallel composition.

Definition 2.2.2. Assume two DTMCRs $\mathcal{M}_i = (\mathcal{S}_i, P_i, \mathsf{AP}_i, L_i, rew_i)$ with k_i-ary rewards, $i \in \{1, 2\}$. We define the *parallel composition* $\mathcal{M}_1 \times \mathcal{M}_2 = (\mathcal{S}_1 \times \mathcal{S}_2, P, \mathsf{AP}_1 \cup \mathsf{AP}_2, L, rew)$ with (k_1+k_2)-ary rewards, such that for all (s_1, s_2), $(s'_1, s'_2) \in \mathcal{S} \times \mathcal{S}$:

- $P\big((s_1, s_2),(s'_1, s'_2)\big) = P_1(s_1, s'_1) \cdot P_2(s_2, s'_2)$,
- $L((s_1, s_2)) = L_1(s_1) \cup L_2(s_2)$ and
- $rew((s_1, s_2)) = (rew_1(s_1), rew_2(s_2))$.

Next, we extend the probabilistic nature of DTMCRs with non-determinism.

Definition 2.2.3. A *Markov Decision Process with k-ary rewards (MDPR)* is a tuple $\mathcal{M} = (\mathcal{S}, Act, P, \mathsf{AP}, L, rew)$ with

- a non-empty set of states $\mathcal{S}$,
- a non-empty finite set of actions Act,
- a transition function $P : \mathcal{S} \times Act \times S \to [0, 1] \subseteq \mathbb{R}$ such that for each $s \in \mathcal{S}$ we have $\sum_{s' \in S} P(s, \alpha, s') \in \{0, 1\}$. For all $\alpha \in Act$, there is at least one action that can be chosen in each state, such that $\alpha \in Act(s) = \{\alpha \in Act | \sum_{s' \in S} P(s, \alpha, s') = 1\}$ and for $\alpha \in Act \setminus Act(s), \sum_{s' \in S} P(s, \alpha, s') = 0$,
- a finite set of atomic propositions AP,
- a labelling function $L : \mathcal{S} \to 2^{\mathsf{AP}}$, and

- a reward function $rew : \mathcal{S} \to \mathbb{R}^k_{\geq 0}$.

Fig.2.3b shows an example MDPR. In each state, for the next execution step, any of the enabled actions can be chosen non-deterministically. *Schedulers* are used to eliminate this non-determinism.

Definition 2.2.4. A *scheduler* for an MDPR $\mathcal{M} = (\mathcal{S}, Act, P, \mathsf{AP}, L, rew)$ is a tuple $\sigma = (Q, act, mode, init)$ with

- a countable set of modes Q,
- a function $act : Q \times S \times Act \to [0,1] \subseteq \mathbb{R}$ such that for every $s \in S$ and $q \in Q$,
  $\sum_{\alpha \in Act(s)} act(q,s,\alpha) = 1 \quad \text{and} \sum_{\alpha \in Act \setminus Act(s)} act(q,s,\alpha) = 0$,
- a mode transition function $mode : Q \times S \to Q$, and
- $init : S \to Q$ assigning to each state of $\mathcal{M}$ a starting mode.

Let $\Sigma^{\mathcal{M}}$ be the set of all schedulers for $\mathcal{M}$. A scheduler is *finite-memory* if Q is finite, *memoryless* if $|Q| = 1$, and *non-probabilistic* if $act(q,s,\alpha) \in \{0,1\}$ for all $q \in Q$, $s \in S$ and $\alpha \in Act$.

Definition 2.2.5. Assume an MDPR $\mathcal{M} = (\mathcal{S}, Act, P, \mathsf{AP}, L, rew)$ with k-ary rewards and a scheduler $\sigma = (Q, act, mode, init)$ for $\mathcal{M}$. Then $\mathcal{M}$ and σ *induce* the DTMCR with k-ary rewards $\mathcal{M}^\sigma = (S^\sigma, P^\sigma, \mathsf{AP}, L^\sigma, rew^\sigma)$, where $S^\sigma = Q \times S$,

$$P^\sigma((q,s),(q',s')) = \begin{cases} \sum_{\alpha \in Act(s)} act(q,s,\alpha) \cdot P(s,\alpha,s') & \text{if } q' = mode(q,s) \\ 0 & \text{if } q' \neq mode(q,s) \end{cases},$$

with $L^\sigma(q,s) = L(s)$ and $rew^\sigma(q,s) = rew(s)$, for all $q \in Q$ and $s \in S$.

If σ is memoryless, we sometimes omit its mode and write (s) instead of (q,s). For the MDPR in Fig. 2.3b and a scheduler that chooses action α in states s_0, s_1 and action τ in states s_2, s_3, the induced DTMCR is shown in Fig. 2.3c.

Different executions in several models can be seen as executions in the composition of the models. To simplify notation, in this dissertation we restrict ourselves to comparing executions in the same model, leading to the notion of *self-composition*.

Definition 2.2.6. Assume an MDPR $\mathcal{M} = (\mathcal{S}, Act, P, \mathsf{AP}, L, rew)$ and a sequence $\boldsymbol{\sigma} = (\sigma_0, \ldots, \sigma_{n-1}) \in (\Sigma^{\mathcal{M}})^n$ of schedulers for $\mathcal{M}$. For $i \in \{0, \ldots, n-1\}$, let $\mathcal{M}_i = (S, Act, P, \mathsf{AP}_i, L_i, rew)$ with $\mathsf{AP}_i = \{a_i \mid a \in \mathsf{AP}\}$, and $L_i : \mathcal{S} \to 2^{\mathsf{AP}_i}$ with $L_i(s) = \{a_i \mid a \in L(s)\}$. We define the *n-ary self composition* of $\mathcal{M}$ under $\boldsymbol{\sigma}$ as the DTMCR $\mathcal{M}^{\boldsymbol{\sigma}} = (S^{\boldsymbol{\sigma}}, P^{\boldsymbol{\sigma}}, \mathsf{AP}^{\boldsymbol{\sigma}}, L^{\boldsymbol{\sigma}}, rew^{\boldsymbol{\sigma}}) = \mathcal{M}_0^{\sigma_0} \times \ldots \times \mathcal{M}_{n-1}^{\sigma_{n-1}}$.

In the above definition, $\mathcal{M}_i^{\sigma_i}$ is the DTMCR induced by $\mathcal{M}_i$ and σ_i. Note that the reward of a state $\boldsymbol{s} = ((q_0, s_0), \ldots, (q_{n-1}, s_{n-1})) \in \mathcal{S}^{\boldsymbol{\sigma}}$ in the n-ary self-composition $\mathcal{M}^{\boldsymbol{\sigma}}$ is the sequence $rew^{\boldsymbol{\sigma}}(\boldsymbol{s}) = (rew(s_0), \ldots, rew(s_{n-1}))$, i.e. the ith state reward in the jth execution is $rew^{\boldsymbol{\sigma}}_{j,i}(\boldsymbol{s}) = rew_i(s_j)$. For a finite path π in $\mathcal{M}^{\boldsymbol{\sigma}}$, we denote its cumulative ith reward in the jth execution as $rew_{j,i}(\pi) = \sum_{k=0}^{|\pi|-1} rew_{j,i}(\pi[k])$.

2.3 Probabilistic Hyperproperties

HyperPCTL [ÁB18] was the first logic proposed to express probabilistic hyperproperties. It generalized PCTL by allowing explicit quantification over initial states, and hence, multiple computation trees. This laid out the syntax, semantics, and main applications which we have extended in this dissertation.

2.3.1 HyperPCTL Syntax

HyperPCTL (quantified) state formulas φ^q are inductively defined as follows:

$$
\begin{array}{lllll}
\textit{quantified formula} & \varphi^q & ::= & \forall \hat{s}.\varphi^q \mid \exists \hat{s}.\varphi^q \mid \varphi^{nq} \\
\textit{non-quantified formula} & \varphi^{nq} & ::= & \texttt{true} \mid a_{\hat{s}} \mid \varphi^{nq} \wedge \varphi^{nq} \mid \neg\varphi^{nq} \mid \varphi^{pr}\ \varphi^{pr} \\
\textit{probability expression} & \varphi^{pr} & ::= & \mathbb{P}(\varphi^{path}) \mid f(\varphi^{pr}_1, \ldots, \varphi^{pr}_k) \\
\textit{path formula} & \varphi^{path} & ::= & \bigcirc \varphi^{nq} \mid \varphi^{nq}\, \mathcal{U}\, \varphi^{nq} \mid \varphi^{nq}\, \mathcal{U}^{[k_1,k_2]}\, \varphi^{nq}
\end{array}
$$

where $\hat{s}$ is a *state variable* from an infinite set $\hat{\mathcal{S}}$, φ^{nq} is a quantifier-free state formula, $a \in \mathsf{AP}$ is an atomic proposition, φ^{pr} is a *probability expression*, $\sim \in \{<, \leq, =, >, \geq\}$, $f : [0,1]^k \to \mathbb{R}$ are k-ary elementary functions to express arithmetic operations (binary addition, binary subtraction, binary multiplication) over probabilities where constants are viewed as 0-ary

functions, and φ^{path} is a *path formula*, such that $k_1 \leq k_2 \in \mathbb{N}_{\geq 0}$. The probability operator $\mathbb{P}$ allows the usage of probabilities in arithmetic constraints and relations.

A HyperPCTL construct φ (probability expression φ^{pr}, state formula φ^{q}, φ^{nq} or path formula φ^{path}) is *well-formed* if each occurrence of any $a_{\hat{s}}$ with $a \in \mathsf{AP}$ and $\hat{s} \in \hat{\mathcal{S}}$ is in the scope of a *state quantifier* for $\hat{s}$.

HyperPCTL formulas are well-formed HyperPCTL state formulas, where we additionally allow standard syntactic sugar like $\texttt{false} = \neg\texttt{true}$, $\varphi_1 \vee \varphi_2 = \neg(\neg\varphi_1 \wedge \neg\varphi_2)$, $\Diamond\varphi = \texttt{true}\,\mathcal{U}\,\varphi$, and $\mathbb{P}(\Box\varphi) = 1 - \mathbb{P}(\Diamond\neg\varphi)$.

2.3.2 HyperPCTL Semantics

HyperPCTL state formulas are evaluated in the context of self-composition of a DTMC as described above. We use () to denote the empty sequence (of any type) and $\circ$ for concatenation. Intuitively, these sequences store instantiations for state variables. The satisfaction of a HyperPCTL quantified formula by $\mathcal{M}$ is defined by

$$\mathcal{M} \models \varphi \qquad \textit{iff} \qquad \mathcal{M}, () \models \varphi \ .$$

The formula evaluates the logical value of the quantified state and path subformulas in the context of a DTMC and an n-tuple state combination from the composed DTMC. The semantics evaluates HyperPCTL formulas by structural recursion. Let $\mathbb{Q}$ denote a quantifier from $\{\forall, \exists\}$. For instantiating a state quantifier $\mathbb{Q}\hat{s}$ by a state s, we concatenate s_{n+1} at the end of the existing state tuple, where n is the number of quantifiers already processed. We also replace each $a_{\hat{s}}$ in the scope of the given quantifier by $a_{s_{n+1}}$, resulting in a formula that we denote by $\varphi[\hat{s} \rightsquigarrow (n+1)]$.

Formally, the semantics judgment rules are as follows:

$$\begin{aligned}
&\mathcal{M}, s \models \texttt{true}, \\
&\mathcal{M}, s \models a_i \qquad\quad \textit{iff} \quad a_i \in L(s_i), \\
&\mathcal{M}, s \models \varphi_1 \wedge \varphi_2 \quad \textit{iff} \quad \mathcal{M}, s \models \varphi_1 \ \textit{and}\ \mathcal{M}, s \models \varphi_2,
\end{aligned}$$

$$
\begin{aligned}
\mathcal{M}, s \models \neg\varphi \quad & \textit{iff} \quad \mathcal{M}, s \not\models \varphi, \\
\mathcal{M}, s \models \forall \hat{s}.\varphi \quad & \textit{iff} \quad \forall s_{n+1} \in \mathcal{S}.\ \mathcal{M}, s \circ s_{n+1} \models \varphi[\hat{s} \rightsquigarrow s_{n+1}] \\
\mathcal{M}, s \models \exists \hat{s}.\varphi \quad & \textit{iff} \quad \exists s_{n+1} \in \mathcal{S}.\ \mathcal{M}, s \circ s_{n+1} \models \varphi[\hat{s} \rightsquigarrow s_{n+1}] \\
\mathcal{M}, s \models \varphi_1^{pr} \sim \varphi_2^{pr} \quad & \textit{iff} \quad [\![\varphi_1^{pr}]\!]_{\mathcal{M},s} \sim [\![\varphi_2^{pr}]\!]_{\mathcal{M},s} \\
[\![\mathbb{P}(\varphi_{path})]\!]_{\mathcal{M},s} \quad & = \quad Pr^{\mathcal{M}}\big(\{\pi \in Paths^s(\mathcal{M}) \mid \mathcal{M}, \pi \models \varphi_{path}\}\big) \\
[\![f(\varphi_1^{pr}, \ldots \varphi_k^{pr})]\!]_{\mathcal{M},s} \quad & = \quad f\big([\![\varphi_1^{pr}]\!]_{\mathcal{M},s} \ldots,\ [\![\varphi_k^{pr}]\!]_{\mathcal{M},s}\big)
\end{aligned}
$$

where $\mathcal{M}$ is an DTMC; $n \in \mathbb{N}_{\geq 0}$ is non-negative integer; s is a state of $\mathcal{M}$; $a \in \mathsf{AP}$ is an atomic proposition and $i \in \{1, \ldots, n\}$; $\varphi, \varphi_1, \varphi_2$ are HyperPCTL state formulas; $\varphi_1^{pr} \cdots \varphi_k^{pr}$ are probability expressions, and φ_{path} is a HyperPCTL path formula whose satisfaction relation is as follows:

$$
\begin{aligned}
\mathcal{M}, \pi \models \bigcirc \varphi \quad & \textit{iff} \ \ \mathcal{M}, \pi[1] \models \varphi \\
\mathcal{M}, \pi \models \varphi_1 \mathcal{U} \varphi_2 \quad & \textit{iff} \ \ \exists j \geq 0.\Big(\mathcal{M}, \pi[j] \models \varphi_2 \wedge \forall i \in [0, j).\ \mathcal{M}, \pi[i] \models \varphi_1\Big) \\
\mathcal{M}, \pi \models \varphi_1 \mathcal{U}^{[k_1,k_2]} \varphi_2 \ \ & \textit{iff} \ \ \exists j \in [k_1, k_2].\Big(\mathcal{M}, \pi[j] \models \varphi_2 \wedge \forall i \in [0, j).\mathcal{M}, \pi[i] \models \varphi_1\Big)
\end{aligned}
$$

where $\pi = s_0 s_1 \cdots$ is a path of $\mathcal{M}$; and $k_1 \leq k_2 \in \mathbb{N}_{\geq 0}$.

Example — Consider the example from Fig. 2.1 and the HyperPCTL property below.

$$
\varphi = \forall s.\ \forall s'.(init_s \wedge init_{s'}) \rightarrow \big(Pr(\Diamond a_s) = Pr(\Diamond a_{s'})\big) \tag{2.1}
$$

The formula in 2.1 is satisfied by $\mathcal{M}$ if for all pairs of initial states (labeled by the atomic proposition *init*) the probability to satisfy a is the same, i.e., for each $(s_i, s_j) \in \mathcal{S}^2$ with *init* $\in L(s_i)$ and *init* $\in L(s_j)$ it holds that $\mathcal{M}, (s_i, s_j) \models Pr(a_1) = Pr(a_2)$. The probability of reaching a from s_0 is $0.4 + (0.2 \times 0.2) = 0.44$. Moreover, the probability of reaching a from s_1 is $0.3 + (0.7 \times 0.2) = 0.44$. Hence, we have $\mathcal{M} \models \varphi$.

The model checking problem of probabilistic hyperproperties over DTMCs was decidable and exponential in the number of quantifiers [ÁB18]. They also proposed a symbolic model checking algorithm that recursively evaluates the truth of sub-formulas φ_{sub} and labels the root node of the computation tree with φ_{sub} if the sub-formula is true. On termination

of this labeling algorithm, if the starting states (per the quantifiers) contain the required satisfaction labels corresponding to the hyperproperty, the algorithm is true else false.

This dissertation focuses on extending HyperPCTL [ÁB18], both its theory base in terms of logic and complexity results, and the model checking algorithm for model checking probabilistic hyperproperties to allow for non-determinism, rewards, and fragment-specific solutions.

Chapter 3

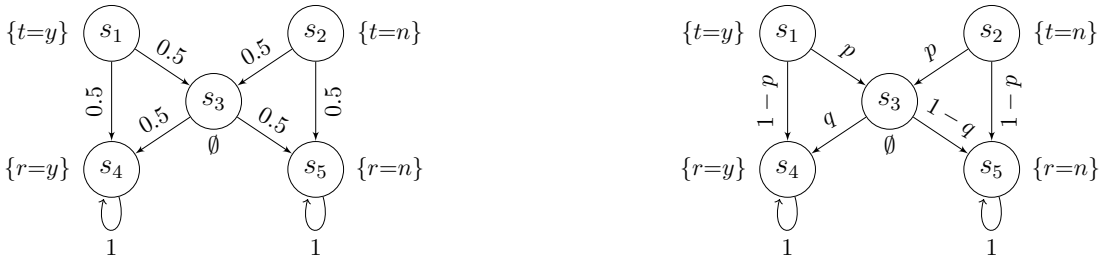
Parameter Synthesis for Probabilistic Hyperproperties

In this chapter, we discuss the problem of parameter synthesis for our probabilistic hyperproperties. Given a model with unknown parameters and a HyperPCTL specification that we want to be satisfied in the model, we want to study the complexity of the problem to synthesize values for the unknown parameters and propose a symbolic constraint-based algorithm for the same.

3.1 Introduction

We first motivate the problem through a simple example. Consider the concept of *differential privacy* [DR14, DMNS16], that is, a commitment by a data holder to a data subject that he/she will not be affected by allowing his/her data to be used in any study or analysis. More formally, let ϵ be a positive real number and $\mathcal{A}$ be a randomized algorithm that makes a query to an input database and produces an output. Algorithm $\mathcal{A}$ is called ϵ-*differentially private*, if for all databases D_1 and D_2 that differ on a single element, and all subsets S of possible outputs of $\mathcal{A}$, we have:

$$Pr[\mathcal{A}(D_1) \in S] \leq e^{\epsilon} \cdot Pr[\mathcal{A}(D_2) \in S] \tag{3.1}$$



(a) DTMC

(b) Parametric DTMC

Figure 3.1: The randomized response protocol.

One way to guarantee differential privacy is by introducing *randomized response* to create noise and provide plausible deniability. For example, let A be an embarrassing or illegal activity. In a social study, each participant is faced with the query, "Have you engaged in activity A in the past week?" and is instructed to respond by the following protocol: (1) flip a fair coin, (2) if tail, then answer truthfully, and (3) if head, then flip the coin again and respond "Yes" if head and "No" if tail. Thus, there are no good or bad responses and an answer cannot be incriminating. The discrete-time Markov chain (DTMC) of this protocol conducted by a fair coin is shown in Fig. 3.1a, where $t = y$ (respectively, $t = n$) denotes that the truth is 'Yes' (respectively, 'No') and $r = y$ (respectively, $r = n$) denotes the fact that the response is 'Yes' (respectively, 'No'). It is straightforward to show that this protocol is $(\ln 3)$-differentially private.

Now, let us imagine that we intend to change this protocol in order to make it $(\ln 2)$-differentially private. To this end, one can first transform the DTMC shown in Fig. 3.1a into a *parametric* DTMC (see Fig. 3.1b) that allows two different types of coins to be used during the protocol, hence, parameters p and q. Then, we solve the *parameter synthesis problem* by finding a value for parameters p and q that result in an $(\ln 2)$-differentially private protocol. Differential privacy is a *probabilistic hyperproperty*, as it prescribes a probability relation between a set of independent executions. Although the parameter synthesis problem has been extensively studied in the context of conventional properties, to our knowledge, it has not yet been solved in the context of hyperproperties.

Here, our goal is to solve the parameter synthesis problem for a fragment of the temporal logic HyperPCTL [ÁB18]. HyperPCTL lifts the well-known probabilistic temporal logic PCTL by allowing to express stochastic relations between computations starting in different initial states. The fragment studied here, called ReachHyperPCTL, is restricted to non-nested probability operators. For the above randomized response protocol the following ReachHyperPCTL formula states that whenever a computation starts in a state σ and another computation in σ' with a different truth value, the probabilities to get the same response satisfy the $(\ln 3)$-differential privacy condition:

$$\begin{aligned}\varphi_{\mathsf{dp}} = \forall\sigma.\forall\sigma'.\Big[\Big((t{=}n)_\sigma \wedge (t{=}y)_{\sigma'}\Big) \Rightarrow \Big(\mathbb{P}\Big(\Diamond(r{=}n)_\sigma\Big) \leq e^{\ln 3}\cdot\mathbb{P}\Big(\Diamond(r{=}n)_{\sigma'}\Big)\Big)\Big] \wedge \\ \Big[\Big((t{=}y)_\sigma \wedge (t{=}n)_{\sigma'}\Big) \Rightarrow \Big(\mathbb{P}\Big(\Diamond(r{=}y)_\sigma\Big) \leq e^{\ln 3}\cdot\mathbb{P}\Big(\Diamond(r{=}y)_{\sigma'}\Big)\Big)\Big] \quad (3.2)\end{aligned}$$

Given a parametric DTMC $\mathcal{D}$ and ReachHyperPCTL formula ψ, we solve the parameter synthesis problem for ReachHyperPCTL in two steps. In the first step, we compute for each possible instantiation of the quantified (initial) states an arithmetic formula over the model parameters that is true for exactly those parameter configurations that instantiate $\mathcal{D}$ to satisfy ψ. In a second step, we use those formulas to compute not only single solutions, but whole regions of satisfying parameter configurations: we decompose the configuration domain and identify smaller regions in which either all or none of the configurations lead to the satisfaction of ψ.

We illustrate the application of our technique by using four case studies. Our first example is differential privacy, as described above. The second example is *probabilistic noninterference* [III92], which establishes a connection between information theory and information flow by employing probabilities to address covert channels. The third case study is *probabilistic conformance*, where we want to find the parameter values of two systems (e.g., a model and an implementation) such that they conform with each other with respect to a specification. The last case study is the dining cryptographers problem [Cha88], where we show that the anonymity of the cryptographers is assured even when using a biased coin in the protocol.

3.1.1 The Logic ReachHyperPCTL

Here we consider *parametric* DTMCs and the problem to synthesize parameter configurations that satisfy a certain probabilistic hyperproperty. As this problem involves symbolic encodings of reachability probabilities and the truth values of hyperproperties, in order to provide an effective synthesis algorithm, we restrict ourselves to a fragment called ReachHyperPCTL, which excludes nested probability operators.

Syntax. ReachHyperPCTL is syntactically defined over a set AP of atomic propositions by the following abstract grammar:

$$\begin{array}{lcl} \psi & ::= & \forall\sigma.\psi \;\Big|\; \exists\sigma.\psi \;\Big|\; a_\sigma \;\Big|\; \psi \wedge \psi \;\Big|\; \neg\psi \;\Big|\; p \sim p \\ p & ::= & \mathbb{P}(\bigcirc \varphi) \;\Big|\; \mathbb{P}(\varphi \,\mathcal{U}\, \varphi) \;\Big|\; f(p,\ldots,p) \\ \varphi & ::= & a_\sigma \;\Big|\; \varphi \wedge \varphi \;\Big|\; \neg\varphi \end{array}$$

where $a \in \mathsf{AP}$ is an atomic proposition, $\sim \in \{<, \leq, =, \geq, >\}$, σ are *state variables* from a countably infinite set $\mathcal{V}$, and $f : [0,1]^k \to \mathbb{R}$ are k-ary elementary functions to express arithmetic operations over probabilities, where constants are viewed as 0-ary functions. We call φ and ψ *state formulas*, a_σ an *indexed atomic proposition* and p a *probability expression*. We denote by $\mathcal{F}$ the set of all ReachHyperPCTL state formulas and probability expressions (over AP). The difference to HyperPCTL is that temporal operator may be applied to Boolean combinations of atomic propositions only (operands φ instead of ψ). We use standard syntactic sugar $\texttt{false} = a_\sigma \wedge \neg a_\sigma$, $\texttt{true} = \neg\texttt{false}$, $\varphi_1 \vee \varphi_2 = \neg(\neg\varphi_1 \wedge \neg\varphi_2)$, $\Diamond\varphi = \texttt{true}\,\mathcal{U}\,\varphi$, $\Box\varphi = \neg\Diamond\neg\varphi$, etc. An occurrence of an indexed atomic proposition a_σ in a ReachHyperPCTL state formula ψ is *free* if it is not in the scope of a quantifier bounding σ and otherwise *bound*. ReachHyperPCTL *sentences* are ReachHyperPCTL state formulas in which all occurrences of all indexed atomic propositions are bound. ReachHyperPCTL (*quantified*) *formulas* are ReachHyperPCTL sentences. Each ReachHyperPCTL quantified formula can be transformed into an equivalent formula in prenex normal form $\mathbb{Q}_1\sigma_1 \ldots \mathbb{Q}_n\sigma_n.\psi$, where each $\mathbb{Q}_i \in \{\forall, \exists\}$

is a quantifier, σ_i is a state variable, and ψ is a quantifier-free ReachHyperPCTL formula. In the following, we assume all ReachHyperPCTL quantified formulas to be in prenex normal form.

Example. The formula

$$\forall \sigma_1 . \exists \sigma_2 . \Big(\mathbb{P}(\Diamond\, a_{\sigma_1}) = \mathbb{P}(\Diamond\, b_{\sigma_2}) \Big) \tag{3.3}$$

holds if for each state s_1, there exists another state s_2, such that the probability to finally reach a state labeled with a from s_1 equals the probability of reaching b from s_2.

Semantics. We present the semantics of ReachHyperPCTL based on the n-ary *self-composition* of a DTMC. We emphasize that it is possible to define the semantics in terms of the non-self-composed DTMC, but it will essentially result in a very similar setting, but more difficult to understand.

Definition 3.1.1. The *n-ary self-composition* of a PDTMC $\mathcal{M} = (S, V, \mathbf{P}, \mathsf{AP}, L)$ is the PDTMC $\mathcal{M}^n = (S^n, V, \mathbf{P}^n, \mathsf{AP}^n, L^n)$ with

- $S^n = S \times \ldots \times S$ is the n-ary Cartesian product of S,
- $\mathbf{P}^n\big(s, s'\big) = \Pi_{i \in \underline{n}} \mathbf{P}(s_i, s_i')$ for all $s = (s_1, \ldots, s_n) \in S^n$ and $s' = (s_1', \ldots, s_n') \in S^n$,
- $\mathsf{AP}^n = \cup_{i \in \underline{n}} \mathsf{AP}_i$, where $\mathsf{AP}_i = \{a_i \,|\, a \in \mathsf{AP}\}$ for $i \in \underline{n}$, and
- $L^n(s) = \cup_{i \in \underline{n}}\, L_i(s_i)$ for all $s = (s_1, \ldots, s_n) \in S^n$ with $L_i(s_i) = \{a_i \,|\, a \in L(s_i)\}$ for $i \in \underline{n}$. ■

The satisfaction relation for ReachHyperPCTL sentences by a DTMC $\mathcal{M} = (S, \mathbf{P}, \mathsf{AP}, L)$ is defined by:

$$\mathcal{M} \models \psi \qquad \textit{iff} \qquad \mathcal{M}, () \models \psi$$

where $()$ is the empty sequence of states. Thus, the satisfaction relation $\models$ defines the values of ReachHyperPCTL quantified, state, and path formulas in the context of a DTMC $\mathcal{M} = (S, \mathbf{P}, \mathsf{AP}, L)$ and an n-tuple $s = (s_1, \ldots, s_n) \in S^n$ of states (which is $()$ for $n = 0$). Intuitively, the state sequence s stores instantiations for quantified state variables. The

semantics evaluates ReachHyperPCTL formulas by structural recursion. Quantifiers are instantiated and the instantiated values for state variables are stored in the state sequence s. To maintain the connection between a state in this sequence and the state variable which it instantiates, we introduce the auxiliary syntax a_i with $a \in \mathsf{AP}$ and $i \in \mathbb{N}_{>0}$, and if we instantiate σ in $\exists\sigma.\psi$ or $\forall\sigma.\psi$ by state s, then we append s at the end of the state sequence and replace all a_σ that is bound by the given quantifier by a_i with i being the index of s in the state sequence. We will express the meaning of path formulas based on the n-ary self-composition of $\mathcal{M}$; the index i for the instantiation of σ also fixes the component index in which we keep track of the paths starting in σ. The semantics judgment rules to evaluate formulas in the context of a DTMC $\mathcal{M} = (S, \mathbf{P}, \mathsf{AP}, L)$ and an n-tuple $s = (s_1, \ldots, s_n) \in S^n$ of states are the following:

$$
\begin{array}{lcl}
\mathcal{M}, s \models \forall\sigma.\psi & \textit{iff} & \mathcal{M}, (s_1, \ldots, s_n, s_{n+1}) \models \psi[\mathsf{AP}_{n+1}/\mathsf{AP}_\sigma] \textit{ for all } s_{n+1} \in S \\
\mathcal{M}, s \models \exists\sigma.\psi & \textit{iff} & \mathcal{M}, (s_1, \ldots, s_n, s_{n+1}) \models \psi[\mathsf{AP}_{n+1}/\mathsf{AP}_\sigma] \textit{ for some } s_{n+1} \in S \\
\mathcal{M}, s \models a_i & \textit{iff} & a \in L(s_i) \\
\mathcal{M}, s \models \psi_1 \wedge \psi_2 & \textit{iff} & \mathcal{M}, s \models \psi_1 \textit{ and } \mathcal{M}, s \models \psi_2 \\
\mathcal{M}, s \models \neg\psi & \textit{iff} & \mathcal{M}, s \not\models \psi \\
\mathcal{M}, s \models p_1 \sim p_2 & \textit{iff} & [\![p_1]\!]_{\mathcal{M},s} \sim [\![p_2]\!]_{\mathcal{M},s} \\
[\![\mathbb{P}(\bigcirc\varphi)]\!]_{\mathcal{M},s} & = & Pr\Big(\{\pi \in Paths^s(\mathcal{M}^n) \mid \mathcal{M}, \pi[1] \models \varphi\}\Big) \\
[\![\mathbb{P}(\varphi_1 \mathcal{U} \varphi_2)]\!]_{\mathcal{M},s} & = & Pr\Big(\{\pi \in Paths^s(\mathcal{M}^n) \mid \textit{exists } j \geq 0 \textit{ such that } \mathcal{M}, \pi[j] \models \varphi_2 \\
 & & \quad \textit{and } \pi[i] \models \varphi_1 \textit{ for all } 0 \leq i < j\}\Big) \\
[\![f(p_1, \ldots, p_k)]\!]_{\mathcal{M},s} & = & f([\![p_1]\!]_{\mathcal{M},s}, \ldots, [\![p_k]\!]_{\mathcal{M},s}) \\
\mathcal{M}, s \models \varphi_1 \wedge \varphi_2 & \textit{iff} & \mathcal{M}, s \models \varphi_1 \textit{ and } \mathcal{M}, s \models \varphi_2 \\
\mathcal{M}, s \models \neg\varphi & \textit{iff} & \mathcal{M}, s \not\models \varphi
\end{array}
$$

where ψ, ψ_1, and ψ_2 are ReachHyperPCTL state formulas; the substitution $\psi[\mathsf{AP}_{n+1}/\mathsf{AP}_\sigma]$ replaces for each atomic proposition $a \in \mathsf{AP}$ each free occurrence of a_σ in ψ by a_{n+1}; $a \in \mathsf{AP}$ is an atomic proposition and $1 \leq i \leq n$; p_1 and p_2 are probability expressions and $\sim \in \{<, \leq$

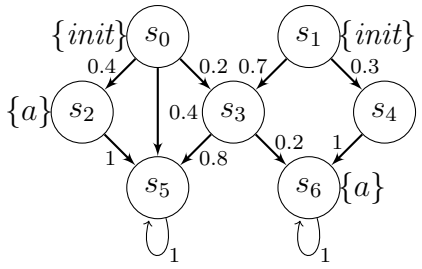


Figure 3.2: Semantics example.

$, =, \geq, >\}$; φ is a ReachHyperPCTL path formula.

Example. The ReachHyperPCTL formula

$$\psi = \forall\sigma.\forall\sigma'.(init_\sigma \wedge init_{\sigma'}) \Rightarrow \left(\mathbb{P}(\Diamond a_\sigma) = \mathbb{P}(\Diamond a_{\sigma'})\right) \tag{3.4}$$

is satisfied by the DTMC $\mathcal{M}$ in Figure 3.2 if for all pairs of *init*-labelled states, the probability to reach a is the same, i.e., for each $(s_i, s_j) \in S^2$ with $init \in L(s_i)$ and $init \in L(s_j)$, it holds that $\mathcal{M}, (s_i, s_j) \models \mathbb{P}(\Diamond a_1) = \mathbb{P}(\Diamond a_2)$. The probability of reaching a from s_0 is $0.4 + (0.2 \times 0.2) = 0.44$. Moreover, the probability of reaching a from s_1 is $(0.7 \times 0.2) + (0.3 \times 1) = 0.44$. Hence, $\mathcal{M} \models \psi$.

3.2 Parameter Synthesis Algorithm for ReachHyperPCTL

Assume in the following a ReachHyperPCTL quantified formula (i.e. sentence) ψ in prenex normal form $\psi = \mathbb{Q}_1\sigma_1 \ldots \mathbb{Q}_n\sigma_n.\psi'$ with quantifiers $\mathbb{Q}_i \in \{\forall, \exists\}$ for $i = 1, \ldots, n$. Assume furthermore, a parametric DTMC $\mathcal{D} = (S, V, \mathbf{P}, \mathsf{AP}, L)$ with valid parameter configuration domain I and let $\mathcal{D}^n = (S^n, V, \mathbf{P}^n, \mathsf{AP}^n, L^n)$ be the n-ary self-composition of $\mathcal{D}$, defined over the same set of atomic propositions as ψ, where n is the number of quantifiers in ψ. Our aim in this section is to provide an algorithm for the synthesis of valid parameter configurations for $\mathcal{D}$ such that ψ is satisfied.

The problem is to decide whether a given fixed valid parameter configuration leads to

the satisfaction of ψ is decidable. Moreover, for a box R of valid parameter configurations of $\mathcal{D}$, also the problem to decide whether all, none, or some of the parameter configurations in R lead to the satisfaction of ψ is solvable. In the following, we propose a parameter synthesis algorithm that will use these computations to decompose a set of valid parameter configurations into a finite set of subsets, and for each of those subsets provide information whether all, none or some configuration in it leads to the satisfaction of the formula. This way, we provide not only a single configurations that satisfy the requirements but even sets of them, and point also to regions that contain no satisfying configurations.

Algorithm 1: Main parameter synthesis algorithm

Input : $\mathcal{D}$: PDTMC; ψ: ReachHyperPCTL formula;
I: a box of valid parameter configurations;
max_{it}: iteration limit.

Output: $(\mathcal{R}_{green}, \mathcal{R}_{white}, \mathcal{R}_{red})$: a decomposition of I into boxes from which all $(\mathcal{R}_{green})$, none $(\mathcal{R}_{red})$ resp. some $(\mathcal{R}_{white})$ configurations make $\mathcal{D}$ satisfy ψ.

1 **Function** *Main*($\mathcal{D}$, ψ, I, max_{it})
2 SymbolicEncoding($\mathcal{D}$, ψ, 0);
3 **return** checkParameterSpace($\mathcal{D}$, ψ, I, max_{it});

The main method is shown in Algorithm 1. In line 2, we first compute for each state $s = (s_1, \ldots, s_n) \in S^n$ of the n-ary self-composition $\mathcal{D}^n$ a real-arithmetic formula $Symb(\psi', s)$ over the model parameters that are true for exactly those parameter configurations under which ψ' holds in state s of $\mathcal{D}^n$. Given a set description R of parameter configurations, unsatisfiability of the formula $R \wedge Symb(\psi', s)$ will thus mean that there is no satisfying configuration in R, whereas unsatisfiability of $R \wedge \neg Symb(\psi', s)$ means that all configurations in R are satisfying for ψ'. If both are satisfiable then some configurations in R satisfy ψ' and some violate it.

Once the symbolic truth values of the input formula are constructed, in line 3 of Algorithm 1 we try to determine regions in the parameter space such that the input formula either evaluates to true under all parameter values in the region or it evaluates to false for all of them. We will use the constants GREEN $= 1$ respectively RED $= -1$ to encode these

properties, and we will use WHITE $= 0$ to express that none of these properties hold.

Next, we explain the two above-mentioned computations. The symbolic values for ψ' and all of its sub-formulas are computed by Algorithm 2. Besides the PDTMC model and a probability expression of a ReachHyperPCTL formula F whose value needs to be computed, the algorithm receives as a third input how many quantifiers we have already processed; this is needed to be able to determine the position of quantifiers during recursive calls on sub-formulas. If F is atomic then we can compute its value in a given state by looking at the state labeling. If F is non-atomic then it is the application of an operator to some operands; the interesting case is when F is the application of a probability operator, in all other cases we call the same method recursively to compute symbolic values for the operands

Algorithm 2: Symbolic value encoding: Main algorithm

Input : $\mathcal{D}$=$(S, V, \mathbf{P}, \mathsf{AP}, L)$: PDTMC; φ: ReachHyperPCTL formula or expression; i: number of already processed quantifiers.

Function *SymbolicEncoding*($\mathcal{D}$, φ, i)
  if φ *is* $(\forall\sigma.\ \psi)$ *or* $(\exists\sigma.\ \psi)$ **then**
    $i := i + 1$;
    SymbolicEncoding($\mathcal{D}$, $\psi[\mathsf{AP}_i/\mathsf{AP}_\sigma]$, i);
  else if φ *is* a_j **then**
    foreach $s = (s_1, \ldots, s_n) \in S^n$ **do**
      if $a \in L(s_j)$ **then** $Symb(\varphi, s) :=$ true **else** $Symb(\varphi, s) :=$ false;
  else if φ *is* $\psi_1 \wedge \psi_2$ **then**
    SymbolicEncoding($\mathcal{D}$, ψ_1, i); SymbolicEncoding($\mathcal{D}$, ψ_2, i);
    foreach $s = (s_1, \ldots, s_n) \in S^n$ **do** $Symb(\varphi, s) := Symb(\psi_1, s) \wedge Symb(\psi_2, s)$;
  else if φ *is* $\neg\psi$ **then**
    SymbolicEncoding($\mathcal{D}$, ψ, i);
    foreach $s = (s_1, \ldots, s_n) \in S^n$ **do** $Symb(\varphi, s) := \neg Symb(\psi, s)$;
  else if φ *is* $\mathbb{P}(\bigcirc\varphi)$ **then** SymbolicEncodingNext($\mathcal{D}$, φ);
  else if φ *is* $\mathbb{P}(\varphi_1 \mathcal{U} \varphi_2)$ **then** SymbolicEncodingUntil($\mathcal{D}$, φ);
  else if φ *is* $p_1 \sim p_2$ **then**
    SymbolicEncoding($\mathcal{D}$, p_1, i); SymbolicEncoding($\mathcal{D}$, p_2, i);
    foreach $s = (s_1, \ldots, s_n) \in S^n$ **do** $Symb(\varphi, s) := Symb(p_1, s) \sim Symb(p_2, s)$;
  else if φ *is* c **then** **foreach** $s = (s_1, \ldots, s_n) \in S^n$ **do** $Symb(\varphi, s) := c$;
  else if φ *is* p_1 *op* p_2 *with* $op \in \{+, -, *\}$ **then**
    SymbolicEncoding($\mathcal{D}$, p_1, i); SymbolicEncoding($\mathcal{D}$, p_2, i);
    foreach $s = (s_1, \ldots, s_n) \in S^n$ **do** $Symb(\varphi, s) := Symb(p_1, s)$ *op* $Symb(p_2, s)$;

Algorithm 3: Symbolic value encoding: Computation for next

Input : $\mathcal{D} = (S, V, \mathbf{P}, \mathsf{AP}, L)$: PDTMC; ReachHyperPCTL expression $\mathbb{P}(\bigcirc \varphi)$

Function *SymbolicEncodingNext*($\mathcal{D}$, $\mathbb{P}(\bigcirc \varphi)$)
  $K = \{\ s \in S^n \ \mid\ \mathcal{D}^n, s \models \varphi\ \}$;
  foreach $s \in S^n$ **do** $\mathit{Symb}(\mathbb{P}(\bigcirc \varphi), s) := \sum_{s' \in K} \mathbf{P}(s, s')$;

Algorithm 4: Symbolic value encoding: Computation for until

Input : $\mathcal{D} = (S, V, \mathbf{P}, \mathsf{AP}, L)$: PDTMC; ReachHyperPCTL expression $\mathbb{P}(\varphi_1 \mathcal{U} \varphi_2)$

Function *SymbolicEncodingUntil*($\mathcal{D}$, $\mathbb{P}(\varphi_1 \mathcal{U} \varphi_2)$)
  $S_1 := \{s \in S^n \ \mid\ \mathcal{D}^n, s \models \varphi_2\}$;
  $S_0 := \{s \in S^n \ \mid\ \mathcal{D}^n, s \models \neg\varphi_1 \wedge \neg\varphi_2\}$;
  $S_? := S^n \setminus (S_1 \cup S_0)$;
  foreach $s \in S_1$ **do**
    $\mathit{Symb}(\mathbb{P}(\varphi_1 \mathcal{U} \varphi_2), s) := 1$; $\mathbf{P}^n(s, s) := 1$;
    foreach *successor* $s_2 \in S^n \setminus \{s\}$ *of* s **do** $\mathbf{P}^n(s, s_2) := 0$;
  foreach $s \in S_0$ **do**
    $\mathit{Symb}(\mathbb{P}(\varphi_1 \mathcal{U} \varphi_2), s) := 0$; $\mathbf{P}^n(s, s) := 1$;
    foreach *successor* $s_2 \in S^n \setminus \{s\}$ *of* s **do** $\mathbf{P}^n(s, s_2) := 0$;
  foreach $s \in S_?$ **do**
    if $\mathbf{P}^n(s, s) \notin \{0, 1\}$ **then**
      foreach *successor* $s_2 \in S^n \setminus \{s\}$ *of* s **do** $\mathbf{P}^n(s, s_2)\ {}^*{=} \frac{1}{1 - \mathbf{P}^n(s,s)}$;
      $\mathbf{P}^n(s, s) := 0$;
    foreach *predecessor* $s_1 \in S^n \setminus \{s\}$ *of* s **do**
      foreach *successor* $s_2 \in S^n \setminus \{s\}$ *of* s **do**
        $\mathbf{P}^n(s_1, s_2)\ {+}{=}\ \mathbf{P}^n(s_1, s) \cdot \mathbf{P}^n(s, s_2)$;
      $\mathbf{P}^n(s_1, s) := 0$;
  foreach $s \in S_?$ **do** $\mathit{Symb}(\mathbb{P}(\varphi_1 \mathcal{U} \varphi_2), s) := \sum_{s_2 \in S_1} \mathbf{P}(s, s_2)$;

and subsequently syntactically connect those by the respective operator.

There are two cases for the probability operator, one for the probability of a next-expression and one for the until. The symbolic encodings for them are computed by the Algorithms 3 and 4, respectively. The former is quite straightforward: to encode the value of $F = \mathbb{P}(\bigcirc \varphi)$ we first determine the set K of those states of $\mathcal{D}^n$ that satisfy φ and then for each $s \in S^n$ the value of F can be encoded by summing up for each direct successor of s that is included in K the probability to move there (in one step). Note that φ is a Boolean combination of atomic propositions, therefore its truth can be easily determined for each state.

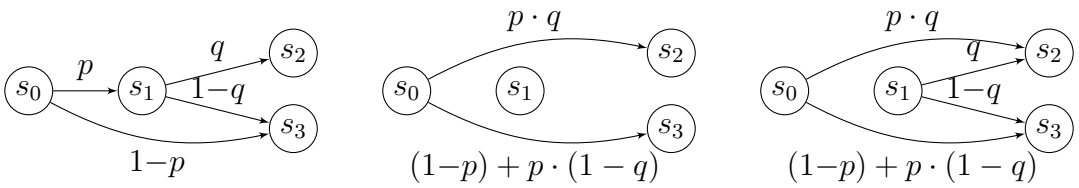


Figure 3.3: A PDTMC (left) and the result of eliminating s_1 in [DJJ+15] (mid) and in Alg. 4 (right).

The case for F being a probability expression $\mathbb{P}(\varphi_1 \mathcal{U} \varphi_2)$ is a bit more involved. We use state elimination, similar to the method used in [DJJ+15] to symbolically express reachability probabilities by arithmetic expressions (rational functions). However, whereas in [DJJ+15] such a reachability probability needs to be computed for a single initial state, for Reach-HyperPCTL properties we need it for *all* states of a parametric DTMC. An algorithm that computes these probability expressions independently, repeatedly applying the method from [DJJ+15] to each state, would work but it would re-do a lot of computations.

Instead, we apply a slight modification to the standard state elimination approach to make some additional bookkeeping. We first identify states for which the probability is known to be one (state set S_1) resp. zero (S_0), and make them absorbing (lines 2–10). Then for each remaining state s, we remove self-loops, and connect pairs of predecessors and successors by direct transitions without visiting s in-between, and then remove the *incoming* edges of s. As illustrated in Fig. 3.3, the difference to the approach in [DJJ+15] is that we do not remove the *outgoing* edges of s, such that after having iterated over all states (lines 11–18), direct transitions from all states to the absorbing ones will remain that allow to express the reachability properties for all states similarly as it was done for the next operator (line 19).

Once we have for all states $s \in S^n$ a symbolic description of the satisfaction of ψ' in s, we can start to search for satisfying and violating parameter configurations using Algorithm 5. It maintains three sets, each of which contains zero or more boxes. Boxes from $\mathcal{R}_{green}$ contain only satisfying parameter configurations, boxes from $\mathcal{R}_{red}$ only unsatisfying ones, whereas

boxes from $\mathcal{R}_{white}$ are mixed and contain both configuration types. We call the boxes from the respective sets accordingly green, red, or white.

Algorithm 5: checkParameterSpace

Input : $\mathcal{D}$: PDTMC; ψ: ReachHyperPCTL formula;
I: a box of valid parameter configurations;
max_{it}: upper iteration limit.

Output: $(\mathcal{R}_{green}, \mathcal{R}_{white}, \mathcal{R}_{red})$: three sets of boxes decomposing I, such that each box from $\mathcal{R}_{green}$, $\mathcal{R}_{red}$ resp. $\mathcal{R}_{white}$ contains configurations from which all, none resp. some make $\mathcal{D}$ satisfy ψ.

```
1  Function checkParameterSpace(D, ψ, I, max_it)
2      R_green := ∅; R_red := ∅; R_white := ∅; R := {I}; l := 1;
3      while R ≠ ∅ do
4          let R ∈ R;   R := R\{R};
5          color:= checkRegion(D, ψ, R, ());
6          if color= GREEN then  R_green := R_green ∪ {R}
7          else if color= RED then  R_red := R_red ∪ {R}
8          else
9              if l < max_it then
10                 S := split(R);  l += |S|;  R := R ∪ S;
11             else  R_white := R_white ∪ {R};
12     return (R_green, R_white, R_red);
```

A queue $\mathcal{R}$ contains at the start of the initial box. Iteratively, we take a box R from $\mathcal{R}$ and determine with Algorithm 6 its colour. If the colour is green or red then we put the box into the corresponding set $\mathcal{R}_{green}$ resp. $\mathcal{R}_{red}$. Otherwise, if the colour is white then we *split* R into smaller boxes which are then added to $\mathcal{R}$ for further processing. For the split, any heuristics can be used, in the hope that the smaller boxes will become conclusive in their colour. To ensure termination, after an upper limit of max_{it} boxes have been scheduled for processing in $\mathcal{R}$, we finish by checking the remaining boxes in the queue without splitting and collect the inconclusive ones in $\mathcal{R}_{white}$.

Finally, the last module to discuss is Algorithm 6 which determines the colour of a box R, i.e. the truth value of $\psi = Q_1 x_1 \ldots Q_n x_n . \psi'$ under configurations from R. Let us first have a look at the lines 11-13, where the truth value of ψ' is checked for a fixed state $(s_1, \ldots, s_n)$ of the n-ary self-composition. Here, for a box $R = [l_1, u_1] \times \ldots \times [l_n, u_n]$ we overload notation and

use R also to denote its logical description $\bigwedge_{i=1}^{n} l_i \leq x_i \wedge x_i \leq u_i$. Since $Symb(\psi, (s_1, \ldots, s_n))$ encodes the value of ψ in $(s_1, \ldots, s_n)$, the formula $R \wedge Symb(\psi, (s_1, \ldots, s_n))$ is true for all configurations in R that satisfy ψ. If this formula is unsatisfiable then we know that no configuration in R satisfies ψ and return the colour RED. In contrary, if $R \wedge \neg Symb(\psi, (s_1, \ldots, s_n))$ is unsatisfiable then we know that none of the configurations in R violate ψ and the colour of the box is GREEN. Otherwise, if both formulas are satisfiable then some configurations in R satisfy ψ and some others do not, therefore the colour of the box is WHITE.

It depends on the quantifiers for which states we need to execute this check, as implemented in lines 2-9. For each existential quantifier $Q_i = \exists$ we need to find just a single state that makes the box GREEN in order to make the formula true, whereas for universal quantifiers $Q_i = \forall$ it needs to hold for each state. For the latter case it means also that if the box is red for one state then we know that According to this, quantifiers are instantiated from left to right, and the previously described code in lines 11-13 is applied to check the colour of the box for the chosen n-ary state vector.

As a result of the satisfiability checks in line 11 of Algorithm 6, for purely existentially quantified formulas we can also provide a satisfying configuration for each white box.

Given the soundness of [DJJ+15], which we use basically unchanged (with the additional bookkeeping shown in Fig. 3.3) to compute symbolic probabilities for the states, our computations in lines 11–13 are sound for each state. Furthermore, for universal state quantifiers, we take the weakest satisfaction result under all states, and for existential state quantifiers the strongest one, such that the soundness of our algorithm is easy to see assuming soundness of [DJJ+15]).

3.3 Case Studies and Evaluation

We developed a prototypical implementation of our algorithm in Python, with the help of several libraries that facilitate the handling of complex mathematical equations involved. There is extensive use of STORMPY [stob], which is a set of Python bindings for the prob-

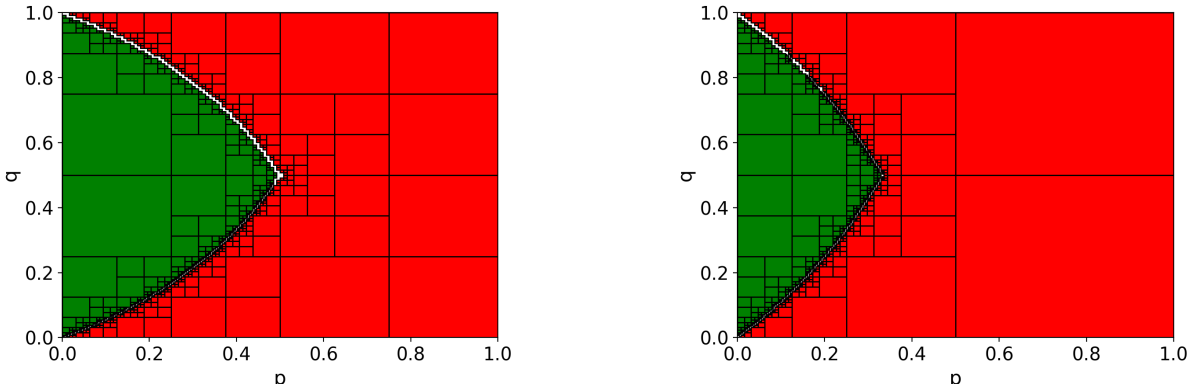
Algorithm 6: checkRegion

Input : $\mathcal{D} = (S, V, \mathbf{P}, \mathsf{AP}, L)$: PDTMC; ψ: ReachHyperPCTL formula; R: a box of valid parameter configurations; $(s_1, \ldots, s_{i-1}) \in S^{i-1}$.

Output: *color*: one of the colors GREEN=1, WHITE=0 or RED=-1 encoding whether ψ is satisfied by $\mathcal{D}$ under all, some respectively none of the configurations in R.

1 **Function** *checkRegion*($\mathcal{D}$, ψ, R, $(s_1, \ldots, s_{i-1})$)
2 **if** ψ *is* $Q_i x_i. \ldots Q_n x_n.\psi'$ **then**
3 **if** $Q_i = \exists$ **then** *color*:= RED **else** *color*:= GREEN;
4 **foreach** $s_i \in S$ **do**
5 *color*$'$:= checkRegion($\mathcal{D}$, $Q_{i+1}x_{i+1}. \ldots Q_n x_n.\psi'$, R, $(s_1, \ldots, s_{i-1}, s_i)$);
6 **if** $Q_i = \exists$ **then** *color*:= $\max\{$*color*, *color*$'\}$;
7 **else if** $Q_i = \forall$ **then** *color*:= $\min\{$*color*, *color*$'\}$;
8 **if** *(*$Q_i = \exists \wedge$ *color* = GREEN*) or (*$Q_i = \forall$ *and color* = RED*)* **then** break
9 **return** *color*
10 **else**
11 **if** $R \wedge$ *Symb*$(\psi, (s_1, \ldots, s_n))$ *is unsatisfiable* **then return** RED
12 **else if** $R \wedge \neg$*Symb*$(\psi, (s_1, \ldots, s_n))$ *is unsatisfiable* **then return** GREEN
13 **else return** WHITE

abilistic model checker STORM [DJKV17]. It has provided an efficient solution for parsing, building, and storage of parametric DTMC models. Internally, STORMPY uses pycarl [pyc], the python binding of CARL, an Open Source C++ Library for Computer Arithmetic and Logic. Several data structures and data types have been used from pycarl and STORMPY to handle complex polynomials, equations, and rational numbers. Finally, we have used the SMT solver Z3 [dMB08] to implement lines 11 and 12 of Algorithm 6. All of our experiments are run on a MacBook Pro laptop with a 2.7 GHz i7 processor with 8GB of RAM. We set $max_{it} = 1500$ in Algorithm 5, the process always the oldest inconclusive box in $\mathcal{R}$ (FIFO) and split inconclusive d-dimensional white boxes into 2^d new box by splitting in the middle in each dimension. We start with two smaller examples (randomized response and probabilistic conformance) and then switch to larger case studies (probabilistic noninterference and information leakage).



(a) $(\ln 3)$-differential privacy. (b) $(\ln 2)$-differential privacy.

Figure 3.4: Synthesized probability distribution for the randomized response protocol.

3.3.1 Randomized Response

We first synthesize configurations for the randomized response protocol described in Section 3.1. We experimented with the following scenarios. In the first scenario, we synthesized parameters to achieve $\ln 3$-differential privacy with parameters p and q as shown in Fig 3.1b. The green area in Fig. 3.4a includes the valid values ($p = 0.75$ and $q = 0.25$, or, $p = q = 0.5$). The white area consists of the values that remain unknown due to the termination of the algorithm after 1500 rounds. In the second scenario, our goal is to achieve $\ln 2$-differential privacy. Again, the green area in Fig. 3.4b includes the valid values (e.g., $p = \frac{2}{3}$ and $q = 0.25$). The time spent to synthesize parameters in all the above scenarios was $0.1s$.

3.3.2 Probabilistic Conformance

The notion of *conformance* describes how well a system implements correctly a given specification in terms of observable behaviours, or, whether two systems (e.g., a model and an implementation) conform with each other with respect to a specification. In our setting, we model both specifications and implementation as PDTMCs.

As an example, let us consider the specification of a protocol, where a A 6-sided die is rolled as long as we get the number six (state d_6). Figure 3.5 (left) illustrates graphically our example. Our specification states also that the die, after behaving fairly the first time, can

be biased only once according to a parameter p. Now, our goal is to implement this protocol using an adaptation of the Knuth-Yao algorithm [KY76] that was designed originally for simulating a 6-sided die by repeatedly tossing a fair coin, illustrated in Figure 3.5 (right). In the considered adaptation, we allow bias in the tossing of the coin according to the same parameter p, only when the state d'_6 (this state represents the number six in the simulated die) is encountered.

In this experiment, we are interested to find the value of p, such that the implementation of the protocol conforms with its specifications according to the probability of terminating in each one of the five possible states that represent the numbers of the die from one to five. This property can be formally expressed using the following ReachHyperPCTL formula:

$$\varphi_{\mathsf{pc}} = \forall\sigma.\exists\sigma'.\Big(s_{0_\sigma} \wedge s'_{0_{\sigma'}}\Big) \;\Rightarrow\; \bigwedge_{i=1}^{5}\Big(\mathbb{P}(\Diamond\, d_{i_\sigma}) = \mathbb{P}(\Diamond\, d'_{i_{\sigma'}})\Big) \tag{3.5}$$

We synthesized parameter value $p = 0.5$, meaning that applying a fair coin ensures conformance of the implementation (the right model in Figure 3.5) with the specification (the left PDTMC in Figure 3.5). The synthesis for this experiment was $28s$, where $27s$ was spent in Algorithms 2-4 and $1s$ in Algorithms 5-6. The imbalance is mainly due to the fact that the PDTMCs have multiple nested cycles. We also note that $p = 0.5$ is the only valid solution.

3.3.3 Probabilistic Noninterference in Randomized Schedulers

Noninterference is an information-flow security policy that enforces that a low-privileged user (e.g., an attacker) should not be able to distinguish two computations from their publicly observable outputs if they only vary in their inputs by a high-privileged user (e.g., a secret).

Probabilistic noninterference [III92] establishes connection between information theory and information flow by employing probabilities to address covert channels. Intuitively, it requires that the probability of every low-observable trace pattern is the same for every low-equivalent initial state. Consider the following example [Smi03] of a program with two

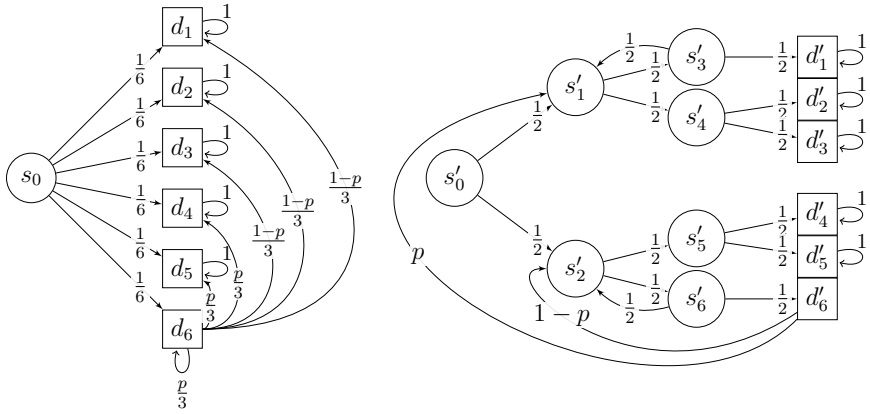


Figure 3.5: Parametric die-coin using Knuth-Yao protocol.

threads th_1 and th_2:

$$th_1 : \ \textbf{while}\ h > 0\ \textbf{do}\ \{h := h - 1\};\ l := 2 \qquad || \qquad th_2 : \ l := 1 \tag{3.6}$$

where h is a *secret* input by a high-privileged user and l is a *public* output observable by low-privileged users. Figure 3.6 depicts PDTMC models of this program for secret input $h = 0$ (left) and $h = 1$ (right). The solid (resp., dotted) transitions correspond to thread th_1 (resp., th_2), i labels initial states, proposition, w denotes execution of the while-loop condition checking, and f denotes the terminating state. The parameter configuration $p = 0.5$ resp. $q = 0.5$ models a fair scheduler that chooses each of the threads with equal probability for the execution of the next atomic statement. Probabilistic noninterference requires that l obtains the value of 1 (and 2) with equal probability, regardless of the initial value of h:

$$\varphi_{\mathsf{pni}} = \forall\sigma.\forall\sigma'.\Big(i_\sigma \wedge (h{=}0)_\sigma \wedge i_{\sigma'} \wedge (h{=}1)_{\sigma'}\Big) \Rightarrow \Big(\Big(\mathbb{P}\big(\Diamond(f_\sigma \wedge (l{=}1)_\sigma)\big) = \mathbb{P}\big(\Diamond(f_{\sigma'} \wedge (l{=}1)_{\sigma'})\big)\Big) \wedge \Big(\mathbb{P}\big(\Diamond(f_\sigma \wedge (l{=}2)_\sigma)\big) = \mathbb{P}\big(\Diamond(f_{\sigma'} \wedge (l{=}2)_{\sigma'})\big)\Big)\Big) \tag{3.7}$$

However, when using a fair scheduler, the likely outcome of the race between the two assignments $l := 1$ and $l := 2$ depends on the initial value of h: the larger the initial value

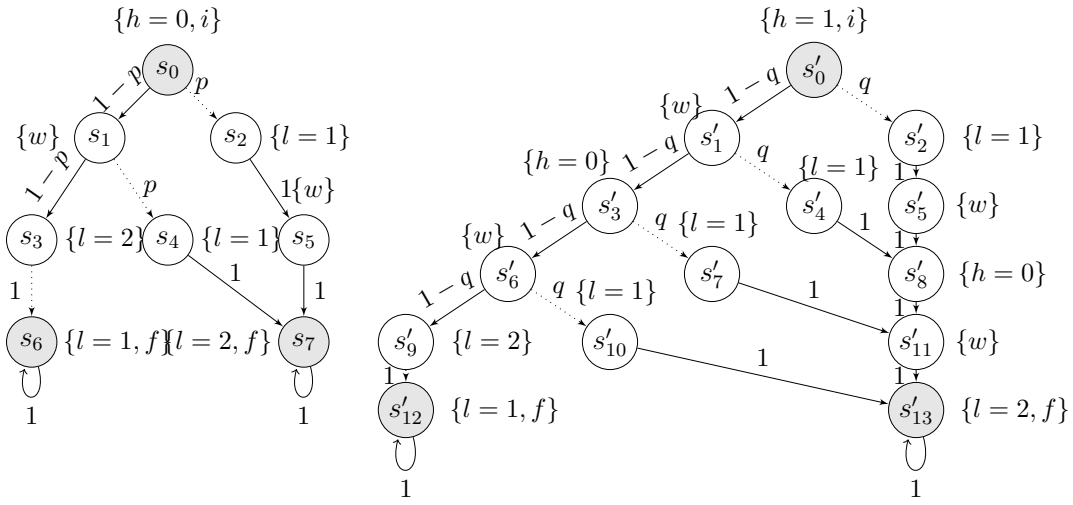


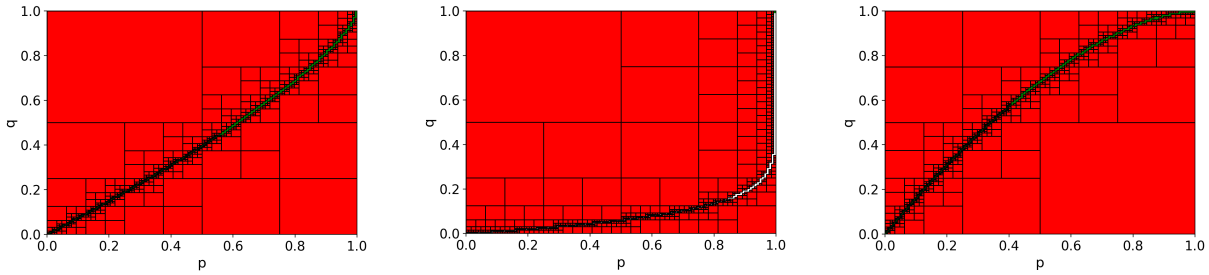
Figure 3.6: Parametric DTMC for the probabilistic noninterference example program with two threads th_1 and th_2.

Input h	**Running time** (s)			#white boxes	#red boxes	red area	#samples
	Alg.2-4	**Alg.5-6**	**Total**				
$(0, 1)$	2.90	100.03	102.93	378	748	0.79	477
$(0, 5)$	15.61	143.58	159.2	374	752	0.815	421
$(0, 10)$	55.73	259.3	315.06	374	752	0.8164	480
$(0, 15)$	113.58	459.60	573.18	377	749	0.711	413
$(1, 2)$	8.33	114.55	122.88	368	758	0.706	425
$(3, 5)$	31.95	204.42	236.38	411	715	0.831	496
$(4, 8)$	72.23	397.91	470.14	371	755	0.6622	481
$(8, 14)$	213.96	2924.61	3138.07	378	748	0.825	496

Table 3.1: Experimental results for thread scheduling.

of h, the greater the probability that the final value of l is 2. For example, it is easy to recognize in Fig. 3.6 that for the secret input $h = 0$ the final value is $l = 1$ with probability $1/4$ and $l = 2$ with probability $3/4$, but for the input $h = 1$ the final value is $l = 1$ with probability $1/16$ and $l = 2$ with probability $15/16$. Thus, it holds that for two independent executions with initial h values 0 resp. 1 the larger h value leads to a lower probability for $l = 1$ upon termination i.e., this program does not satisfy φ_{pni}.

Now, let us repair this system by allowing the scheduler to use *biased coins* for different secret input values h. Table 3.1 shows experimental results for different input value combina-



(a) $(h = 3)_\sigma, (h = 5)_{\sigma'}$. (b) $(h = 0)_\sigma, (h = 10)_{\sigma'}$. (c) $(h = 14)_\sigma, (h = 8)_{\sigma'}$.

Figure 3.7: Synthesized probability distribution for a randomized scheduler.

tions, for which we want to determine parameters that assure probabilistic noninterference. The table shows for each considered pair of h-values the time spent in Algorithms 1–4 and Algorithms 5–6, the total running time, the number of returned white and red boxes (no green boxes have been detected), the percentage of the configuration domain covered by red boxes (i.e. provably non-satisfying area), and the number of satisfying configurations (samples) detected. Figure 3.7 shows the 2-dimensional plot of synthesized valid values of parameters p and q for different pair values of h. As can be seen, as the values of h converge (e.g., in Fig. 3.7a), the probabilities of p and q also converge, as the resulting DTMC is balanced. On the contrary, as the values of h diverge (e.g., in Fig. 3.7b), the probabilities of p and q also diverge, as the resulting DTMC is more imbalanced.

3.3.4 Information Leakage in Dining Cryptographers

Three cryptographers gather around a table for dinner. The waiter informs them that the meal has been paid for by someone, who could be either one of the three cryptographers or the master. The cryptographers respect each other's privacy but want to find out whether the master paid. To decide this, they execute the following two-stage protocol [Cha88]:

- Each cryptographer flips a coin and informs only the cryptographer on the right about the outcome.
- Each cryptographer who did not pay for the dinner announces whether the two coins that it can see (the own flipped one and the one on the left-hand neighbor flipped) are

the same ("agree") or different ("disagree").

- If a cryptographer paid for dinner, then it instead states the opposite ("disagree" if the coins are the same and "agree" if the coins are different).

A parametric DTMC model of the protocol with three cryptographers and three biased coins (with parameters p_1, p_2, and p_3, respectively) is illustrated in Figure 3.8. It consists of four independent sub-PDTMCS, staying for the four cases who paid: the master (M), the first (C_1), the second (C_2) resp. third (C_3) cryptographer. We depict all four PDTMCS in one illustration as they differ only in their state labeling. The labels pay^M and pay^i, $i \in \underline{3}$ encode that the master resp. cryptographer i paid; t^i resp. h^i encodes tail resp. head flipped by cryptographer i; the labels a^i resp. d^i encode that cryptographer i announced "agree" resp. "disagree"; finally, *done* stays for a terminated protocol. In the text, the above state identifiers s^i are lower indexed with the cases to distinguish between s^i_M, $s^i_{C_1}$, $s^i_{C_2}$, $s^i_{C_3}$.We are interested in deciding which parts of the valid parameter domain $[0,1]^3 \subseteq \mathbb{R}^3$ satisfy the following ReachHyperPCTL formula ($\oplus$ denotes the exclusive-or operator):

$$\varphi_{\mathsf{dc}} = \quad \forall\sigma.\, \forall\sigma'. \quad \Big((\bigvee_{i\in\underline{3}} pay^i_\sigma) \wedge (\bigvee_{i\in\underline{3}} pay^i_{\sigma'})\Big) \;\Rightarrow \tag{3.8}$$
$$\underbrace{\mathbb{P}\Big(\diamondsuit(done_\sigma \wedge (a^1_\sigma \oplus a^2_\sigma \oplus a^3_\sigma))\Big)}_{F_1} = \underbrace{\mathbb{P}\Big(\diamondsuit(done_{\sigma'} \wedge (a^1_{\sigma'} \oplus a^2_{\sigma'} \oplus a^3_{\sigma'}))\Big)}_{F_2}$$

In other words, if the master does not pay, then the different outcomes are observed with the same probabilities independently of the fact which cryptographer has paid. A careful reader recognizes that the above property holds for all parameters. Intuitively, independently of the flip outcomes, when ordered in a circle, the number of changes in the outcomes will be always even. Thus, the number of "agree"s will be even if and only if an even number of cryptographers lie. Therefore, when the master paid (zero lies) we have an even number of "agree"s, and when one of the cryptographers paid then one lies and we have an odd number.

To check this property, we follow Algorithm 1 and call first the SymbolicEncoding method

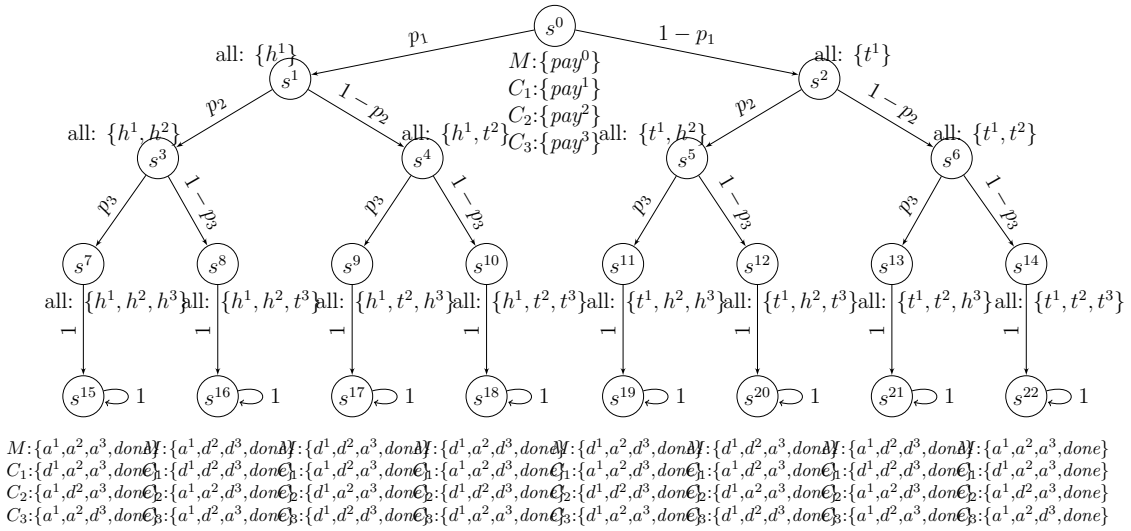


Figure 3.8: A parametric DTMC model for the dining cryptographers protocol with three cryptographers and three biased coins.

from Algorithm 2 on the 2-ary self-composition of the PDTMC in Figure 3.8. The states of this self-composition are pairs $(s^i_{T_1}, s^j_{T_2})$ with $i, j \in \underline{22}$ and $T_1, T_2 \in \{M, C_1, C_2, C_3\}$. Note that the self-composition is synchronous, i.e., each the non-absorbing state has four successors; for example, the state $(s^0_{C_1}, s^0_{C_2})$ has the successors (1) $(s^1_{C_1}, s^1_{C_2})$ with probability $p_1 \cdot p_1$, (2) $(s^1_{C_1}, s^2_{C_2})$ with probability $p_1 \cdot (1-p_1)$, (3) $(s^2_{C_1}, s^1_{C_2})$ with probability $(1-p_1) \cdot p_1$, and (4) $(s^2_{C_1}, s^2_{C_2})$ with probability $(1-p_1) \cdot (1-p_1)$.

The formula 3.8 is trivially satisfied by all states where the left-hand-side of the implication is false, i.e., the only relevant initial states (instantiating σ and σ') are $(s^0_{T_1}, s^0_{T_2})$ with $T_1, T_2 \in \{C_1, C_2, C_3\}$. Due to the synchronous nature of the self-composition, both runs start in σ resp. σ' will execute the same number of steps, i.e. stay at the same "depth" in Figure 3.8. For all such state pairs, the probability expressions F_1 and F_2 in Formula 3.8 both simplify to 1, therefore the equality $F_1 = F_2$ holds independently from the parameter configuration.

Starting with the parameter domain $[0,1]^3$, our implementation reports that the whole box $[0,1]^3$ is green, without any splits. However, the (symbolic) transition matrix is quite large (we have 7744 states in the 2-ary self-composition and our implementation does not detect non-reachable states), so it takes about 40 minutes running time to get this answer.

3.4 Summary

When designing a system, often designers have to abide by certain pre-specified requirements. Working backward, the randomness or bias in the system is then designed according to the requirement it has to satisfy. This is the parameter synthesis problem. In this work, we focused on defining and solving the parameter synthesis problem for probabilistic hyperproperties on DTMCs in particular. We have defined the problem, the fragment of the logic we considered for simplicity, and proposed an algorithm to solve the problem. We further demonstrated our approach in a few interesting case studies. Although computationally challenging, the problem is interesting and useful.

Chapter 4

Probabilistic Hyperproperties with Nondeterminism

4.1 Introduction

Hyperproperties [CS10] can express probabilistic relations between multiple executions of a system and can describe the requirements of probabilistic systems. For example, in information-flow security, adding probabilities is motivated by establishing a connection between information theory and information flow across multiple traces. A prominent example is probabilistic schedulers that open up an opportunity for an attacker to set up a probabilistic covert channel. Or, *probabilistic causation* compares the probability of occurrence of an effect between scenarios where the cause is or is not present.

The state of the art on probabilistic hyperproperties has exclusively been studied in the context of discrete-time Markov chains (DTMCs). The temporal logic HyperPCTL [ÁB18], extends PCTL by allowing explicit and simultaneous quantification over computation trees. For example, the DTMC in Fig. 4.1 satisfies the following HyperPCTL formula:

$$\psi = \forall \hat{s}.\forall \hat{s}'.\Big(init_{\hat{s}} \wedge init_{\hat{s}'}\Big) \Rightarrow \Big(\mathbb{P}(\Diamond\, a_{\hat{s}}) = \mathbb{P}(\Diamond\, a_{\hat{s}'})\Big) \tag{4.1}$$

which means that the probability of reaching proposition a from any pair of states $\hat{s}$ and $\hat{s}'$ labelled by *init* should be equal. Other works on probabilistic hyperproperties for DTMCs

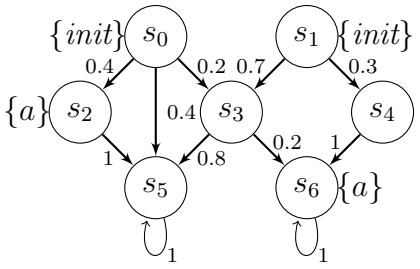
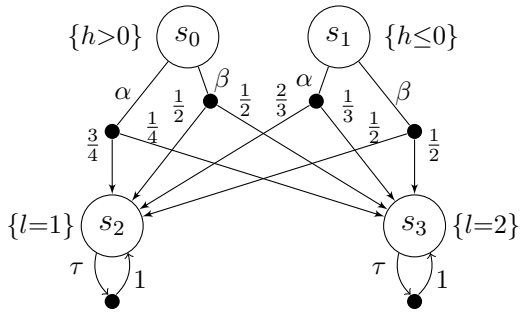


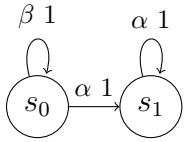
Figure 4.1: Example DTMC.

include parameter synthesis [ÁBBD20a] and statistical model checking [WZBP19, WZBP20].

An important gap in the spectrum is the verification of probabilistic hyperproperties concerning models that allow *nondeterminism*, in particular, *Markov decision processes* (MDP). Nondeterminism plays a crucial role in many probabilistic systems. For instance, nondeterministic queries can be exploited to make targeted attacks on databases with private information [GMB17].



(a) MDP showing the actions and probabilistic distributions.



(b) MDP requiring probabilistic schedulers.

Figure 4.2: MDPs that require different types of schedulers.

To motivate the idea, consider the MDP in Fig. 4.2a, where h is a high secret and l is a low publicly observable variable. To protect the secret, there should be no probabilistic dependencies between observations of l and the value of h. On one hand, an attacker that chooses a scheduler that always takes action α from states s_0 and s_1 can learn whether or not $h \leq 0$ by observing the probability of obtaining $l = 1$ (or $l = 2$). On the other hand, a scheduler that always chooses action β does not leak any information about the value of h. Thus, a natural question to ask is whether a certain property holds for all or some schedulers.

With the above motivation, we focus on probabilistic hyperproperties in the context of MDPs. Such hyperproperties inherently need to consider different nondeterministic choices in different executions, and naturally call for quantification over schedulers. There are several challenges to achieving this. In general, there are schedulers whose reachability probabilities cannot be achieved by any memoryless non-probabilistic scheduler, and, hence finding a scheduler is not reducible to checking non-probabilistic memoryless schedulers, as it is done in PCTL model checking for MDPs. Consider for example the MDP in Fig. 4.2b, for which we want to know whether there is a scheduler such that the probability of reaching s_1 from s_0 equals 0.5. There are two non-probabilistic memoryless schedulers, one choosing action α and the other, action β in s_0. The first one is the maximal scheduler for which s_1 is reached with probability 1, and the second one is the minimal scheduler leading to probability 0. However, the probability 0.5 cannot be achieved by any non-probabilistic scheduler. *Memoryless* probabilistic schedulers can neither achieve probability 0.5: if a memoryless scheduler would take action α with any positive probability, then the probability to reach s_1 is always 1. The only way to achieve the reachability probability 0.5 (or any value strictly between 0 and 1) is by a probabilistic scheduler with memory, e.g., taking α and β in s_0 with probabilities 0.5 each when this is the first step on a path, and β with probability 1 otherwise.

To this effect, we first extend the temporal logic HyperPCTL [ÁB18] to the context of MDPs. To this end, we augment the syntax and semantics of HyperPCTL to quantify over schedulers and relate probabilistic computation trees for different schedulers. For example, the following formula generalizes (4.1) by requiring that the respective property should hold for all computation trees starting in any states $\hat{s}$ and $\hat{s}'$ of the DTMC induced by any scheduler $\hat{\sigma}$:

$$\forall\hat{\sigma}.\forall\hat{s}(\hat{\sigma}).\forall\hat{s}'(\hat{\sigma}).\Big(init_{\hat{s}} \wedge init_{\hat{s}'}\Big) \Rightarrow \Big(\mathbb{P}(\Diamond a_{\hat{s}}) = \mathbb{P}(\Diamond a_{\hat{s}'})\Big) \tag{4.2}$$

On the negative side, we show that the problem to check HyperPCTL properties for MDPs is in general undecidable. On the positive side, we show that the problem be-

comes decidable when we restrict the scheduler quantification domain to memoryless non-probabilistic schedulers. We also show that this restricted problem is already NP-complete (respectively, coNP-complete) in the size of the given MDP for HyperPCTL formulas with a single existential (respectively, universal) scheduler quantifier. Subsequently, we propose an SMT-based encoding to solve the restricted model checking problem. We have implemented our method and analyse it experimentally on three case studies: probabilistic scheduling attacks, side-channel timing attacks, and probabilistic conformance (available at https://github.com/TART-MSU/HyperProb).

4.2 HyperPCTL for MDPs

We now describe the syntax and semantics of our extension of the temporal logic HyperPCTL to the context of MDPs.

4.2.1 HyperPCTL Syntax

HyperPCTL (quantified) state formulas φ^q are inductively defined as follows:

$$
\begin{array}{lllll}
\textit{quantified formula} & \varphi^q & ::= & \forall\hat{\sigma}.\varphi^q \mid \exists\hat{\sigma}.\varphi^q \mid \forall\hat{s}(\hat{\sigma}).\varphi^q \mid \exists\hat{s}(\hat{\sigma}).\varphi^q \mid \varphi^{nq} \\
\textit{non-quantified formula} & \varphi^{nq} & ::= & \texttt{true} \mid a_{\hat{s}} \mid \varphi^{nq} \wedge \varphi^{nq} \mid \neg\varphi^{nq} \mid \varphi^{pr} < \varphi^{pr} \\
\textit{probability expression} & \varphi^{pr} & ::= & \mathbb{P}(\varphi^{path}) \mid f(\varphi^{pr}_1, \ldots, \varphi^{pr}_k) \\
\textit{path formula} & \varphi^{path} & ::= & \bigcirc\varphi^{nq} \mid \varphi^{nq}\,\mathcal{U}\,\varphi^{nq} \mid \varphi^{nq}\,\mathcal{U}^{[k_1,k_2]}\,\varphi^{nq}
\end{array}
$$

where $\hat{\sigma}$ is a *scheduler variable*[1] from an infinite set $\hat{\Sigma}$, $\hat{s}$ is a *state variable* from an infinite set $\hat{\mathcal{S}}$, φ^{nq} is a quantifier-free state formula, $a \in \mathsf{AP}$ is an atomic proposition, φ^{pr} is a *probability expression*, $f : [0,1]^k \to \mathbb{R}$ are k-ary elementary functions to express operations over probabilities, arithmetic operators (binary addition, unary/binary subtraction, binary multiplication) over probabilities, where constants are viewed as 0-ary functions, and φ^{path} is a *path formula*, such that $k_1 \leq k_2 \in \mathbb{N}_{\geq 0}$. The probability operator $\mathbb{P}$ allows the usage of

[1]We use the notation $\hat{\sigma}$ for scheduler variables and σ for schedulers, and analogously $\hat{s}$ for state variables and s for states.

probabilities in arithmetic constraints and relations.

A HyperPCTL construct φ (probability expression φ^{pr}, state formula φ^{q}, φ^{nq} or path formula φ^{path}) is *well-formed* if each occurrence of any $a_{\hat{s}}$ with $a \in \mathsf{AP}$ and $\hat{s} \in \hat{\mathcal{S}}$ is in the scope of a *state quantifier* for $\hat{s}(\hat{\sigma})$ for some $\hat{\sigma} \in \hat{\Sigma}$, and any quantifier for $\hat{s}(\hat{\sigma})$ is in the scope of a *scheduler quantifier* for $\hat{\sigma}$. We restrict ourselves to quantifying first the schedulers then the states, i.e., different state variables can share the same scheduler. One can consider also *local* schedulers when different players cannot explicitly share the same scheduler, or in other words, each scheduler quantifier belongs to exactly one of the quantified states.

HyperPCTL formulas are well-formed HyperPCTL state formulas, where we additionally allow standard syntactic sugar like $\texttt{false} = \neg\texttt{true}$, $\varphi_1 \vee \varphi_2 = \neg(\neg\varphi_1 \wedge \neg\varphi_2)$, $\Diamond\varphi = \texttt{true}\,\mathcal{U}\,\varphi$, and $\mathbb{P}(\Box\varphi) = 1 - \mathbb{P}(\Diamond\neg\varphi)$. For example, the HyperPCTL state formula $\forall\hat{\sigma}.\exists\hat{s}(\hat{\sigma}).\mathbb{P}(\bigcirc a_{\hat{s}}) < 0.5$ is a HyperPCTL formula. The HyperPCTL state formula $\mathbb{P}(\bigcirc a_{\hat{s}})<0.5$ is not a HyperPCTL formula, but can be extended to such. The HyperPCTL state formula $\forall\hat{s}(\hat{\sigma}).\exists\hat{\sigma}.\mathbb{P}(\bigcirc a_{\hat{s}})<0.5$ is not a HyperPCTL formula, and it even cannot can be extended to such.

4.2.2 HyperPCTL Semantics

Definition 4.2.1. The *n-ary self-composition* of an MDP $\mathcal{M} = (\mathcal{S}, Act, P, \mathsf{AP}, L)$ for a sequence $\boldsymbol{\sigma} = (\sigma_1, \ldots, \sigma_n) \in (\Sigma^{\mathcal{M}})^n$ of schedulers for $\mathcal{M}$ is the DTMC parallel composition $\mathcal{M}^{\boldsymbol{\sigma}} = \mathcal{M}_1^{\sigma_1} \times \ldots \times \mathcal{M}_n^{\sigma_n}$, where $\mathcal{M}_i^{\sigma_i}$ is the DTMC induced by $\mathcal{M}_i$ and σ_i, and where $\mathcal{M}_i = (\mathcal{S}, Act, \mathbb{P}, \mathsf{AP}_i, L_i)$ with $\mathsf{AP}_i = \{a_i \mid a \in \mathsf{AP}\}$ and $L_i(s) = \{a_i \mid a \in L(s)\}$, for all $s \in S$. ■

HyperPCTL state formulas are evaluated in the context of an MDP $\mathcal{M} = (\mathcal{S}, Act, \mathbb{P}, \mathsf{AP}, L)$, a sequence $\boldsymbol{\sigma} = (\sigma_1, \ldots, \sigma_n) \in (\Sigma^{\mathcal{M}})^n$ of schedulers, and a sequence $\vec{r} = ((q_1, s_1), \ldots, (q_n, s_n))$ of $\mathcal{M}^{\boldsymbol{\sigma}}$ states; we use $()$ to denote the empty sequence (of any type) and $\circ$ for concatenation. Intuitively, these sequences store instantiations for scheduler and state variables. The satisfaction of a HyperPCTL quantified formula by $\mathcal{M}$ is defined by

$$\mathcal{M} \models \varphi \qquad \textit{iff} \qquad \mathcal{M}, (), () \models \varphi \ .$$

The semantics evaluates HyperPCTL formulas by structural recursion. Let in the following $\mathbb{Q}, \mathbb{Q}', \ldots$ denote quantifiers from $\{\forall, \exists\}$. When instantiating $\mathbb{Q}\hat{\sigma}.\varphi$ by a scheduler $\sigma \in \Sigma^{\mathcal{M}}$, we replace in φ each sub-formula $\mathbb{Q}'\hat{s}(\hat{\sigma}).\varphi'$, that is not in the scope of a quantifier for $\hat{\sigma}$ by $\mathbb{Q}'\hat{s}(\sigma).\varphi'$, and denote the result by $\varphi[\hat{\sigma} \rightsquigarrow \sigma]$. For instantiating a state quantifier $\mathbb{Q}\hat{s}(\sigma).\varphi$ by a state s, we append $\sigma = (Q, act, mode, init)$ and $(init(s), s)$ at the end of the respective sequences, and replace each $a_{\hat{s}}$ in the scope of the given quantifier by a_s, resulting in a formula that we denote by $\varphi[\hat{s} \rightsquigarrow s]$. To evaluate probability expressions, we use the n-ary self-composition of the MDP.

Formally, the semantics judgment rules are as follows:

$$
\begin{array}{lll}
\mathcal{M}, \boldsymbol{\sigma}, \vec{r} \models \texttt{true} & & \\
\mathcal{M}, \boldsymbol{\sigma}, \vec{r} \models a_i & \textit{iff} & a_i \in L^{\boldsymbol{\sigma}}(\vec{r}) \\
\mathcal{M}, \boldsymbol{\sigma}, \vec{r} \models \varphi_1 \wedge \varphi_2 & \textit{iff} & \mathcal{M}, \boldsymbol{\sigma}, \vec{r} \models \varphi_1 \textit{ and } \mathcal{M}, \boldsymbol{\sigma}, \vec{r} \models \varphi_2 \\
\mathcal{M}, \boldsymbol{\sigma}, \vec{r} \models \neg\varphi & \textit{iff} & \mathcal{M}, \boldsymbol{\sigma}, \vec{r} \not\models \varphi \\
\mathcal{M}, \boldsymbol{\sigma}, \vec{r} \models \forall\hat{\sigma}.\varphi & \textit{iff} & \forall\sigma \in \Sigma^{\mathcal{M}}.\ \mathcal{M}, \boldsymbol{\sigma}, \vec{r} \models \varphi[\hat{\sigma} \rightsquigarrow \sigma] \\
\mathcal{M}, \boldsymbol{\sigma}, \vec{r} \models \exists\hat{\sigma}.\varphi & \textit{iff} & \exists\sigma \in \Sigma^{\mathcal{M}}.\ \mathcal{M}, \boldsymbol{\sigma}, \vec{r} \models \varphi[\hat{\sigma} \rightsquigarrow \sigma] \\
\mathcal{M}, \boldsymbol{\sigma}, \vec{r} \models \forall\hat{s}(\sigma).\varphi & \textit{iff} & \forall s_{n+1} \in \mathcal{S}.\ \mathcal{M}, \boldsymbol{\sigma} \circ \sigma, \vec{r} \circ (init(s_{n+1}), s_{n+1}) \models \varphi[\hat{s} \rightsquigarrow s_{n+1}] \\
\mathcal{M}, \boldsymbol{\sigma}, \vec{r} \models \exists\hat{s}(\sigma).\varphi & \textit{iff} & \exists s_{n+1} \in \mathcal{S}.\ \mathcal{M}, \boldsymbol{\sigma} \circ \sigma, \vec{r} \circ (init(s_{n+1}), s_{n+1}) \models \varphi[\hat{s} \rightsquigarrow s_{n+1}] \\
\mathcal{M}, \boldsymbol{\sigma}, \vec{r} \models \varphi_1^{pr} < \varphi_2^{pr} & \textit{iff} & [\![\varphi_1^{pr}]\!]_{\mathcal{M},\boldsymbol{\sigma},\vec{r}} < [\![\varphi_2^{pr}]\!]_{\mathcal{M},\boldsymbol{\sigma},\vec{r}} \\
[\![\mathbb{P}(\varphi_{path})]\!]_{\mathcal{M},\boldsymbol{\sigma},\vec{r}} & = & Pr^{\mathcal{M}^{\boldsymbol{\sigma}}}\big(\{\pi \in Paths^{\vec{r}}(\mathcal{M}^{\boldsymbol{\sigma}}) \mid \mathcal{M}, \boldsymbol{\sigma}, \pi \models \varphi_{path}\}\big) \\
[\![f(\varphi_1^{pr}, \ldots \varphi_k^{pr})]\!]_{\mathcal{M},\boldsymbol{\sigma},\vec{r}} & = & f\big([\![\varphi_1^{pr}]\!]_{\mathcal{M},\boldsymbol{\sigma},\vec{r}} \ldots,\ [\![\varphi_k^{pr}]\!]_{\mathcal{M},\boldsymbol{\sigma},\vec{r}}\big)
\end{array}
$$

where $\mathcal{M}$ is an MDP; $n \in \mathbb{N}_{\geq 0}$ is non-negative integer; $\boldsymbol{\sigma} \in (\Sigma^{\mathcal{M}})^n$; $\vec{r}$ is a state of $\mathcal{M}^{\boldsymbol{\sigma}}$; $a \in \mathsf{AP}$ is an atomic proposition and $i \in \{1, \ldots, n\}$; $\varphi, \varphi_1, \varphi_2$ are HyperPCTL state formulas; $\sigma = (Q, act, mode, init) \in \Sigma^{\mathcal{M}}$ is a scheduler for $\mathcal{M}$; $\varphi_1^{pr} \cdots \varphi_k^{pr}$ are probability expressions,

and φ_{path} is a HyperPCTL path formula whose satisfaction relation is as follows:

$$\begin{aligned}
&\mathcal{M}, \boldsymbol{\sigma}, \pi \models \bigcirc \varphi && \textit{iff } \mathcal{M}, \boldsymbol{\sigma}, \vec{r}_1 \models \varphi \\
&\mathcal{M}, \boldsymbol{\sigma}, \pi \models \varphi_1 \mathcal{U} \varphi_2 && \textit{iff } \exists j \geq 0. \Big(\mathcal{M}, \boldsymbol{\sigma}, \vec{r}_j \models \varphi_2 \wedge \forall i \in [0, j).\ \mathcal{M}, \boldsymbol{\sigma}, \vec{r}_i \models \varphi_1 \Big) \\
&\mathcal{M}, \boldsymbol{\sigma}, \pi \models \varphi_1 \mathcal{U}^{[k_1, k_2]}\ \varphi_2 && \textit{iff } \exists j \in [k_1, k_2]. \Big(\mathcal{M}, \boldsymbol{\sigma}, \vec{r}_j \models \varphi_2 \wedge \forall i \in [0, j). \mathcal{M}, \boldsymbol{\sigma}, \vec{r}_i \models \varphi_1 \Big)
\end{aligned}$$

where $\pi = \vec{r}_0 \vec{r}_1 \cdots$ with $\vec{r}_i = ((q_{i,1}, s_{i,1}), \ldots, (q_{i,n}, s_{i,n}))$ is a path of $\mathcal{M}^{\boldsymbol{\sigma}}$; formulas φ, φ_1, and φ_2 are HyperPCTL state formulas, and $k_1 \leq k_2 \in \mathbb{N}_{\geq 0}$.

4.3 The Expressiveness Power of HyperPCTL

The standard PCTL semantics define that to satisfy a PCTL formula $\mathbb{P}_{\sim c}(\varphi)$ in a given MDP state s, *all* schedulers should induce a DTMC that satisfies $\mathbb{P}_{\sim c}(\varphi)$ in s. Though it should hold for *all* schedulers, it is known that there exist minimal and maximal schedulers that are non-probabilistic and memoryless, therefore it is sufficient to restrict the reasoning to such schedulers. Since for MDPs with finite state and action spaces, the number of such schedulers is finite, PCTL model checking for MDPs is decidable. Given this analogy, one would expect that HyperPCTL model checking should be decidable, but it is not.

Theorem 4.3.1. HyperPCTL model checking for MDPs is in general undecidable.

Before we prove the above theorem, let us explore shortly, the source of increased expressiveness with respect to PCTL that makes HyperPCTL undecidable. State quantification cannot be the source, as the state space is finite and thus, there are finitely many possible state quantifier instantiations.

Assume an MDP $\mathcal{M} = (\mathcal{S}, Act, Pr, \mathsf{AP}, L)$ with a state $s \in \mathcal{S}$ that is uniquely labeled by the proposition $init \in L(s)$, and let $a, b \in \mathsf{AP}$. In PCTL, each probability bound needs to be satisfied under all schedulers. For example:

$$\begin{aligned}
& \mathcal{M}, s \ \models_{\mathsf{PCTL}} && \mathbb{P}_{<0.5}(a \mathcal{U} b) \\
\Leftrightarrow\ & \mathcal{M} \ \models_{\mathsf{HyperPCTL}} && \forall \hat{\sigma}(\hat{\mathcal{M}}).\forall \hat{s}(\hat{\mathcal{M}}^{\hat{\sigma}}).(init_{\hat{s}} \rightarrow \mathbb{P}(a_{\hat{s}}\ \mathcal{U}\ b_{\hat{s}}) < 0.5)
\end{aligned}$$

Alternatively, we can state:

$$\begin{array}{rlll} & \mathcal{M}, s & \models_{\mathsf{PCTL}} & \mathbb{P}_{<0.5}(a\,\mathcal{U}\,b) \\ \Leftrightarrow \quad \forall\sigma{\in}\Sigma^{\mathcal{M}}. & 1, \mathcal{M}^{\sigma}[1], ((init_{\sigma}(s), s)) & \models_{\mathsf{HyperPCTL}} & \mathbb{P}(a_1\,\mathcal{U}\,b_1){<}0.5 \end{array}$$

where $init_{\sigma}(s)$ is the starting mode of scheduler σ in state s. Generally, the HyperPCTL fragment which starts with a single universal scheduler quantifier and contains a single bound on a single probability operator is still decidable. However, when a PCTL formula has several probability bounds, its satisfaction requires each bound to be satisfied by all schedulers *independently*. For example, $\mathcal{M}, s \models_{\mathsf{PCTL}} \Big(\mathbb{P}_{<0.5}(a\,\mathcal{U}b) \ \vee \ \mathbb{P}_{>0.5}(a\mathcal{U}b)\Big)$ is equivalent to

$\mathcal{M} \models_{\mathsf{HyperPCTL}} \forall\hat{\sigma}.\forall\hat{s}(\hat{\sigma}).(init_{\hat{s}} \rightarrow \mathbb{P}(a_{\hat{s}}\,\mathcal{U}b_{\hat{s}}) < 0.5)$ *or*

$\mathcal{M} \models_{\mathsf{HyperPCTL}} \forall\hat{\sigma}.\forall\hat{s}(\hat{\sigma}).(init_{\hat{s}} \rightarrow \mathbb{P}(a_{\hat{s}}\mathcal{U}b_{\hat{s}}) > 0.5)$

but *not* equivalent to the HyperPCTL formula

$$\mathcal{M} \models_{\mathsf{HyperPCTL}} \forall\hat{\sigma}.\forall\hat{s}(\hat{\sigma}).(init_{\hat{s}} \rightarrow (\mathbb{P}(a_{\hat{s}}\mathcal{U}b_{\hat{s}}) < 0.5 \vee \mathbb{P}(a_{\hat{s}}\mathcal{U}b_{\hat{s}}) > 0.5))$$

which states that the probability is either less than or larger than 0.5 under all schedulers, which is true if *there exists no scheduler* under which the probability is 0.5 (see also [BBGK12]). Thus, even for a fragment restricted to universal scheduler quantification, combinations of probability bounds allows HyperPCTL to express existential *scheduler synthesis* problems.

Finally, consider a scheduler quantifier followed by state quantifiers, whose scope may contain probability expressions. This means that we start several "experiments" in parallel, each one represented by a state quantifier. However, we may use in all experiments the *same scheduler*. Informally, this allows us to express the existence or absence of schedulers with certain probabilistic hyperproperties for the induced DTMCs. It would however also make sense to flip this quantifier order, such that state quantifiers are followed by scheduler quantifiers. This would mean, that we can use different schedulers in the different concurrently running experiments. This would be meaningful e.g., when users can provide input to the system, i.e., when the scheduler choice lies with the "observers" of the individual experiments, and they can adapt their schedulers to observations made in the other concurrently

running experiments.

Proof of Theorem 4.3.1 —

To prove Theorem 4.3.1, we reduce the *emptiness* problem in *probabilistic Büchi automata* (*PBA*), which is known to be undecidable [BBG08], to our problem. PBA can be viewed as a nondeterministic Büchi automaton where the nondeterminism is resolved by a probabilistic choice. That is, for any state q and letter a in alphabet Σ, either q does not have any a-successor or there is a probability distribution for the a-successors of q.

Definition 4.3.1. A *probabilistic Büchi automaton* (PBA) over a finite alphabet Σ is a tuple $\mathcal{P} = (Q, \delta, \Sigma, F)$, where,

- Q is a finite state space,
- $\delta\colon Q \times \Sigma \times Q \to [0,1]$ is the transition probability function, such that for all $q \in Q$ and $a \in \Sigma$: $\sum_{q' \in Q}(q, a, q') \in \{0, 1\}$ for all $q \in Q$ and $a \in \Sigma$,
- $F \subseteq Q$ is the set of *accepting states*.

A *run* for an infinite word $w = a_1 a_2 \cdots \in \Sigma^\omega$ is an infinite sequence $\pi = q_0 q_1 q_2 \cdots$ of states in Q, such that $q_{i+1} \in \delta(q_i, a_{i+1}) = \{q' \mid \delta(q_i, a_{i+1}, q') > 0\}$ for all $i \in \mathbb{N}_{\geq 0}$. Let $\mathsf{Inf}(\pi)$ denote the set of states that are visited infinitely often in π. Run π is called *accepting* if $\mathsf{Inf}(\pi) \cap F \neq \emptyset$. Given an infinite input word $w \in \Sigma^\omega$, the behaviour of $\mathcal{P}$ is given by the infinite Markov chain that is obtained by unfolding $\mathcal{P}$ into a tree using w. This is similar to an induced Markov chain from an MDP by a scheduler. Hence, standard concepts for Markov chains can be applied to define the acceptance probability of w in $\mathcal{P}$, denoted by $Pr_{\mathcal{P}}(w)$ or briefly $Pr(w)$, by the probability measure of the set of accepting runs for w in $\mathcal{P}$. We define the accepted language of $\mathcal{P}$ as: $\mathcal{L}(\mathcal{P}) = \{w \in \Sigma^\omega \mid Pr_{\mathcal{P}}(w) > 0\}$. The *emptiness problem* is to decide whether or not $\mathcal{L}(\mathcal{P})=\emptyset$ for a given input $\mathcal{P}$.

Mapping Our idea of mapping the emptiness problem in PBA to HyperPCTL model checking for MDPs is as follows. We map a PBA to an MDP such that the words of the PBA are mimicked by the runs of the MDP. In other words, the letters of the words in the

PBA appear as propositions on states of the MDP. This way, the existence of a word in the language of the PBA corresponds to the existence of a scheduler that produces a satisfying computation tree in the induced Markov chain of the MDP.

MDP model: Let $\mathcal{P} = (Q, \delta, \Sigma, F)$ be a PBA with alphabet Σ. We obtain an MDP $\mathcal{M} = (\mathcal{S}, Act, P, \mathsf{AP}, L)$ as follows:

- The set of states is $\mathcal{S} = Q \times \Sigma$.
- The set of actions is $Act = \Sigma$.
- The transition probability function P is defined as follows:

$$P\Big((q,a), b, (q',a')\Big) = \begin{cases} \delta(q,b,q') & \text{if } a' = b \\ 0 & \text{otherwise} \end{cases}$$

- The set of atomic propositions is $\mathsf{AP} = \Sigma \cup \{f\}$, where $f \notin \Sigma$ (we use f to label the accepting states).
- The labeling function L is defined for each $a \in \Sigma$ and $q \in Q$ as follows:

$$L(q,a) = \begin{cases} \{a, f\} & \text{if } q \in F \\ \{a\} & \text{otherwise} \end{cases}$$

HyperPCTL formula: The HyperPCTL formula in our mapping is

$$\varphi_{\mathsf{map}} = \exists \hat{\sigma}(\hat{\mathcal{M}}).\exists \hat{s}(\hat{\mathcal{M}}^{\hat{\sigma}}).\forall \hat{s}'(\hat{\mathcal{M}}^{\hat{\sigma}}).\ \Big(\mathbb{P}\big(\square \bigwedge_{a \in \mathsf{AP}\setminus\{f\}} (a_{\hat{s}} \leftrightarrow a_{\hat{s}'})\big) = 1\Big) \wedge \Big(\mathbb{P}\Big(\diamondsuit \mathbb{P}\big(\square \mathbb{P}(\diamondsuit f_{\hat{s}}) = 1\big) = 1\Big) > 0\Big) \tag{4.3}$$

Intuitively, the above formula establishes the connection between the PBA emptiness problem and HyperPCTL model checking problem for MDPs. In particular:

- The existence of a scheduler $\hat{\sigma}(\mathcal{M})$ in φ_{map} corresponds to the existence of a word w in $\mathcal{L}(\mathcal{P})$;
- the state quantifiers and the left conjunct ensure that the path in the induced Markov chain and the PBA follow the sequence of actions (respectively, letters) in the witness to $\hat{\sigma}(\mathcal{M})$ (respectively, w), and
- the right conjunct mimics that a state in F is visited with non-zero probability if and only if a state labeled by proposition f is visited infinitely often in the MDP with non-zero probability.

Reduction We now show that $\mathcal{L}(\mathcal{P}) \neq \emptyset$ if and only if $\mathcal{M} \models \varphi_{\mathsf{map}}$. We distinguish two cases:

($\rightarrow$) Suppose we have $\mathcal{L}(\mathcal{P}) \neq \emptyset$. This means there exists a word $w \in \Sigma^{\omega}$, such that $Pr_{\mathcal{P}}(w) > 0$. We use w to eliminate the existential scheduler quantifier and instantiate $\hat{\sigma}(\hat{\mathcal{M}})$ in formula φ_{map}. This induces a DTMC and now, we show that the induced DTMC satisfies the following HyperPCTL formula:

$$\exists \hat{s}(\hat{\mathcal{M}}^{\hat{\sigma}}).\forall \hat{s}'(\hat{\mathcal{M}}^{\hat{\sigma}}).\Big(\mathbb{P}\big(\square \bigwedge_{a \in \mathsf{AP}\setminus\{f\}} (a_{\hat{s}} \leftrightarrow a_{\hat{s}'})\big) = 1\Big) \wedge \Big(\mathbb{P}\Big(\diamondsuit \mathbb{P}\big(\square \mathbb{P}(\diamondsuit f_{\hat{s}}) = 1\big) = 1\Big) > 0\Big) \tag{4.4}$$

To this end, observe that the right conjunct is trivially satisfied because $Pr_{\mathcal{P}}(w) > 0$. That is since a state in F is visited infinitely often with non-zero probability in $\mathcal{P}$, a state labeled by f in $\mathcal{M}$ is also visited infinitely often with non-zero probability. The left conjunct is also satisfied by the construction of the mapped MDP since the sequence of letters in w appear in all paths of the induced DTMC as propositions.

($\leftarrow$) The reverse direction is pretty similar. Since the answer to the model checking problem is affirmative, a witness to scheduler quantifier $\hat{\sigma}$ exists. This scheduler induces a

DTMC whose paths follow the same sequence of propositions. This sequence indeed provides us with the word w for $\mathcal{P}$. Finally, since the right conjunct in φ_{map} is satisfied by the MDP, we are guaranteed that w reaches an accepting state in F infinitely often with non-zero probability.

This concludes the proof.

4.4 Applications of HyperPCTL on MDPs

Side-channel timing leaks open a channel to an attacker to infer the value of a secret by observing the execution time of a function. For example, the heart of the RSA public-key encryption algorithm is the modular exponentiation algorithm that computes ($a^b \mod n$), where a is an integer representing the plain text and b is the integer encryption key.

```
1  void mexp(){
2    c = 0; d = 1; i = k;
3    while (i >= 0){
4      i = i-1; c = c*2;
5      d = (d*d) % n;
6      if (b(i) = 1)
7        c = c+1;
8        d = (d*a) % n;
9      }
10 }
11 /************/
12 t = new Thread(mexp());
13 j = 0; m = 2 * k;
14 while (j < m & !t.stop) j++;
15 /************/
```

Figure 4.3: Modular exponentiation.

A careless implementation can leak b through a probabilistic scheduling channel (see Fig. 4.3 on the left). This program is not secure since the two branches of the *if* have different timing behaviours. Under a fair execution scheduler for parallel threads, an attacker thread can infer the value of b by running in parallel to a modular exponentiation thread and iteratively incrementing a countervariable until the other thread terminates (lines 12-14).

```
int str_cmp(char * r){
  char * s = 'Bg\$4\0';
  i = 0;
  while (s[i] != '\0'){
    i++;
    if (s[i]!=r[i]) return 0;
    }
    return 1;
}
```

Figure 4.4: String comparison.

To model this program by an MDP, we can use two nondeterministic actions for the two branches of the *if* statement, such that the choice of different schedulers corresponds to the choice of different bit configurations `b(i)` for the key `b`. This algorithm should satisfy the following property: the probability of observing a concrete value in the counter `j` should be independent of the bit configuration of the secret key `b`:

$$\forall\hat{\sigma}_1.\forall\hat{\sigma}_2.\forall\hat{s}(\hat{\sigma}_1).\forall\hat{s}'(\hat{\sigma}_2).\Big(init_{\hat{s}} \wedge init_{\hat{s}'}\Big) \Rightarrow \bigwedge_{l=0}^{m}\Big(\mathbb{P}(\Diamond(j=l)_{\hat{s}}) = \mathbb{P}(\Diamond(j=l)_{\hat{s}'})\Big) \tag{4.5}$$

Another example of timing attacks that can be implemented through a probabilistic scheduling side channel is password verification which is typically implemented by comparing an input string with another confidential string (see Fig 4.4).

Also here, an attacker thread can measure the time necessary to break the loop, and use this information to infer the prefix of the input string matching the secret string.

Scheduler-specific observational determinism policy (SSODP) [NSH13] is a confidentiality policy in multi-threaded programs that defend against an attacker that chooses an appropriate scheduler to control the set of possible traces. In particular, given any scheduler and two initial states that are indistinguishable with respect to a secret input (i.e., low-equivalent), any two executions from these two states should terminate in low-equivalent

states with equal probability. Formally, given a proposition h representing a secret:

$$\forall\hat{\sigma}.\forall\hat{s}(\hat{\sigma}).\forall\hat{s}'(\hat{\sigma}).\big(h_{\hat{s}} \oplus h_{\hat{s}'}\big) \Rightarrow \bigwedge_{l\in L} \big(\mathbb{P}(\Diamond\, l_{\hat{s}}) = \mathbb{P}(\Diamond\, l_{\hat{s}'})\big) \tag{4.6}$$

where $l \in L$ are atomic propositions that classify low-equivalent states and $\oplus$ is the exclusive-or operator. A stronger variation of this policy is that the executions are stepwise low-equivalent:

$$\forall\hat{\sigma}.\forall\hat{s}(\hat{\sigma}).\forall\hat{s}'(\hat{\sigma}).\big(h_{\hat{s}} \oplus h_{\hat{s}'}\big) \Rightarrow \mathbb{P}\Box\big(\bigwedge_{l\in L} \big((\mathbb{P}\bigcirc l_{\hat{s}}) = (\mathbb{P}\bigcirc l_{\hat{s}'})\big)\big) = 1. \tag{4.7}$$

Probabilistic conformance describes how well a model and an implementation conforms with each other with respect to a specification. As an example, consider a 6-sided die. The probability to obtain one possible side of the die is 1/6. We would like to synthesize a protocol that simulates the 6-sided die behaviour only by repeatedly tossing a fair coin. We know that such an implementation exists [KY76], but we aim to find such a solution automatically by modeling the die as a DTMC and by using an MDP to model all the possible coin-implementations with a given maximum number of states, including 6 absorbing final states to model the outcomes. In the MDP, we associate to each state a set of possible non-deterministic actions, each of them choosing two states as successors with equal probability 1/2. Then, each scheduler corresponds to a particular implementation. Our goal is to check whether there exists a scheduler that induces a DTMC over the MDP, such that repeatedly tossing a coin simulates die-rolling with equal probabilities for the different outcomes:

$$\exists\hat{\sigma}.\forall\hat{s}(\hat{\sigma}).\exists\hat{s}'(\hat{\sigma}).\Big(init_{\hat{s}} \wedge init_{\hat{s}'}\Big) \Rightarrow \bigwedge_{l=1}^{6} \Big(\mathbb{P}(\Diamond(die = l)_{\hat{s}}) = \mathbb{P}(\Diamond(die = l)_{\hat{s}'})\Big) \tag{4.8}$$

4.5 Summary

In this chapter, we have elaborated on the extension of syntax and semantics needed to accommodate non-determinism in HyperPCTL. We have proved that the model checking problem for MDPs, in general, is undecidable, owing to the extensive set of possible schedulers

that can be used to resolve non-determinism in the system. Next, we will focus on a restricted segment of this logic.

Chapter 5

Decidable Fragment of Probabilistic Hyperproperties with Nondeterminism

5.1 Introduction

Following from the previous chapter, due to the undecidability of HyperPCTL formulas for MDPs, we focus in this chapter on the restricted semantics of HyperPCTL, where scheduler quantification ranges over non-probabilistic memoryless schedulers only. It is easy to see that limiting ourselves to non-probabilistic memoryless schedulers makes the model checking problem decidable, as there are only finitely many such schedulers. Regarding complexity, we have the following result.

Theorem 5.1.1. The problem to decide for MDPs the truth of HyperPCTL formulas with a single existential (respectively, universal) scheduler quantifier over non-probabilistic memoryless schedulers is NP-complete (respectively, coNP-complete) in the state set size of the given MDP.

5.2 Proof of decidability of the restricted fragment

In order to show membership to NP, let $\mathcal{M}$ be an MDP and $\varphi = \exists\hat{\sigma}(\hat{\mathcal{M}}).\varphi'$ be a HyperPCTL formula, where φ' is a state quantified formula. We show that given a solution

to the problem, we can verify the solution in polynomial time. Observe that given a non-probabilistic memoryless scheduler as a witness to the existential quantifier $\exists\hat{\sigma}(\hat{\mathcal{M}})$, one can compute the induced DTMC and then verify the DTMC against the resulting HyperPCTL formula in polynomial time in the size of the induced DTMC [ÁB18].

Inspired by the proof technique introduced in [BF18], for the lower bound, we reduce the SAT problem to our model checking problem. The SAT problem is as follows:

> Let $y = y_1 \wedge y_2 \wedge \cdots \wedge y_m$ be a Boolean formula where each y_j, for $j \in [1, m]$, is a disjunction of at least three literals using propositions $\{x_1, x_2, \ldots, x_n\}$. Is y satisfiable, i.e., is there an assignment of truth values to $x_1, x_2, \ldots, x_n$, such that y evaluates to true?

Mapping We now present a mapping from an arbitrary SAT problem instance to the model checking problem of an MDP and a HyperPCTL formula of the form $\exists\hat{\sigma}(\hat{\mathcal{M}}).\ \exists\hat{s}(\hat{\mathcal{M}}^{\hat{\sigma}}).\ \forall\hat{s}'(\hat{\mathcal{M}}^{\hat{\sigma}}).\ \varphi$. Then, we show that the MDP satisfies this formula if and only if the answer to the SAT problem is affirmative. Fig. 5.1 shows an example.

MDP: For a given propositional logic formula in conjunctive normal form, we define the MDP $\mathcal{M} = (\mathcal{S}, Act, P, \mathsf{AP}, L)$ as follows.

- *(Atomic propositions* AP*)* We include four atomic propositions: p and $\bar{p}$ to mark the positive and negative literals in each clause and c and $\bar{c}$ to mark paths that correspond to clauses of the SAT formula. Thus, $\mathsf{AP} = \{p, \bar{p}, c, \bar{c}\}$.

- *(Set of states* $\mathcal{S}$*)* We now identify the members of $\mathcal{S}$:

  - For each clause y_j, where $j \in [1, m]$, we include a state r_j, labeled by proposition c. We also include a state r_0 labeled by $\bar{c}$.

  - For each clause y_j, $j \in [1, m]$, we introduce the following n states:

$$\Big\{v_i^j \mid i \in [1, n]\Big\}.$$

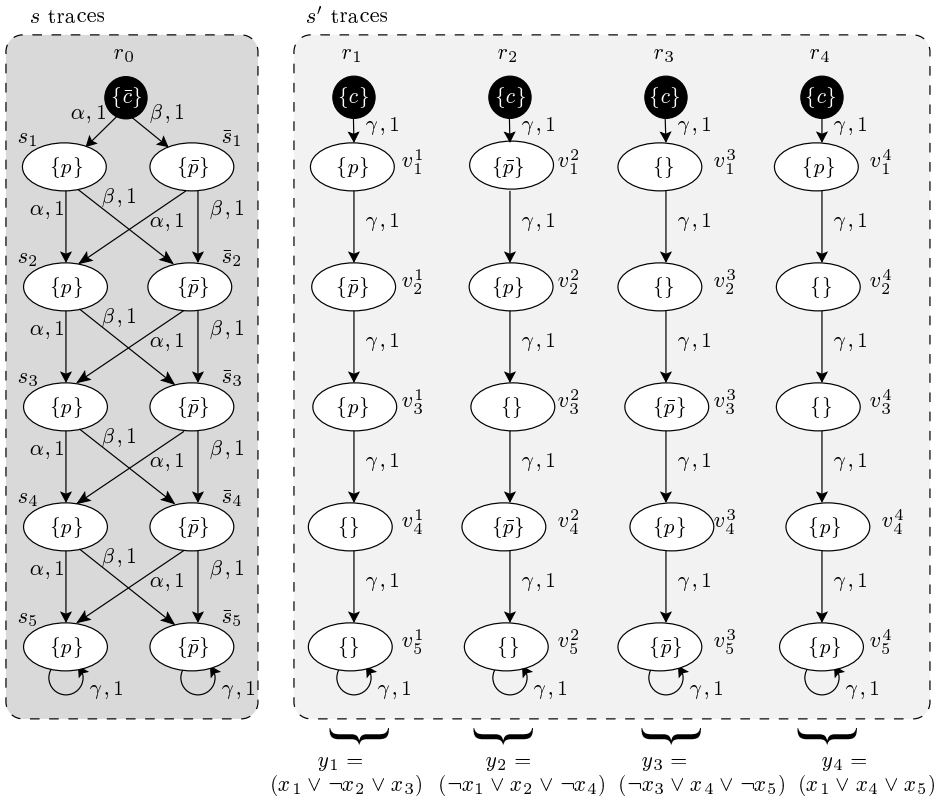


Figure 5.1: Example of mapping SAT to HyperPCTL model checking.

Each state v_i^j is labelled with proposition p if x_i is a literal in y_j, or with $\bar{p}$ if $\neg x_i$ is a literal in y_j.

– For each Boolean variable x_i, where $i \in [1, n]$, we include two states: a state s_i labelled with p and a state $\bar{s}_i$ labelled with $\bar{p}$.

- *(Set of actions Act)* The set of actions is $Act = \{\alpha, \beta, \gamma\}$. Intuitively, the scheduler chooses action α (respectively, β) at a state s_i or $\bar{s}_i$ to assign true (respectively, false) to variable x_{i+1}. Action γ is the sole action available at all other states.

- *(Transition probability function P)* We now identify the members of P. All transitions have probability 1, so we only discuss the actions.

  – We add transitions (r_j, γ, v_1^j) for each $j \in [1, m]$, where from r_j, the probability of reaching v_1^j is 1.

– We also add transitions $(v_i^j, \gamma, v_{i+1}^j)$ for each $i \in [1, n)$, connecting the states representing literals in each clause y_j, $j \in [1, m]$.

– For each $i \in [1, n)$, we include four transitions (s_i, α, s_{i+1}), $(s_i, \beta, \bar{s}_{i+1})$, $(\bar{s}_i, \alpha, s_{i+1})$, and $(\bar{s}_i, \beta, \bar{s}_{i+1})$. The intuition here is that when the scheduler chooses action α at state s_i or $\bar{s}_i$, variable x_{i+1} evaluates to true and when the scheduler chooses action β at state s_i or $\bar{s}_i$, variable x_{i+1} evaluates to false in the SAT instance. We also include two transitions (r_0, α, s_1) and $(r_0, \beta, \bar{s}_1)$ with the same intended meaning.

– Finally, we include self-loops (s_n, γ, s_n), $(\bar{s}_n, \gamma, \bar{s}_n)$, and (v_n^j, γ, v_n^j), for each $j \in [1, m]$.

HyperPCTL formula: The HyperPCTL formula in our mapping is:

$$\varphi_{\mathsf{map}} = \exists \hat{\sigma}(\hat{\mathcal{M}}).\exists \hat{s}(\hat{\mathcal{M}}^{\hat{\sigma}}).\forall \hat{s}'(\hat{\mathcal{M}}^{\hat{\sigma}}).\ \bar{c}_{\hat{s}} \wedge \left(c_{\hat{s}'} \rightarrow \mathbb{P}\left(\Diamond \left((p_{\hat{s}} \wedge p_{\hat{s}'}) \vee (\bar{p}_{\hat{s}} \wedge \bar{p}_{\hat{s}'}) \right) \right) = 1 \right) \quad (5.1)$$

The intended meaning of the formula is that if there exists a scheduler that makes the formula true by choosing the α and β actions, this scheduler gives us the assignment to the Boolean variables in the SAT instance. This is achieved by making all clauses true, hence, the $\forall \hat{s}'(\hat{\mathcal{M}}^{\hat{\sigma}})$ sub-formula.

Reduction We now show that the given SAT formula is satisfiable if and only if the MDP obtained by our mapping satisfies the HyperPCTL formula φ_{map}.

($\rightarrow$) Suppose that y is satisfiable. Then, there is an assignment that makes each clause y_j, where $j \in [1, m]$, true. We now use this assignment to instantiate a scheduler for the formula φ_{map}. If $x_i = \texttt{true}$, then we instantiate scheduler $\hat{\sigma}$ such that in state s_{i-1} or $\bar{s}_{i-1}$, it chooses action α. Likewise, if $x_i = \texttt{false}$, then we instantiate scheduler $\hat{\sigma}$, such that in state s_{i-1} or $\bar{s}_{i-1}$, it chooses action β. We now show that this scheduler instantiation evaluates formula φ_{map} to true. First observe that $\hat{s}(\hat{\mathcal{M}}^{\hat{\sigma}})$ can only be instantiated with state r_0 and $\hat{s}'(\hat{\mathcal{M}}^{\hat{\sigma}})$ can only be instantiated with states r_j, where

$j \in [1, m]$. Otherwise, the left side of the implication in φ_{map} becomes false, making the formula vacuously true. Since each y_j is true, there is at least one literal in y_j that is true. If this literal is of the form x_i, then we have $x_i = \texttt{true}$ and the path that starts from r_0 will include s_i, which is labelled by p. Hence, the values of p, in both paths that start from $\hat{s}(\hat{\mathcal{M}}^{\hat{\sigma}})$ and $\hat{s}'(\hat{\mathcal{M}}^{\hat{\sigma}})$ are eventually equal. If the literal in y_j is of the form $\neg x_i$, then $x_i = \texttt{false}$ and the path that starts from $\hat{s}(\hat{\mathcal{M}}^{\hat{\sigma}})$ will include $\bar{s}_i$. Again, the values of $\bar{p}$ are eventually equal. Finally, since all clauses are true, all paths that start from $\hat{s}'(\hat{\mathcal{M}}^{\hat{\sigma}})$ reach a state where the right side of the implication becomes true.

($\leftarrow$) Suppose our mapped MDP satisfies formula φ_{map}. This means that there exists a scheduler and state $\hat{s}(\hat{\mathcal{M}}^{\hat{\sigma}})$ that makes the sub-formula $\forall \hat{s}'(\hat{\mathcal{M}}^{\hat{\sigma}})$ true, i.e., since $\hat{s}$ can uniquely be instantiated by r_0 due to its labeling by $\bar{c}$, the path that starts from r_0 results in making the inner PCTL formula true for all paths that start from r_j, where $1 \leq j \leq m$, as the left of the implication is false for all other states. We obtain the truth assignment to the SAT problem as follows. If the scheduler chooses action α to state s_i, then we assign $x_i = \texttt{true}$. Likewise, if the scheduler chooses action β to state $\bar{s}_i$, then we assign $x_i = \texttt{false}$. Observe that since in no state p and $\bar{p}$ are simultaneously true and no path includes both s_i and $\bar{s}_i$, variable x_i will have only one truth value. Similar to the forward direction, it is straightforward to see that this valuation makes every clause y_j of the SAT instance true.

5.3 Model Checking for Non-probabilistic Memoryless Schedulers

Due to the undecidability of model checking HyperPCTL formulas for MDPs, we now restrict the semantics, where scheduler quantification ranges over non-probabilistic memoryless schedulers only. It is easy to see that this restriction makes the model checking problem de-

Algorithm 7: Main SMT encoding algorithm

Input : $\mathcal{M} = (\mathcal{S}, Act, \mathbb{P}, \mathsf{AP}, L)$: MDP;
$Q\hat{\sigma}.Q_1\hat{s}_1(\hat{\sigma}).\ldots.Q_n\hat{s}_n(\hat{\sigma}).\varphi^{nq}$: HyperPCTL formula.
Output: Whether $\mathcal{M}$ satisfies the input formula.

1 **Function** *Main*($\mathcal{M}$, $Q\hat{\sigma}.\ Q_1\hat{s}_1(\hat{\sigma}).\ldots.Q_n\hat{s}_n(\hat{\sigma}).\ \varphi^{nq}$)
2 $E := \bigwedge_{s\in\mathcal{S}}(\bigvee_{\alpha\in Act(s)} \sigma_s = \alpha)$ `// scheduler choice`
3 **if** Q *is existential* **then**
4 $E := E \wedge$ Semantics($\mathcal{M}$, φ^{nq}, n)
5 $E := E \wedge$ Truth($\mathcal{M}$, $\exists\hat{\sigma}.\ Q_1\hat{s}_1(\hat{\sigma}).\ldots.Q_n\hat{s}_n(\hat{\sigma}).\ \varphi^{nq}$)
6 **if** *check*(E) = *SAT* **then return** *TRUE*
7 **else return** *FALSE*
8 **else if** Q *is universal* **then**
// $\overline{Q}_i$ `is` $\forall$ `if` $Q_i = \exists$ `and` $\exists$ `else`
9 $E := E \wedge$ Semantics($\mathcal{M}, \neg\varphi^{nq}, n$)
10 $E := E \wedge$ Truth($\mathcal{M}$, $\exists\hat{\sigma}.\overline{Q}_1\hat{s}_1(\hat{\sigma}).\ldots.\overline{Q}_n\hat{s}_n(\hat{\sigma}).\neg\varphi^{nq}$)
11 **if** *check*(E) = *SAT* **then return** *FALSE*
12 **else return** *TRUE*

cidable, as there are only finitely many such schedulers that can be enumerated. Regarding complexity, we have the following property.

Theorem 5.3.1. The problem to decide for MDPs the truth of HyperPCTL formulas with a single existential (respectively, universal) scheduler quantifier over non-probabilistic memoryless schedulers is NP-complete (respectively, coNP-complete) in the state set size of the given MDP.

Next, we propose an SMT-based technique for solving the model checking problem for non-probabilistic memoryless scheduler domains, and for the simplified case of having a single scheduler quantifier; the general case for an arbitrary number of scheduler quantifiers is similar, but a bit more involved, so the simplified setting might be more suitable for understanding the basic ideas.

The main method listed in the Algorithm 7 constructs a formula E that is satisfiable if and only if the input MDP $\mathcal{M}$ satisfies the input HyperPCTL formula with a single scheduler quantifier over the non-probabilistic memoryless scheduler domain. Let us first deal with the case that the scheduler quantifier is *existential*. In line 2 we encode possible instantiations

σ for the scheduler variable $\hat{\sigma}$, for which we use a variable σ_s for each MDP state $s \in \mathcal{S}$ to encode which action is chosen in that state. In line 4 we encode the meaning of the quantifier-free inner part φ^{nq} of the input formula, whereas line 5 encodes the meaning of the state quantifiers, i.e. for which sets of composed states φ^{nq} needs to hold in order to satisfy the input formula. In lines 6–7 we check the satisfiability of the encoding and return the corresponding answer. Formulas with a *universal* scheduler quantifier $\forall\hat{\sigma}.\varphi$ are semantically equivalent to $\neg\exists\hat{\sigma}.\neg\varphi$. We make use of this fact in lines 8–12 to check first the satisfaction of encoding for $\exists\hat{\sigma}.\neg\varphi$ and return the inverted answer.

The Semantics method, shown in Algorithm 8, applies structural recursion to encode the meaning of its quantifier-free input formula. As variables, the encoding uses (1) propositions $prob_{\boldsymbol{s},\varphi^{nq}} \in \{0,1\}$ $holds_{\boldsymbol{s},\varphi^{nq}} \in \{\texttt{true}, \texttt{false}\}$ to encode the truth of each Boolean sub-formula φ^{nq} of the input formula in each state $\boldsymbol{s} \in \mathcal{S}^n$ of the n-ary self-composition of $\mathcal{M}$, (2) numeric variables $prob_{\boldsymbol{s},\varphi^{pr}} \in [0,1] \subseteq \mathbb{R}$ to encode the value of each probability expression φ^{pr} in the input formula in the context of each composed state $\boldsymbol{s} \in \mathcal{S}^n$, (3) variables $holdsToInt_{\boldsymbol{s},\varphi^{pr}} \in \{0,1\}$ to encode truth values in a pseudo-Boolean form, i.e. we set $holdsToInt_{\boldsymbol{s},\varphi^{pr}} = 1$ for $holds_{\boldsymbol{s},\varphi^{nq}} = \texttt{true}$ and $prob_{\boldsymbol{s},\varphi^{pr}} = 0$ else and (4) variables $d_{\boldsymbol{s},\varphi}$ to encode the existence of a loop-free path from state $\boldsymbol{s}$ to a state satisfying φ.

There are two base cases: the Boolean constant `true` holds in all states (line 2), whereas atomic propositions hold in exactly those states that are labeled by them (line 3). For conjunction (line 5) we recursively encode the truth values of the operands and state that the conjunction is true if and only if both operands are true. For negation (line 8) we again encode the meaning of the operand recursively and flip its truth value. For the comparison of two probability expressions (line 10), we recursively encode the probability values of the operands and state the respective relation between them for the satisfaction of the comparison.

The remaining cases encode the semantics of probability expressions. The cases for constants (line 22) and arithmetic operations (line 23) are straightforward. For the probability

Algorithm 8: SMT encoding for the meaning of the input formula

Input : $\mathcal{M} = (\mathcal{S}, Act, \mathbb{P}, \mathsf{AP}, L)$: MDP;
φ: quantifier-free HyperPCTL formula or expression;
n: number of state variables in φ.

Output: SMT encoding of the meaning of φ in the n-ary self-composition of $\mathcal{M}$.

1 **Function** *Semantics*($\mathcal{M}$, φ, n)
2 **if** φ *is* `true` **then** $E := \bigwedge_{\boldsymbol{s} \in \mathcal{S}^n} holds_{\boldsymbol{s},\varphi}$
3 **else if** φ *is* $a_{\hat{s}_i}$ **then**
4 $E := (\bigwedge_{\boldsymbol{s} \in \mathcal{S}^n,\ a \in L(s_i)} (holds_{\boldsymbol{s},\varphi})) \wedge (\bigwedge_{\boldsymbol{s} \in \mathcal{S}^n,\ a \notin L(s_i)} (\neg holds_{\boldsymbol{s},\varphi}))$;
5 **else if** φ *is* $\varphi_1 \wedge \varphi_2$ **then**
6 $E :=$ Semantics$(\mathcal{M}, \varphi_1, n) \wedge$ Semantics$(\mathcal{M}, \varphi_2, n) \wedge$
7 $\bigwedge_{\boldsymbol{s} \in \mathcal{S}^n} ((holds_{\boldsymbol{s},\varphi} \wedge holds_{\boldsymbol{s},\varphi_1} \wedge holds_{\boldsymbol{s},\varphi_2}) \vee (\neg holds_{\boldsymbol{s},\varphi} \wedge (\neg holds_{\boldsymbol{s},\varphi_1} \vee \neg holds_{\boldsymbol{s},\varphi_2})))$
8 **else if** φ *is* $\neg\varphi'$ **then**
9 $E :=$ Semantics$(\mathcal{M}, \varphi', n) \wedge \bigwedge_{\boldsymbol{s} \in \mathcal{S}^n} (holds_{\boldsymbol{s},\varphi} \oplus holds_{\boldsymbol{s},\varphi'})$
10 **else if** φ *is* $\varphi_1 < \varphi_2$ **then**
11 $E :=$ Semantics$(\mathcal{M}, \varphi_1, n) \wedge$ Semantics$(\mathcal{M}, \varphi_2, n) \wedge$
12 $\bigwedge_{\boldsymbol{s} \in \mathcal{S}^n} ((holds_{\boldsymbol{s},\varphi} \wedge prob_{\boldsymbol{s},\varphi_1} < prob_{\boldsymbol{s},\varphi_2}) \vee (\neg holds_{\boldsymbol{s},\varphi} \wedge prob_{\boldsymbol{s},\varphi_1} \geq prob_{\boldsymbol{s},\varphi_2}))$
13 **else if** φ *is* $\mathbb{P}(\bigcirc \varphi')$ **then**
14 $E :=$ Semantics$(\mathcal{M}, \varphi', n) \wedge$
15 $\bigwedge_{\boldsymbol{s} \in \mathcal{S}^n} \big((holdsToInt_{\boldsymbol{s},\varphi'} = 1 \wedge holds_{\boldsymbol{s},\varphi'}) \vee (holdsToInt_{\boldsymbol{s},\varphi'} = 0 \wedge \neg holds_{\boldsymbol{s},\varphi'})\big)$
16 **foreach** $\boldsymbol{s} = (s_1, \ldots, s_n) \in \mathcal{S}^n$ **do**
17 **foreach** $\vec{\alpha} = (\alpha_1, \ldots, \alpha_n) \in Act(s_1) \times \ldots \times Act(s_n)$ **do**
18 $E := E \wedge \big(\big[\bigwedge_{i=1}^{n} \sigma_{s_i} = \alpha_i\big] \rightarrow \big[prob_{\boldsymbol{s},\varphi} =$
19 $\sum_{\boldsymbol{s}' \in supp(\alpha_1) \times \ldots \times supp(\alpha_n)} ((\Pi_{i=1}^{n} \mathbb{P}(s_i, \alpha_i, s_i')) \cdot holdsToInt_{\boldsymbol{s}',\varphi'})\big]\big)$;
20 **else if** φ *is* $\mathbb{P}(\varphi_1 \mathcal{U} \varphi_2)$ **then** $E :=$ SemanticsUnboundedUntil$(\mathcal{M}, \varphi, n)$
21 **else if** φ *is* $\mathbb{P}(\varphi_1 \mathcal{U}^{[k_1,k_2]} \varphi_2)$ **then** $E :=$ SemanticsBoundedUntil$(\mathcal{M}, \varphi, n)$
22 **else if** φ *is* c **then** $E := \bigwedge_{\boldsymbol{s} \in \mathcal{S}^n} (prob_{\boldsymbol{s},\varphi} = c)$
23 **else if** φ *is* φ_1 *op* φ_2 /* *op* $\in \{+, -, *\}$ */ **then**
24 $E :=$ Semantics$(\mathcal{M}, \varphi_1, n) \wedge$ Semantics$(\mathcal{M}, \varphi_2, n) \wedge$
$\bigwedge_{\boldsymbol{s} \in \mathcal{S}^n} (prob_{\boldsymbol{s},\varphi} = (prob_{\boldsymbol{s},\varphi_1}\ op\ prob_{\boldsymbol{s},\varphi_2}))$;
25 **return** E;

$\mathbb{P}(\bigcirc \varphi')$ (line 13), we encode the Boolean value of φ' in the variables $holds_{\boldsymbol{s},\varphi'}$ (line 14), turn them into arithmetic pseudo-Boolean values $holdsToInt_{\boldsymbol{s},\varphi'}$ (1 for true and 0 for false, line 15), and state that for each composed state, the probability value of $\mathbb{P}(\bigcirc \varphi')$ is the sum of the probabilities to get to a successor state where the operand φ' holds; since the successors and their probabilities are scheduler-dependent, we need to iterate over all scheduler choices and use $supp(\alpha_i)$ to denote the support $\{s \in \mathcal{S} \mid \alpha_i(s) > 0\}$ of the distribution α_i (line 17). The encodings for the probabilities of unbounded until formulas (line 20) and bounded until

Algorithm 9: SMT encoding for the meaning of unbounded until formulas

Input : $\mathcal{M} = (\mathcal{S}, Act, \mathbb{P}, \mathsf{AP}, L)$: MDP; φ: HyperPCTL unbounded until formula of the form $\mathbb{P}(\varphi_1 \mathcal{U} \varphi_2)$; n: number of state variables in φ.

Output: SMT encoding of φ's meaning in the n-ary self-composition of $\mathcal{M}$.

1 **Function** *SemanticsUnboundedUntil*($\mathcal{M}$, $\varphi = \mathbb{P}(\varphi_1 \mathcal{U} \varphi_2)$, n)

2 $E :=$ Semantics($\mathcal{M}$, φ_1, n)$\wedge$ Semantics($\mathcal{M}$, φ_2, n)

3 **foreach** $\boldsymbol{s} = (s_1, \ldots, s_n) \in \mathcal{S}^n$ **do**

4 $E := E \wedge (holds_{\boldsymbol{s},\varphi_2} \rightarrow prob_{\boldsymbol{s},\varphi}{=}1) \wedge ((\neg holds_{\boldsymbol{s},\varphi_1} \wedge \neg holds_{\boldsymbol{s},\varphi_2}) \rightarrow prob_{\boldsymbol{s},\varphi}{=}0)$

5 **foreach** $\vec{\alpha} = (\alpha_1, \ldots, \alpha_n) \in Act(s_1) \times \ldots \times Act(s_n)$ **do**

6 $E := E \wedge \Big(\big[holds_{\boldsymbol{s},\varphi_1} \wedge \neg holds_{\boldsymbol{s},\varphi_2} \wedge \bigwedge_{i=1}^{n} \sigma_{s_i} = \alpha_i \big] \rightarrow$

7 $\big[prob_{\boldsymbol{s},\varphi} = \sum_{\boldsymbol{s}' \in supp(\alpha_1) \times \ldots \times supp(\alpha_n)} ((\Pi_{i=1}^{n} \mathbb{P}(s_i, \alpha_i, s_i')) \cdot prob_{\boldsymbol{s}',\varphi}) \wedge$

8 $(prob_{\boldsymbol{s},\varphi}{>}0 \rightarrow (\bigvee_{\boldsymbol{s}' \in supp(\alpha_1) \times \ldots \times supp(\alpha_n)} (holds_{\boldsymbol{s}',\varphi_2} \vee d_{\boldsymbol{s},\varphi_2} {>} d_{\boldsymbol{s}',\varphi_2})))\big] \Big)$

9 **return** E;

formulas (line 21) are listed in Algorithm 9 and 10, respectively.

For the probabilities $\mathbb{P}(\varphi_1 \mathcal{U} \varphi_2)$ to satisfy an unbounded until formula, the method SemanticsUnboundedUntil shown in Algorithm 9 first encodes the meaning of the until operands (line 2). For each composed state $\boldsymbol{s} \in \mathcal{S}^n$, the probability of satisfying the until formula in $\boldsymbol{s}$ is encoded in the variable $prob_{\boldsymbol{s},\mathbb{P}(\varphi_1 \mathcal{U} \varphi_2)}$. If the second until-operand φ_2 holds in $\boldsymbol{s}$ then this probability is 1 and if none of the operands are true in $\boldsymbol{s}$ then it is 0 (line 4). Otherwise, depending on the scheduler σ of $\mathcal{M}$ (line 5), the value of $prob_{\boldsymbol{s},\mathbb{P}(\varphi_1 \mathcal{U} \varphi_2)}$ is a sum, adding up for each successor state $\boldsymbol{s}'$ of $\boldsymbol{s}$ the probability to get from $\boldsymbol{s}$ to $\boldsymbol{s}'$ in one step times the probability to satisfy the until-formula on paths starting in $\boldsymbol{s}'$ (line 7). However, these encodings work only when at least one state satisfying φ_2 is reachable from $\boldsymbol{s}$ with a positive probability: for any bottom SCC whose states all violate φ_2, the probability $\mathbb{P}(\varphi_1 \mathcal{U} \varphi_2)$ is obviously 0, however, assigning any fixed value from $[0, 1]$ to all states of this bottom SCC would yield a fixed-point for the underlying equation system. To assure correctness, in line 8 we enforce smallest fixed-points by requiring that if $prob_{\boldsymbol{s},\mathbb{P}(\varphi_1 \mathcal{U} \varphi_2)}$ is positive then there exists a loop-free path from $\boldsymbol{s}$ to any state satisfying φ_2. In the encoding of this property we use fresh variables $d_{\boldsymbol{s},\varphi_2}$ and require a path over states with strong monotonically decreasing $d_{\boldsymbol{s},\varphi_2}$-values to a φ_2-state (where the decreasing property serves to exclude loops).

Algorithm 10: SMT encoding for the meaning of bounded until formulas

Input : $\mathcal{M} = (\mathcal{S}, Act, \mathbb{P}, \mathsf{AP}, L)$: MDP; φ: HyperPCTL bounded until formula of the form $\mathbb{P}(\varphi_1 \mathcal{U}^{[k_1,k_2]} \varphi_2)$; n: number of state variables in φ.

Output: SMT encoding of φ's meaning in the n-ary self-composition of $\mathcal{M}$.

1 **Function** *SemanticsBoundedUntil*($\mathcal{M}$, $\varphi = \mathbb{P}(\varphi_1 \mathcal{U}^{[k_1,k_2]} \varphi_2)$, n)

2 | **if** $k_2 = 0$ **then**

3 | | $E :=$ Semantics$(\mathcal{M}, \varphi_1, n) \wedge$ Semantics$(\mathcal{M}, \varphi_2, n)$

4 | | **foreach** $\boldsymbol{s} = (s_1, \ldots, s_n) \in \mathcal{S}^n$ **do**

5 | | | $E := E \wedge (holds_{\boldsymbol{s},\varphi_2} \rightarrow prob_{\boldsymbol{s},\varphi} = 1) \wedge (\neg holds_{\boldsymbol{s},\varphi_2} \rightarrow prob_{\boldsymbol{s},\varphi} = 0)$

6 | **else if** $k_1 = 0$ **then**

7 | | $E :=$ SemanticsBoundedUntil($\mathcal{M}$, $\mathbb{P}(\varphi_1 \mathcal{U}^{[0,k_2-1]} \varphi_2), n)$

8 | | **foreach** $\boldsymbol{s} = (s_1, \ldots, s_n) \in \mathcal{S}^n$ **do**

9 | | | $E := E \wedge (holds_{\boldsymbol{s},\varphi_2} \rightarrow prob_{\boldsymbol{s},\varphi} = 1) \wedge ((\neg holds_{\boldsymbol{s},\varphi_1} \wedge \neg holds_{\boldsymbol{s},\varphi_2}) \rightarrow prob_{\boldsymbol{s},\varphi} = 0)$

10 | | | **foreach** $\vec{\alpha} = (\alpha_1, \ldots, \alpha_n) \in Act(s_1) \times \ldots \times Act(s_n)$ **do**

11 | | | | $E := E \wedge \Big(\big[\; holds_{\boldsymbol{s},\varphi_1} \wedge \neg holds_{\boldsymbol{s},\varphi_2} \wedge \bigwedge_{i=1}^{n} \sigma_{s_i} = \alpha_i \; \big] \rightarrow \big[prob_{\boldsymbol{s},\varphi} =$

12 | | | | $\sum_{\boldsymbol{s}' \in supp(\alpha_1) \times \ldots \times supp(\alpha_n)} ((\Pi_{i=1}^{n} \mathbb{P}(s_i, \alpha_i, s_i')) \cdot prob_{\boldsymbol{s}', \mathbb{P}(\varphi_1 \mathcal{U}^{[0,k_2-1]} \varphi_2)}) \; \big] \Big)$

13 | **else if** $k_1 > 0$ **then**

14 | | $E :=$ SemanticsBoundedUntil($\mathcal{M}$, $\mathbb{P}(\varphi_1 \mathcal{U}^{[k_1-1,k_2-1]} \varphi_2), n)$

15 | | **foreach** $\boldsymbol{s} = (s_1, \ldots, s_n) \in \mathcal{S}^n$ **do**

16 | | | $E := E \wedge (\neg holds_{\boldsymbol{s},\varphi_1} \rightarrow prob_{\boldsymbol{s},\varphi} = 0)$

17 | | | **foreach** $\vec{\alpha} = (\alpha_1, \ldots, \alpha_n) \in Act(s_1) \times \ldots \times Act(s_n)$ **do**

18 | | | | $E := E \wedge \Big(\big[\; holds_{\boldsymbol{s},\varphi_1} \wedge \bigwedge_{i=1}^{n} \sigma_{s_i} = \alpha_i \; \big] \rightarrow \big[\; prob_{\boldsymbol{s},\varphi} =$

19 | | | | $\sum_{\boldsymbol{s}' \in supp(\alpha_1) \times \ldots \times supp(\alpha_n)} ((\Pi_{i=1}^{n} \mathbb{P}(s_i, \alpha_i, s_i')) \cdot prob_{\boldsymbol{s}', \mathbb{P}(\varphi_1 \mathcal{U}^{[k_1-1,k_2-1]} \varphi_2)}) \; \big] \Big)$

20 | **return** E;

The domain of the distance-variables $d_{\boldsymbol{s},\varphi_2}$ can be e.g. integers, rationals or reals; the only restriction is that is should contain at least $|\mathcal{S}|^n$ ordered values. Especially, it does not need to be lower bounded (note that each solution assigns to each $d_{\boldsymbol{s},\varphi_2}$ a fixed value, leading a finite number of distance values).

The SemanticsBoundedUntil method, listed in Algorithm 10, encodes the probability $\mathbb{P}(\varphi_1 \mathcal{U}^{[k_1,k_2]} \varphi_2)$ of a bounded until formula in the numeric variables $prob_{\boldsymbol{s},\mathbb{P}(\varphi_1 \mathcal{U}^{[k_1,k_2]} \varphi_2)}$ for all (composed) states $\boldsymbol{s} \in \mathcal{S}^n$ and recursively reduced time bounds. There are three main cases: (i) the satisfaction of $\varphi_1 \mathcal{U}^{[0,k_2-1]} \varphi_2$ requires to satisfy φ_2 immediately (lines 2–5); (ii) $\varphi_1 \mathcal{U}^{[0,k_2-1]} \varphi_2$ can be satisfied by either satisfying φ_2 immediately or satisfying it later, but in the latter case φ_1 needs to hold currently (lines 6–12); (iii) φ_1 has to hold and φ_2 needs to

Algorithm 11: SMT encoding of the truth of the input formula

Input : $\mathcal{M} = (\mathcal{S}, Act, \mathbb{P}, \mathsf{AP}, L)$: MDP;
$\exists\hat{\sigma}.Q_1\hat{s}_1(\hat{\sigma}).\ldots.Q_n\hat{s}_n(\hat{\sigma}).\varphi^{nq}$: HyperPCTL formula.
Output: Encoding of the truth of the input formula in $\mathcal{M}$.

```
1 Function Truth(M, ∃σ̂. Q_1 ŝ_1(σ̂). ... Q_n ŝ_n(σ̂). φ^nq)
2     foreach i = 1, ..., n do
3         if Q_i = ∀ then B_i :="⋀_{s_i ∈ S}"
4         else B_i :="⋁_{s_i ∈ S}"
5     return B_1 ... B_n holds_{(s_1,...,s_n),φ^nq};
```

be satisfied sometime later (lines 13–19). To avoid the repeated encoding of the semantics of the operands, we do it only when we reach case (i) where recursion stops (line 3). For the other cases, we recursively encode the probability to reach a φ_2-state over φ_1 states where the deadlines are reduced with one step (lines 7 resp. 14) and use these to fix the values of the variables $prob_{\boldsymbol{s},\mathbb{P}(\varphi_1\mathcal{U}^{[k_1,k_2]}\varphi_2)}$, similarly to the unbounded case but under additional consideration of time bounds.

Finally, the Truth method listed in Algorithm 11 encodes the meaning of the state quantification: it states for each universal quantifier that instantiating it with any MDP state should satisfy the formula (conjunction over all states in line 3), and for each existential state quantification that at least one state should lead to satisfaction (disjunction in line 4).

Theorem 5.3.2. Algorithm 7 returns a formula that is true iff its input HyperPCTL formula is satisfied by the input MDP.

We note that the satisfiability of the generated SMT encoding for a formula with an existential scheduler quantifier does not only prove the truth of the formula but provides also a scheduler as a witness, encoded in the solution of the SMT encoding. Conversely, the unsatisfiability of the SMT encoding for a formula with a universal scheduler quantifier provides a counterexample scheduler.

5.3.1 Example of the Encoding

Consider the problem where we are trying to verify the following property on the MDP in Fig. 4.2a.

$$\exists\hat{\sigma}(\hat{\mathcal{M}}).\forall\hat{s}(\hat{\mathcal{M}}^{\hat{\sigma}}).\exists\hat{s}'(\hat{\mathcal{M}}^{\hat{\sigma}}).\ \Big((h{>}0)_{\hat{s}}\wedge(h{\leq}0)_{\hat{s}'}\Big)\rightarrow\Big(\mathbb{P}(\Diamond\,(l{=}1)_{\hat{s}}){=}\mathbb{P}(\Diamond\,(l{=}2)_{\hat{s}'})\Big) \tag{5.2}$$

We encode the actions in the MDP as described in line 2 of Algorithm 7. Here, σ_i refers to the action in s_i.

$$E_{sch} = (\sigma_0 = \alpha \vee \sigma_0 = \beta) \wedge (\sigma_1 = \alpha \vee \sigma_1 = \beta) \wedge (\sigma_2 = \tau) \wedge (\sigma_3 = \tau) \tag{5.3}$$

We handle the encoding of the state quantifiers using Algorithm 11. We use φ_{nq} to represent the quantifier-free part of the above property and $holds_{s_i,s_j,\varphi_{nq}}$ refers to the encoding to ensure φ_{nq} holds in the composed state of (s_i, s_j).

$$\begin{aligned} E_{truth} = (&holds_{s_0,s_0,\varphi_{nq}} \vee \ldots \vee holds_{s_0,s_3,\varphi_{nq}}) \wedge \\ &\ldots \wedge (holds_{s_3,s_0,\varphi_{nq}} \vee \ldots \vee holds_{s_3,s_3,\varphi_{nq}}) \end{aligned} \tag{5.4}$$

We handle atomic propositions as described in line 3 in Algorithm 8. For example, we have the encoding of $(h{>}0)_{\hat{s}}$ below. Please note here that the first quantifier is relevant and the second is not. Also, $(h{>}0)_{\hat{s}}$ is true only in s_0, hence we encode the atomic proposition with a negation for all other states.

$$E_{(h>0)_{\hat{s}}} = (holds_{s_0,s_0,(h>0)_{\hat{s}}}) \wedge (\neg holds_{s_1,s_0,(h>0)_{\hat{s}}} \wedge \ldots \wedge \neg holds_{s_3,s_0,(h>0)_{\hat{s}}}) \tag{5.5}$$

To encode operators, we include both the satisfaction and dissatisfaction clauses in the encoding. Depending on the atomic propositions involved, one of the clauses would be satisfied. For example, the encoding for conjunction of $\Big((h{>}0)_{\hat{s}}\wedge(h{\leq}0)_{\hat{s}'}\Big)$ for (s_0, s_0) would

be as follows.

$$E_{conj} = (holds_{s_0,s_0,(h>0)_{\hat{s}}} \wedge holds_{s_0,s_0,(h\leq 0)_{\hat{s}'}} \wedge holds_{s_0,s_0,(h>0)_{\hat{s}}\wedge(h\leq 0)_{\hat{s}'}}) \vee$$
$$((\neg holds_{s_0,s_0,(h>0)_{\hat{s}}} \vee \neg holds_{s_0,s_0,(h\leq 0)_{\hat{s}'}}) \wedge \neg holds_{s_0,s_0,(h>0)_{\hat{s}}\wedge(h\leq 0)_{\hat{s}'}}) \tag{5.6}$$

The encoding of $\diamondsuit$ is similar to Algorithm 9 except that we consider φ_1 to be `true` and ignore its encoding. For example, the encoding of $\mathbb{P}(\diamondsuit\,(l{=}1)_{\hat{s}})$ for (s_0, s_0) and action $(\alpha_{\hat{s}}, \alpha_{\hat{s}'})$ would be as below. Note that since the first quantifier is relevant, we will only encode (s_2, s_0) and (s_3, s_0) as the successor states.

$$\begin{aligned} E_{\diamondsuit} =& (holds_{s_0,s_0,(l=1)_{\hat{s}}} \rightarrow prob_{s_0,s_0,\mathbb{P}(\diamondsuit\,(l=1)_{\hat{s}})} = 1) \wedge (prob_{s_0,s_0,\mathbb{P}(\diamondsuit\,(l=1)_{\hat{s}})} \geq 0) \wedge \\ & (\neg holds_{s_0,s_0,(l=1)_{\hat{s}}} \wedge \sigma_0 = \alpha \wedge \sigma_1 = \alpha) \rightarrow \Big(prob_{s_0,s_0,\mathbb{P}(\diamondsuit\,(l=1)_{\hat{s}})} = \\ & (3/4 \times 1 \times prob_{s_2,s_0,\mathbb{P}(\diamondsuit\,(l=1)_{\hat{s}})}) + (1/4 \times 1 \times prob_{s_3,s_0,\mathbb{P}(\diamondsuit\,(l=1)_{\hat{s}})})\Big) \wedge \\ & \Big(prob_{s_0,s_0,\mathbb{P}(\diamondsuit\,(l=1)_{\hat{s}})} > 0 \rightarrow (holds_{s_2,s_0,(l=1)_{\hat{s}}} \vee d_{s_0,s_0,(l=1)_{\hat{s}}} > d_{s_2,s_0,(l=1)_{\hat{s}}}) \\ & \vee (holds_{s_3,s_0,(l=1)_{\hat{s}}} \vee d_{s_0,s_0,(l=1)_{\hat{s}}} > d_{s_3,s_0,(l=1)_{\hat{s}}})\Big) \end{aligned} \tag{5.7}$$

For multiple scheduler quantifiers: To extend this algorithm to work for multiple schedulers, we first have to encode the combination of actions for every state combination. Hence in line 2 of Algorithm 7, we will need n nested loops to encode all the possible scheduler choices of n scheduler quantifiers. In the rest of the algorithm, we have to make similar changes to account for the combinations of actions possible from each state $\boldsymbol{s} \in \mathcal{S}^n$. So we will need similar nested loops in cases like line 5 of Algorithm 9.

5.4 Evaluation

We developed a prototypical implementation of our algorithm in Python, with the help of several libraries. There is extensive use of STORMPY [stob, DJKV17], which provides an efficient solution to parsing, building, and storage of MDPs. We used the SMT-solver Z3 [dMB08] to solve the logical encoding generated by Algorithm 7. All of our experiments

were run on a MacBook Pro laptop with a 2.3GHz i7 processor with 32GB of RAM. The results are presented in Table 5.1.

As the first case study, we model and analyse information leakage in the modular exponentiation algorithm (function `modexp` in Fig. 4.4); the corresponding results in Table 5.1 are marked by **TA**. We experimented with 1, 2, and 3 bits for the encryption key (hence, $m \in \{2, 4, 6\}$). The specification checks whether there is a timing channel for all possible schedulers, which is the case for the implementation in `modexp`.

Our second case study is the verification of password leakage through the string comparison algorithm (function `str_cmp` in Fig 4.4). Here, we also experimented with $m \in \{2, 4, 6\}$; results in Table 5.1 are denoted by **PW**.

In our third case study, we assume two concurrent processes. The first process decrements the value of a secret h by 1 as long as the value is still positive, and after this it sets a low variable l to 1. A second process just sets the value of the same low variable l to 2. The two threads run in parallel; as long as none of them terminated, a fair scheduler chooses for each CPU cycle the next executing thread. As discussed in Section 4.1, this MDP opens a probabilistic thread scheduling channel and leaks the value of h. We denote this case study by **TS** in Table 5.1, and compare observations for executions with different secret values h_1 and h_2 (denoted as $h = (h_1, h_2)$ in the table). There is an interesting relation between the execution times for **TA** and **TS**. For example, although the MDP for **TA** with $m = 4$ has 60 reachable states and the MDP for **TS** comparing executions for $h = (0, 15)$ has 35 reachable states, verification of **TS** takes 20 times more than **TA**. We believe this is because the MDP of **TS** is twice deeper than the MDP of **TA**, making the SMT constraints more complex.

Our last case study is on probabilistic conformance, denoted **PC**. The input is a DTMC that encodes the behaviour of a 6-sided die as well as a structure of actions having probability distributions with two successor states each; these transitions can be pruned using a scheduler to obtain a DTMC which simulates the die outcomes using a fair coin. Given a fixed state space, we experiment with different numbers of transitions. In particular, we started from the

Case study		Running time (s)			#SMT variables	#subformulas	#states	#transitions
		SMT encoding	**SMT solving**	**Total**				
TA	$m = 2$	5.43	0.31	5.74	8088	50654	24	46
	$m = 4$	114.00	20.00	134.00	50460	368062	60	136
	$m = 6$	1721.00	865.00	2586.00	175728	1381118	112	274
PW	$m = 2$	5.14	0.30	8.14	8088	43432	24	46
	$m = 4$	207.00	40.00	247.00	68670	397852	70	146
	$m = 6$	3980.00	1099.00	5079.00	274540	1641200	140	302
TS	$h = (0, 1)$	0.83	0.07	0.90	1379	7913	7	13
	$h = (0, 15)$	60.00	1607.00	1667.00	34335	251737	35	83
	$h = (4, 8)$	11.86	17.02	28.88	12369	87097	21	48
	$h = (8, 15)$	60.00	1606.00	1666.00	34335	251737	35	83
PC	s=(0)	277.00	1996.00	2273.00	21220	1859004	20	158
	s=(0,1)	822.00	5808.00	6630.00	21220	5349205	20	280
	s=(0,1,2)	1690.00	58095.00	59785.00	21220	11006581	20	404

Table 5.1: Experimental results. **TA:** Timing attack. **PW:** Password leakage. **TS:** Thread scheduling. **PC:** Probabilistic conformance.

implementation in [KY76] and then we added all the possible nondeterministic transitions from the first state to all the other states (s=0), from the first and second states to all the others (s=0,1), and from the first, second, and third states to all the others (s=0,1,2). Each time we were able not only to satisfy the formula, but also to obtain the witness corresponding to the scheduler satisfying the property.

Regarding the running times listed in Table 5.1, we note that our implementation is only prototypical and there are possibilities for numerous optimizations. Most importantly, for purely existentially or purely universally quantified formulas, we could define a more efficient encoding with much less variables. However, it is clear that the running times for even relatively small MDPs are large. This is simply because of the high complexity of the verification of hyperproperties. In addition, the HyperPCTL formulas in our case studies have multiple scheduler and/or state quantifiers, making the problem significantly more difficult.

5.5 Summary

One important gap in HyperPCTL was the inability to express properties involving nondeterminism in systems. In this chapter, we have discussed the reasoning of hyperproperties

over schedulers. This new dimension added an extra layer of complexity due to the increase in expressiveness of the language. In terms of practical results, we have proposed an NP-complete algorithm for the logic which only allows reasoning over memoryless and deterministic schedulers. We have also demonstrated our approach in a few interesting case studies. However, the logic still had an important gap, in terms of its inability to argue over reward models, which caused us to further explore it as described in the next chapter.

Chapter 6

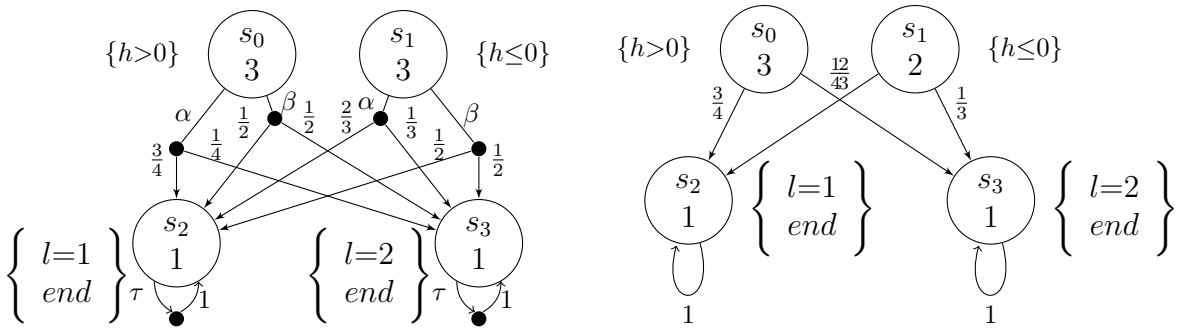
Probabilistic Hyperproperties with Rewards

6.1 Introduction

Stochastic phenomena appear in many systems such as those that interact with the physical environment (e.g., due to environmental uncertainties, thermal fluctuations, random message loss, and processor failure). Traditionally, system specifications that deal with uncertainties are expressed in some form of probabilistic temporal logic such as PCTL and PCTL* [BK08]. These logics can express the properties of *single* probabilistic computation trees. The temporal logic HyperPCTL generalizes PCTL to express *probabilistic hyperproperties* by allowing quantification over multiple computation trees and expressing the probability relation among them. For instance, consider the Markov Decision Process (MDP) in Fig. 6.1a. The HyperPCTL formula

$$\forall\hat{\sigma}.\forall\hat{s}(\hat{\sigma}).\forall\hat{s}'(\hat{\sigma}).\ \Big((h>0)_{\hat{s}} \wedge (h\leq 0)_{\hat{s}'}\Big) \Rightarrow \Big(\mathbb{P}\Diamond(l=1)_{\hat{s}} = \mathbb{P}\Diamond(l=1)_{\hat{s}'}\Big) \tag{6.1}$$

requires that the probability of reaching a state with proposition $l = 1$ from any pair of states $\hat{s}$ and $\hat{s}'$ labelled by $h > 0$ and $h \leq 0$ respectively, should be equal for the Discrete Time Markov Chain (DTMC) induced by any scheduler $\hat{\sigma}$. In addition to the probability relation between certain events and computations, it is natural to analyse the average behaviour of



(a) An MDPR. (b) An induced DTMCR.

Figure 6.1: Example Markov models.

Markov models as well as the interrelation of average behaviours in different executions. For example:

- *Service-level agreements* (e.g., average system response time and uptime) are generally concerned with the average performance metrics of a system among a set of executions. This is, of course, a system-wide performance requirement rather than the property of individual executions.

- *Side-channel timing* leaks can potentially reveal sensitive information through the execution time of a function call. The execution time can be captured as a reward model where each instruction is associated with a cost and the probabilistic hyperproperty expresses that every pair of executions should exhibit the same expected execution cost.

- *Distributed algorithms* often use randomization to break symmetry to tackle impossibility results. Although one can reason about the expected performance of a randomized distributed algorithm by the traditional reward models, from a design perspective, it is desirable to determine and mitigate states from where convergence to the objective of the algorithm takes much longer than others.

These examples motivate the need to somehow augment probabilistic hyperproperties with reward constraints.

With this motivation, our first contribution is to make the connection between reward models and probabilistic hyperproperties. In the context of a hyperproperty, analogous to the probability relation between multiple executions in a HyperPCTL formula, a reward mechanism should be able to express the expected reward relation along different quantified computation trees. To this end, we extend the syntax and semantics of HyperPCTL by allowing arithmetic functions over expected rewards and comparing them over multiple executions. For instance, for the MDP in Fig. 6.1a one may express whether there exist two schedulers such that starting from any two states, labeled with $h{>}0$ and $h{\leq}0$, resp., the expected reward of reaching an *end*-labeled state is the same using the following property:

$$\exists\hat{\sigma}_1.\exists\hat{\sigma}_2.\forall\hat{s}(\hat{\sigma}_1).\forall\hat{s}'(\hat{\sigma}_2).\Big((h{>}0)_{\hat{s}} \wedge (h{\leq}0)_{\hat{s}'}\Big) \rightarrow \Big(\mathcal{R}_{\hat{s}}(\diamondsuit\, end_{\hat{s}}) = \mathcal{R}_{\hat{s}'}(\diamondsuit\, end_{\hat{s}'})\Big) \tag{6.2}$$

In the MDP in Fig. 6.1a, if we instantiate $\hat{s}$ with s_0, and choose the action α, we collect a reward of $(3 + \frac{3}{4} \times 1 + \frac{1}{4} \times 1) = 4$, on reaching s_2 and s_3 with label *end*. Similarly, if we instantiate $\hat{s}'$ with s_1, and choose the action α, we collect a reward of $(3 + \frac{2}{3} \times 1 + \frac{1}{3} \times 1) = 4$, on reaching s_2 and s_3 with label *end*. Hence, we can prove the existence of schedulers that satisfy the above property in the MDP in Fig. 6.1a. On a closer look, no matter which action we choose at s_0 and s_1, the property is always satisfied. Also, if we instantiate $\hat{s}$ and $\hat{s}'$ with any other states different from s_0 resp. s_1, the property is vacuously true. On the contrary, if we replace the equality of rewards with inequality then the property is false as there are no such schedulers. Besides comparing reward values, our HyperPCTL extension offers further expressive power to e.g. measure accumulated rewards in an execution until an observable property, say termination, gets satisfied in another one.

Our second contribution is an algorithm for model checking HyperPCTL formulas with rewards for MDPs. Since the general verification problem is shown to be undecidable, we focus on memoryless non-probabilistic schedulers which yield a decidable problem, for which we propose a model checking algorithm based on logical problem encoding and SMT solving. We have implemented a prototype of our method and analysed it experimentally on three

case studies: (1) side-channel timing attacks, (2) probabilistic performance conformance, and (3) randomized path planning for multi-agent robotics applications.

6.2 HyperPCTL with Rewards

In this section, we elaborate on the syntax and semantics of the newly added reward-based operators in HyperPCTL. We additionally discuss the cases where we encounter undecidability of rewards and how we can avoid them in specific cases.

6.2.1 Syntax of HyperPCTL with Rewards

Hyperproperties of executions in an MDPR can be specified using the logic HyperPCTL. As shown in Fig. 6.2, a *quantified formula* ϕ^{q} starts with a sequence of quantifiers over scheduler variables $\hat{\sigma} \in \hat{\Sigma}$, fixing the schedulers under which executions are considered. Inside, a *state-quantified formula* ϕ^{sq} defines a sequence of quantifiers over state variables $\hat{s} \in \hat{\mathcal{S}}$, where each quantifier specifies a new execution from a given state under a given scheduler. Note that different executions might use the same scheduler.

In the scope of these quantifiers is a *non-quantified state formula* ϕ^{nq}, which can be the constant `true`, an atomic proposition indexed with a state variable, conjunction, negation, or a relational constraint comparing two arithmetic expressions via $\sim \in \{>, \geq, =, \neq, <, \leq\}$.

$$
\begin{array}{lcl}
\phi^{q} & ::= & \forall \hat{\sigma}.\phi^{q} \mid \exists \hat{\sigma}.\phi^{q} \mid \phi^{sq} \\
\phi^{sq} & ::= & \forall \hat{s}(\hat{\sigma}).\phi^{sq} \mid \exists \hat{s}(\hat{\sigma}).\phi^{sq} \mid \phi^{nq} \\
\phi^{nq} & ::= & \texttt{true} \mid a_{\hat{s}} \mid \phi^{nq} \wedge \phi^{nq} \mid \neg\phi^{nq} \mid \phi^{ar} \sim \phi^{ar} \\
\phi^{ar} & ::= & \mathbb{P}(\phi^{path}) \mid \mathcal{R}_{\hat{s},i}(\phi^{path}) \mid f(\phi^{ar}, \ldots, \phi^{ar}) \\
\phi^{path} & ::= & \bigcirc \phi^{nq} \mid \phi^{nq}\, \mathcal{U}\, \phi^{nq} \mid \phi^{nq}\, \mathcal{U}^{[k_1,k_2]}\, \phi^{nq}
\end{array}
$$

Figure 6.2: HyperPCTL syntax.

Arithmetic expressions are constructed from probability expressions, reward expressions, or applying arithmetic function symbols (e.g., addition, subtraction, multiplication, etc., where constants are 0-ary functions) to arithmetic expressions. Note that the reward operator $\mathcal{R}$ is indexed with a state variable $\hat{s}$ specifying the execution for which we consider the reward,

and an integer i specifying the reward component; for models with unary rewards, like in our examples, we skip the second index (as it is always 0). Finally, the parameters of probabilistic and reward expressions are *path formulas*, which apply one of the temporal operators, next ($\bigcirc$), unbounded until ($\mathcal{U}$), or bounded until ($\mathcal{U}^{[k_1,k_2]}$, $k_1 \leq k_2 \in \mathbb{N}_{\geq 0}$) to non-quantified state formulas.

A HyperPCTL *formula* is a quantified formula in that every occurrence of an indexed atomic proposition $a_{\hat{s}}$ is in the scope of a state quantifier for $\hat{s}(\hat{\sigma})$, which in turn is in the scope of a scheduler quantifier for $\hat{\sigma}$. W.l.o.g., in the following we assume that each scheduler or state variable is quantified at most once. In addition to standard syntactic sugar $\vee, \rightarrow, \Diamond, \Box, \ldots$, we can express expected cumulative reward over the next $t \in \mathbb{N}$ steps and expected reward in the state reached after t steps as follows:

$$\mathcal{R}_{\hat{s},i}(C_t) = \mathcal{R}_{\hat{s},i}(\texttt{true}\mathcal{U}^{[t,t]}\texttt{true}) \text{ and } \mathcal{R}_{\hat{s},i}(I_t) = \begin{cases} \mathcal{R}_{\hat{s},i}(C_t) - \mathcal{R}_{\hat{s},i}(C_{t-1}) & \text{if } t{>}0 \\ \mathcal{R}_{\hat{s},i}(C_t) & else\ . \end{cases} \tag{6.3}$$

6.2.2 Semantics of HyperPCTL with Rewards

HyperPCTL formulas are evaluated recursively in the context of an MDPR $\mathcal{M}$, a sequence $\boldsymbol{\sigma}$ of actions, and a sequence $\boldsymbol{s}$ of states, both of the same length. Intuitively, the length of these sequences says how many executions we run in parallel, and the ith elements in these sequences specify the ith execution of the scheduler and the initial state in the induced DTMCR, respectively. An MDPR $\mathcal{M}$ satisfies a HyperPCTL formula ϕ (written $\mathcal{M} \models \phi$) iff $\mathcal{M}, (), () \models \phi$.

In the semantic rules shown in Fig. 6.3, the substitution $\varphi[\hat{\sigma} \rightsquigarrow \sigma]$ remembers the instantiation of a scheduler variable $\hat{\sigma}$ by a concrete scheduler $\sigma = (Q, act, mode, init)$ through syntactically transforming in φ each $\forall \hat{s}(\hat{\sigma})$ and $\exists \hat{s}(\hat{\sigma})$ into $\forall \hat{s}(\sigma)$ and $\exists \hat{s}(\sigma)$, resp. When instantiating the nth state quantifier $\forall \hat{s}(\sigma)$ or $\exists \hat{s}(\sigma)$ by a state s, we "start" an nth execution in state $(init(s), s)$ of $\mathcal{M}^\sigma$, which corresponds to extending the previously $(n{-}1)$-ary self-composition of $\mathcal{M}$ to *arity* n. We remember this by adding σ and s at the end of the

corresponding sequences in the context (using concatenation $\circ$), and applying the substitution $\varphi[\hat{s}{\rightsquigarrow}n]$ to replace each indexed atomic proposition $a_{\hat{s}}$ and each reward operator $\mathcal{R}_{\hat{s},i}$ in φ by a_n and $\mathcal{R}_{n,i}$, respectively.[1] We recall from Chapter 4 the semantics of constructs that are not related to rewards:

$$
\begin{array}{lll}
\mathcal{M},\boldsymbol{\sigma},\boldsymbol{s} \models \forall\hat{\sigma}.\varphi & \textit{iff} & \mathcal{M},\boldsymbol{\sigma},\boldsymbol{s} \models \varphi[\hat{\sigma}{\rightsquigarrow}\sigma] \textit{ for all } \sigma \in \Sigma^{\mathcal{M}} \\
\mathcal{M},\boldsymbol{\sigma},\boldsymbol{s} \models \exists\hat{\sigma}.\varphi & \textit{iff} & \mathcal{M},\boldsymbol{\sigma},\boldsymbol{s} \models \varphi[\hat{\sigma}{\rightsquigarrow}\sigma] \textit{ for some } \sigma \in \Sigma^{\mathcal{M}} \\
\mathcal{M},\boldsymbol{\sigma},\boldsymbol{s} \models \forall\hat{s}(\sigma).\varphi & \textit{iff} & \mathcal{M},\boldsymbol{\sigma}\circ\sigma,\boldsymbol{s}\circ(\mathit{init}(s),s) \models \varphi[\hat{s}{\rightsquigarrow}|\boldsymbol{\sigma}|] \textit{ for all } s{\in}\mathcal{S} \\
\mathcal{M},\boldsymbol{\sigma},\boldsymbol{s} \models \exists\hat{s}(\sigma).\varphi & \textit{iff} & \mathcal{M},\boldsymbol{\sigma}\circ\sigma,\boldsymbol{s}\circ(\mathit{init}(s),s) \models \varphi[\hat{s}{\rightsquigarrow}|\boldsymbol{\sigma}|] \textit{ for some } s{\in}\mathcal{S} \\
\mathcal{M},\boldsymbol{\sigma},\boldsymbol{s} \models \texttt{true} & & \\
\mathcal{M},\boldsymbol{\sigma},\boldsymbol{s} \models a_i & \textit{iff} & a_i \in L^{\boldsymbol{\sigma}}(\boldsymbol{s}) \\
\mathcal{M},\boldsymbol{\sigma},\boldsymbol{s} \models \varphi_1 \wedge \varphi_2 & \textit{iff} & \mathcal{M},\boldsymbol{\sigma},\boldsymbol{s} \models \varphi_1 \textit{ and } \mathcal{M},\boldsymbol{\sigma},\boldsymbol{s} \models \varphi_2 \\
\mathcal{M},\boldsymbol{\sigma},\boldsymbol{s} \models \neg\varphi & \textit{iff} & \mathcal{M},\boldsymbol{\sigma},\boldsymbol{s} \not\models \varphi \\
\mathcal{M},\boldsymbol{\sigma},\boldsymbol{s} \models \varphi_1^{ar} \sim \varphi_2^{ar} & \textit{iff} & [\![\varphi_1^{ar}]\!]_{\mathcal{M},\boldsymbol{\sigma},\boldsymbol{s}} \sim [\![\varphi_2^{ar}]\!]_{\mathcal{M},\boldsymbol{\sigma},\boldsymbol{s}} \\
[\![\mathbb{P}(\varphi_{path})]\!]_{\mathcal{M},\boldsymbol{\sigma},\boldsymbol{s}} & = & Pr^{\mathcal{M}^{\boldsymbol{\sigma}}}\big(\{\pi \in Paths^{\boldsymbol{s}}(\mathcal{M}^{\boldsymbol{\sigma}}) \mid \mathcal{M},\boldsymbol{\sigma},\pi \models \varphi_{path}\}\big) \\
[\![f(\varphi_1^{ar},\ldots,\varphi_k^{ar})]\!]_{\mathcal{M},\boldsymbol{\sigma},\boldsymbol{s}} & = & f\big([\![\varphi_1^{ar}]\!]_{\mathcal{M},\boldsymbol{\sigma},\boldsymbol{s}},\ \ldots,\ [\![\varphi_k^{ar}]\!]_{\mathcal{M},\boldsymbol{\sigma},\boldsymbol{s}}\big)
\end{array}
$$

Figure 6.3: Existing Semantics rules for HyperPCTL.

We describe the semantics for $\mathcal{R}_{j,i}(\varphi_{path})$ in Fig. 6.4 (note that instantiating a state quantifier for $\hat{s}(\sigma)$ replaces each $\mathcal{R}_{\hat{s},i}$ occurrence by $\mathcal{R}_{j,i}$, where j is the position of the quantifier). The value of $\mathcal{R}_{j,i}(\bigcirc\varphi^{nq})$ is the current ith reward plus the expected ith reward of the successor state in the jth execution if the probability that the successor state satisfies φ^{nq} is 1; otherwise, the value is undefined. The value of $\mathcal{R}_{j,i}(\varphi_1^{nq}\,\mathcal{U}\,\varphi_2^{nq})$ is the expected cumulative ith reward in the jth execution, accumulated until the first time a (global self-composition) state is reached that satisfies φ_2^{nq}, in case the probability of satisfying $\varphi_1^{nq}\,\mathcal{U}\,\varphi_2^{nq}$ is 1; otherwise, the value is undefined. The semantics of $\mathcal{R}_{j,i}(\varphi_1^{nq}\,\mathcal{U}^{[k_1,k_2]}\,\varphi_2^{nq})$ is similar, but the rewards are accumulated until the first satisfaction of φ_2^{nq} within time $[k_1,k_2]$. Formally, the semantics for $[\![\mathcal{R}_{j,i}(\varphi^{path})]\!]_{\mathcal{M},\boldsymbol{\sigma},\boldsymbol{s}}$ is as follows, given that $[\![\mathbb{P}(\varphi^{path})]\!]_{\mathcal{M},\boldsymbol{\sigma},\boldsymbol{s}} = 1$. If that is not the case, $[\![\mathcal{R}_{j,i}(\varphi^{path})]\!]_{\mathcal{M},\boldsymbol{\sigma},\boldsymbol{s}}$ is undefined.

Since adding rewards to HyperPCTL causes arithmetic values to be potentially undefined, we need to extend the above semantics to handle the propagation of undefined values. For

[1] Instead of syntactical substitutions, we could also use binding functions to map scheduler variables to schedulers and state variables to indices in the state sequence in the context.

$$
\begin{array}{lcl}
\mathrm{Paths}^{\boldsymbol{s}}_{fin}(\mathcal{M}^{\boldsymbol{\sigma}})(\varphi_1^{nq}\mathcal{U}\varphi_2^{nq}) & = & \{\boldsymbol{s}_0 \ldots \boldsymbol{s}_n \in \mathit{Paths}^{\boldsymbol{s}}_{fin}(\mathcal{M}^{\boldsymbol{\sigma}}) \mid \mathcal{M}, \boldsymbol{\sigma}, \boldsymbol{s}_n \models \varphi_2^{nq} \textit{ and} \\
 & & \mathrm{M},\boldsymbol{\sigma},\boldsymbol{s}_i \models \varphi_1^{nq} \wedge \neg\varphi_2^{nq} \textit{ for } i = 0, \ldots, n{-}1\} \\
\mathrm{Paths}^{\boldsymbol{s}}_{fin}(\mathcal{M}^{\boldsymbol{\sigma}})(\varphi_1^{nq}\mathcal{U}^{[k_1,k_2]}\varphi_2^{nq}) & = & \{\boldsymbol{s}_0 \ldots \boldsymbol{s}_n \in \mathit{Paths}^{\boldsymbol{s}}_{fin}(\mathcal{M}^{\boldsymbol{\sigma}}) \mid k_1 \leq n \leq k_2 \textit{ and} \\
 & & \mathrm{M},\boldsymbol{\sigma},\boldsymbol{s}_n \models \varphi_2^{nq} \textit{ and} \\
 & & \mathrm{M},\boldsymbol{\sigma},\boldsymbol{s}_i \models \varphi_1^{nq} \textit{ for } i = 0, \ldots, k_1{-}1 \textit{ and} \\
 & & \mathrm{M},\boldsymbol{\sigma},\boldsymbol{s}_i \models \varphi_1^{nq} \wedge \neg\varphi_2^{nq} \textit{ for } i = k_1, \ldots, n{-}1\} \\
[\![\mathcal{R}_{j,i}(\bigcirc\varphi^{nq})]\!]_{\mathcal{M},\boldsymbol{\sigma},\boldsymbol{s}} & = & \mathit{rew}^{\boldsymbol{\sigma}}_{j,i}(\boldsymbol{s}) + \sum_{\boldsymbol{s}' \in S^{\boldsymbol{\sigma}}} P^{\boldsymbol{\sigma}}(\boldsymbol{s},\boldsymbol{s}') \cdot \mathit{rew}^{\boldsymbol{\sigma}}_{j,i}(\boldsymbol{s}') \\
[\![\mathcal{R}_{j,i}(\varphi_1^{nq}\mathcal{U}\varphi_2^{nq})]\!]_{\mathcal{M},\boldsymbol{\sigma},\boldsymbol{s}} & = & \sum_{\pi \in \mathit{Paths}^{\boldsymbol{s}}_{fin}(\mathcal{M}^{\boldsymbol{\sigma}})(\varphi_1^{nq}\mathcal{U}\varphi_2^{nq})} (\mathit{Pr}^{\boldsymbol{\sigma}}(\pi) \cdot \mathit{rew}^{\boldsymbol{\sigma}}_{j,i}(\pi)) \\
[\![\mathcal{R}_{j,i}(\varphi_1^{nq}\mathcal{U}^{[k_1,k_2]}\varphi_2^{nq})]\!]_{\mathcal{M},\boldsymbol{\sigma},\boldsymbol{s}} & = & \sum_{\pi \in \mathit{Paths}^{\boldsymbol{s}}_{fin}(\mathcal{M}^{\boldsymbol{\sigma}})(\varphi_1^{nq}\mathcal{U}^{[k_1,k_2]}\varphi_2^{nq})} (\mathit{Pr}^{\boldsymbol{\sigma}}(\pi) \cdot \mathit{rew}^{\boldsymbol{\sigma}}_{j,i}(\pi))
\end{array}
$$

Figure 6.4: Semantics for reward operators in HyperPCTL.

each syntactic case, the above semantics remain unchanged if all involved statements used in the definition are defined. It would be an easy job to set the values in all other cases to undefined. However, even if some of the arguments are undefined, we still might be able to conclude a defined value. For example, if one of the operands in a conjunction is false then the conjunction is inevitably false, even if the other operand is undefined. In extension to the above semantics for the cases when all terms used in the definition are defined, below we fix the semantics for the remaining cases to reduce the occurrence of undefined values.

We extend the Boolean domain of true (1) and false (0) with undefined ($\perp$). We use the $\models$ relation as before when all sub-expressions (and thus the formula) are known to be defined, and use $[\![\cdot]\!]$. otherwise. Logical constants as well as atomic propositions are always defined. The value of a conjunction is undefined if and only if none of the operands is false and not both operands are true, whereas a negation is undefined if and only if the negated formula is undefined.

The value of a universally state-quantified formula $\forall\hat{s}(\sigma).\varphi$ is undefined if the value of φ is undefined for at least one instantiation of the formula with a state and is not false for any other instantiation. Likewise, the value of an existentially state-quantified formula $\exists\hat{s}(\sigma).\varphi$ is undefined if the value of φ is undefined for at least one instantiation of the formula with a state and is not true for any other instantiation. The undefinedness of scheduler quantifiers is analogous.

Row	$[\![\varphi_1]\!]$	$[\![\varphi_2]\!]$	p	$[\![\varphi]\!]$
1	$*$	1	$*$	1
2	0	0	$*$	0
3	$\bot$	0	0	0
4	$\bot$	0	$\neq 0$	$\bot$
5	1	0	$*$	p
6	0	$\bot$	$*$	$\bot$
7	$\bot$	$\bot$	$*$	$\bot$
8	1	$\bot$	1	1
9	1	$\bot$	$\neq 1$	$\bot$

Table 6.1: Semantics of $\varphi = \mathbb{P}(\varphi_1 \mathcal{U} \varphi_2)$, partly depending on $p = \sum_{\boldsymbol{s}' \in \mathcal{S}^{\boldsymbol{\sigma}}} P(s, s') \cdot [\![\varphi]\!]_{\mathcal{M},\boldsymbol{\sigma},\boldsymbol{s}'} \in [0,1] \cup \{\bot\}$. Here, $[\![.]\!]$ is short for $[\![.]\!]_{\mathcal{M},\boldsymbol{\sigma},\boldsymbol{s}}$.

Also, the domain of arithmetic values gets extended with the undefined value $\bot$. Arithmetic function applications $f(\varphi_1, \ldots, \varphi_k)$ and arithmetic constraints $\varphi_1 \sim \varphi_2$ are undefined if and only if any of their parameters are undefined. However, for probabilistic until $\varphi = \mathbb{P}(\varphi_1 \mathcal{U} \varphi_2)$ we can exploit available information to increase the number of defined cases, even if the satisfaction of one of the operands is undefined in the current state, as shown in Table 6.1. The information we exploit for the semantics in a state s are probabilistic until values in the successor state, or more precisely, the value of $p = \sum_{\boldsymbol{s}' \in \mathcal{S}^{\boldsymbol{\sigma}}} P(s, s') \cdot [\![\varphi]\!]_{\mathcal{M},\boldsymbol{\sigma},\boldsymbol{s}'} \in [0,1] \cup \{\bot\}$, which we consider undefined iff one of the successor probabilities is undefined.

Table 6.1 extends the original probabilistic until semantics from above with the undefined cases, using $*$ to denote an arbitrary (defined or undefined) arithmetic value. This table is split into three parts. The first part states that if φ_2 is true then the formula value is 1. The second part covers the case where φ_2 is false, where the violation of φ_1 leads to the violation of the formula, and if φ_1 is true then the formula probability equals the value of p.

An interesting case in the second block is when φ_1 is undefined: though in most cases the formula is also undefined, if we know that the probability to satisfy the until formula in the future is 0 then we can safely state that the probability to satisfy the same in the

current state is also 0. Similarly in the third block, if φ_1 is true in the current state and the probability to satisfy the until formula in the future is 1 then, irrelevant of the value of φ_2, the probability to satisfy the until formula from the current state is always 1.

Reward expressions are undefined if the respective path property is not satisfied with probability 1. For the reward expression $\mathcal{R}_{j,i}(\bigcirc\varphi)$, this is the only case in which it is undefined. To evaluate $\varphi = \mathcal{R}_{j,i}(\varphi_1 \mathcal{U} \varphi_2)$, if φ_2 is true in the current state then we need to know only the current state's reward; in this case, the reward is defined independently of the successor states. If φ_2 is false currently then the reward is computed from the current state reward plus the expected successor φ-values, thus undefinedness of the reward expression in a successor state causes undefinedness in the current state. However, if φ_2 is undefined in the current state then we do not know which of these two cases apply; the only case where this does not matter is if the reward expression evaluates in all successor states to 0, namely then the value of φ is the current state reward. Thus if φ_2 is undefined in the current state then the reward expression is undefined in all but this special case, even if the probability of the until formula is 1. The definedness of bounded until formulas are similar to the unbounded case for both probability and reward expressions, except that we now also need to account for the bounds.

However, with these definitions, we only exploit some but not all information, to determine the definedness of a property. Assume, for example, the property that from a state s, the probability to eventually satisfy φ is less than p. It might be the case that in some states reachable from s the value of φ is undefined, triggering the above probability to be undefined by our algorithm. However, φ might be reachable along another path with a probability larger than p, in which case we could have safely stated that it is at least p. Hence, it can be a direction of future research to find a tighter bound on the definedness of a property.

```
1  void mexp(){
2          c = 0; d = 1; i = k;
3          while (i >= 0){
4                  i = i-1; c = c*2;
5                  d = (d*d) % n;
6                  if (b(i) = 1){
7                          c = c+1;
8                          d = (d*a) % n;
9                  }
10         }
```

Figure 6.5: Modular exponentiation in RSA.

6.3 Applications of HyperPCTL with Rewards

The introduction of rewards in the logic allows us to quantify certain metrics like time or cost along the path where we are sure to reach our goal state. Additionally, it allows us to filter our solutions based on these metrics. Below we discuss a few extensions to our existing applications that are possible due to the reward-extension of HyperPCTL.

6.3.1 Timing Attacks

Side-channel timing leaks can potentially reveal sensitive information. For example, RSA uses the modular exponentiation algorithm on the right to compute $a^b \ mod \ n$, where a is the message and b is the encryption key. This implementation is flawed because of the *if* in line 6. Due to the lack of an *else* branch, its execution will take longer if b contains more 1-bit. An attacker could therefore run a thread in parallel to measure the execution time of the algorithm to derive the number of 1-bits in the encryption key.

To prevent such vulnerabilities, we would like the execution time to be independent of the bit values in the encryption key, which is captured by assigning a reward of 1 to each state in the MDPR. Here, each state represents the current position in the code and loop iteration. This results in the following HyperPCTL formula:

$$\forall\hat{\sigma}_1.\forall\hat{\sigma}_2.\forall\hat{s}(\hat{\sigma}_1).\forall\hat{s}'(\hat{\sigma}_2).\,(init_{\hat{s}} \wedge init_{\hat{s}'}) \rightarrow (\mathcal{R}_{\hat{s}}(\diamondsuit\, end_{\hat{s}}) = \mathcal{R}_{\hat{s}'}(\diamondsuit\, end_{\hat{s}'})) \tag{6.4}$$

6.3.2 Probabilistic Conformance

The aim here is to ensure that an implementation conforms with the system it is simulating. We consider the implementation of a 6-sided die with repeated tossing of a fair coin using the Knuth-Yao algorithm [KY76]. For conformance, the probabilistic distribution of reaching the 6 sides of a die should be equal in both cases. We model this problem with an MDP consisting of two components: the first component describes the die and its states represent the faces of the die after being rolled. The second component describes the multiple coin tosses and its states represent the unique combined results of the tosses. Extending this model with rewards allows us to synthesize *efficient* implementations: if we assign to every state, except the absorbing states, a state reward of 1, the expected reward on reaching one of the absorbing states in the coin implementation will be equal to the expected number of coin tosses in it. If we limit the rewards collected in such a path, we can filter the implementations with minimum intermediate states. The following formula specifies that the expected number of coin tosses in such an implementation must be less than 4:

$$\exists\hat{\sigma}.\forall\hat{s}(\hat{\sigma}).\exists\hat{s}'(\hat{\sigma}).dieInit_{\hat{s}} \rightarrow \left(\phi \wedge \mathcal{R}_{\hat{s}'}(\diamondsuit(\bigvee_{l=1}^{6}(die = l)_{\hat{s}'})) < 4\right)$$
$$\text{with } \phi = coinInit_{\hat{s}'} \wedge \bigwedge_{l=1}^{6}(\mathbb{P}(\diamondsuit(die = l)_{\hat{s}}) = \mathbb{P}(\diamondsuit(die = l)_{\hat{s}'})) \tag{6.5}$$

6.3.3 Cost Analysis in Multi-Agent Path Planning

We consider the examples in Fig. 6.6 where two robots R_1, R_2 aim to reach the target cell *end* starting their journey from two different initial cells $(start_1, start_2)$. The robots' behavior is modeled as an MDPR where each cell occupied represents a state. Nondeterministic actions represent all possible moves of the robot from each cell, while the successful maneuvering after having executed an action is captured by a probability distribution.

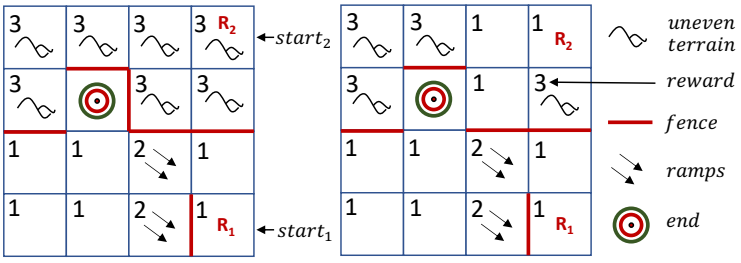


Figure 6.6: The maze on the left satisfies φ_{target}, while on the right it violates φ_{target}.

Fences prevent a robot from moving in a certain direction disabling possible actions in a particular cell, while the presence of ramps or uneven terrain can increase/decrease the probability of correct robot maneuvers. The occupancy of each state has a cost in terms of energy consumption modeled as a positive reward. We want to check that for all possible (memoryless) schedulers, when robots R_1, R_2 start their mission from their respective initial conditions and they can both reach the target state with probability 1, then the expected energy consumption for robot R_1 is less than the expected energy consumption for robot R_2. This can be expressed as the following probabilistic hyperproperty:

$$\varphi_{target} = \forall\hat{\sigma}.\forall\hat{s}(\hat{\sigma}).\forall\hat{s}'(\hat{\sigma}).\psi \rightarrow \Big(\mathcal{R}_{\hat{s}}(\Diamond\, end_{\hat{s}}) < \mathcal{R}_{\hat{s}'}(\Diamond\, end_{\hat{s}'})\Big)$$
$$\text{where } \psi = \Big(start_{1\,\hat{s}} \wedge start_{2\,\hat{s}'} \wedge \mathbb{P}(\Diamond\, end_{\hat{s}}) = 1 \wedge \mathbb{P}(\Diamond\, end_{\hat{s}'}) = 1\Big) \tag{6.6}$$

6.3.4 Probabilistic Self-stabilizing Systems

In distributed systems, randomization is often used to break symmetry between processes to tackle impossibility results. For instance, self-stabilizing token circulation in a ring is impossible in a non-probabilistic setting but Herman's algorithm [Her90] (see Fig. 6.7) uses randomization to ensure recovery to a *stable state* (i.e., there is only one token circulating) with probability one. In such an algorithm, from certain initial states, convergence to a stable state may be faster than others and if faults hit those states with a higher probability, it reduces the average convergence time significantly. Thus, designers of self-stabilizing algorithms often use state encodings to tackle slow recovery [FBT13]. The following formula

intends to check whether there exists a state from which the convergence time is twice slower than from some other state:

$$\forall \hat{\sigma}.\exists \hat{s}(\hat{\sigma}).\exists \hat{s}'(\hat{\sigma}).\Big(\mathcal{R}_{\hat{s}}(\Diamond\, stable_{\hat{s}}) > 2 \cdot \mathcal{R}_{\hat{s}'}(\Diamond\, stable_{\hat{s}'})\Big) \tag{6.7}$$

Note: Herman's algorithm yields a DTMCR and, thus, the choice of scheduler quantification is irrelevant.

1: **Variable:** x_i : *boolean* $\in \{0,1\}$
2: **Guarded Commands:**
3: $x_i = x_{i-1} \longrightarrow p : x_i := 0 \ + \ (1-p) : x_i := 1;$
4: $x_i \neq x_{i-1} \longrightarrow 1 : x_i := x_{i-1};$

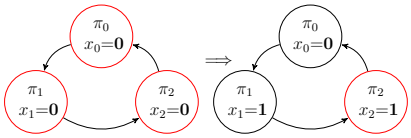


Figure 6.7: Herman's algorithm [Her90] for process i and example for three processes.

6.4 Model Checking Algorithm for Reward Operators

HyperPCTL provides an increased level of expressiveness over PCTL and PCTL*, causing the model checking problem for MDPRs to be undecidable even without rewards, as shown in [ÁBBD20b]. To achieve decidability for HyperPCTL without rewards, in [ÁBBD20b] we restricted the domain of scheduler quantification to *memoryless non-probabilistic schedulers*. For this restricted domain, the model checking problem is NP-complete (or coNP-complete) when the scheduler quantification is existential (or universal). We provided a model checking algorithm by logically encoding HyperPCTL satisfaction problems as linear real-arithmetic formulas and used an SMT solver to check the encodings for satisfiability. Elaborate explanations of encoding non-reward operators can be found in [ÁBBD20b].

After adding rewards, the model checking problem restricted to finite memoryless schedulers is still decidable. Similar to the standard model checking problem for Markov Reward Models, computing the expected reward earned until a certain set of states is reached, has a polynomial time complexity in the size of the MDP: the problem can be solved by determining a linear real-arithmetic equation system via graph reachability analysis and solving it.

This means adding rewards does not change the class of complexity of the model checking problem as identified in [ÁBBD20b].

However, adding rewards to the problem requires a major adaption of the logical encoding. The reason is that expected reward values might be undefined, and undefinedness might propagate from the inner sub-formulas to the formula value. The main contributions of this section are (1) to extend the model checking algorithm from [ÁBBD20b] to encode the semantics of reward-related HyperPCTL expressions and (2) to modify the previous encodings to model undefinedness propagation for the remaining language components. To ease understanding, in the following, we consider unary-reward models and a single existential scheduler quantifier in our properties; extension to multi-dimensional rewards and several scheduler quantifiers without quantifier alternation is doable by little modifications to the algorithms. Given their finite domain, support for scheduler quantifier alternation is possible, too, but it would require more involved extensions. Assume as input an MDPR model $\mathcal{M}$ and a HyperPCTL formula ϕ. In [ÁBBD20b] we used Boolean variables $holds_{s,\phi}$ to encode the truth value of a Boolean-valued formula ϕ in state s. In this work, we replace the two-valued domain for these variables with a three-valued domain over the values *true* (1), *false* (0) and *undefined* ($\perp$). Furthermore, we use variables $val_{s,\phi}$ to store the numerical value of an arithmetic expression ϕ in state s. To also encode the definedness of arithmetic values, we introduce additional Boolean variables $def_{s,\phi}$ which should be true if and only if the corresponding value is defined. Finally, to encode a scheduler, we use for each state of $\mathcal{M}$ a variable σ_s to store the chosen action.

The starting point of the encoding is Algorithm 12, which begins by encoding the scheduler choice [2] in line 2. The semantics of the non-quantified inner formula ϕ^{nq} under a given scheduler choice in each of the states are encoded in line 3. This basic encoding E is extended in two directions: formula T encodes that ϕ can be made true by some suitable quantifier instantiation, whereas U encodes that ϕ can be made true or undefined. Only if none of

[2]For n scheduler quantifiers, we would simply need to include such a scheduler encoding for each of the schedulers $\sigma_1, \ldots, \sigma_n$, and in the rest of the encoding, refer to the respective schedulers σ_i instead of σ.

Algorithm 12: Main SMT encoding algorithm

Input: $\mathcal{M} = (S, Act, P, \mathsf{AP}, L, rew)$: MDPR;
φ: HyperPCTL formula.
Output: Whether $\mathcal{M}$ satisfies φ.

1 **Function** `Main`$(\mathcal{M}, \varphi = \exists\hat{\sigma}.Q_1\hat{s}_1(\hat{\sigma}).\ldots.Q_n\hat{s}_n(\hat{\sigma}).\varphi^{nq})$**:**
2 $\quad E := \bigwedge_{s\in S}(\bigvee_{\alpha\in Act(s)} \sigma_s = \alpha)$
3 $\quad E := E \wedge$ `Semantics`$(\mathcal{M}, \varphi^{nq}, n)$
4 $\quad T := E \wedge$ `Eval`$(\mathcal{M},\ \varphi,\ \{1\})$
5 $\quad U := E \wedge$ `Eval`$(\mathcal{M},\ \varphi,\ \{\bot, 1\})$
6 $\quad$ **if** $check(T) = SAT$ **then return** $TRUE$
7 $\quad$ **else if** $check(U) = SAT$ **then return** $UNDEF$
8 $\quad$ **else return** $FALSE$

these two cases apply (i.e. if both formulas are unsatisfiable), we conclude that $\mathcal{M}$ does not satisfy ϕ. Not listed in the algorithm is the case of a universal scheduler quantifier, where we use negation to get an existential formula, apply the listed algorithm, and negate the answer.

Algorithm 13: Encoding certain formula values

Input: $\mathcal{M} = (S, Act, P, \mathsf{AP}, L, rew)$: MDPR;
ϕ: HyperPCTL formula; $v \subseteq \{0, 1, \bot\}$.
Output: Encoding that $\mathcal{M}, (), () \models \exists\hat{\sigma}.Q_1\hat{s}_1.\ldots.Q_n\hat{s}_n.(\phi^{nq} \in v)$.

1 **Function** `Eval`$(\mathcal{M},\ \phi = \exists\hat{\sigma}.Q_1\hat{s}_1.\ldots.Q_n\hat{s}_n.\phi^{nq},\ v)$**:**
2 $\quad$ **foreach** $i = 1, \ldots, n$ **do**
3 $\quad\quad$ **if** $Q_i = \forall$ **then** $B_i :=$ " $\bigwedge_{s_i\in S}$ " **else** $B_i :=$ " $\bigvee_{s_i\in S}$ "
4 $\quad$ **return** $B_1 \ldots B_n$ $(holds_{(s_1,\ldots,s_n),\phi^{nq}} \in v)$

The semantics of formulas is encoded by Algorithm 14. We omit the pseudocode of sub-algorithms that were needed also without rewards; these are similar to those in [ÁBBD20b] but get extended with the encoding of definedness as explained in Section 6.2.2. Relevant for rewards is line 14, calling the method `RewSemantics` in Algorithm 15 to encode the semantics of the reward operators. In the case of rewards over the next operator $\phi = \mathcal{R}_{\hat{s}_i}(\bigcirc\phi')$, we first encode the probability $\mathbb{P}(\bigcirc\phi')$; ϕ is undefined if this probability is not 1 (line 5). If the probability is defined, then the reward is the expected reward of the successors in the ith execution (line 7).

Algorithm 14: SMT encoding for the meaning of an input formula

Input: $\mathcal{M} = (S, Act, P, \mathsf{AP}, L, rew)$: MDPR; φ: quantifier-free HyperPCTL formula or expression; n: number of state variables in φ.

Output: SMT encoding of the meaning of φ in n-ary self-composition of $\mathcal{M}$.

1 **Function** `Semantics`$(\mathcal{M}, \varphi, n)$**:**
2 **if** φ *is true* **then** $E := \bigwedge_{s \in S^n} holds_{s,\varphi}{=}1$
3 **else if** φ *is* $a_{\hat{s}_i}$ **then**
4 $E := (\bigwedge_{s \in S^n,\, a \in L(s_i)} (holds_{s,\varphi}{=}1)) \wedge (\bigwedge_{s \in S^n,\, a \notin (s_i)} (holds_{s,\varphi}{=}0))$
5 **else if** φ *is* $\neg\varphi'$ **then**
6 $E :=$ `Semantics`$(\mathcal{M}, \varphi', n) \wedge \bigwedge_{s \in S^n} (holds_{s,\varphi'}{=}0 \rightarrow holds_{s,\varphi}{=}1) \wedge$
7 $\bigwedge_{s \in S^n} (holds_{s,\varphi'}{=}1 \rightarrow holds_{s,\varphi}{=}0) \wedge \bigwedge_{s \in S^n} (holds_{s,\varphi'}{=}\bot \rightarrow holds_{s,\varphi}{=}\bot)$
8 **else if** φ *is* $\varphi_1 \wedge \varphi_2$ **then** $E :=$ `SemanticsConjunction`$(\mathcal{M}, \varphi, n)$
9 **else if** φ *is* $\varphi_1^{ar} \sim \varphi_2^{ar}$ **then** $E :=$ `SemanticsComp`$(\mathcal{M}, \varphi, n)$
10 **else if** φ *is* $f(\varphi_1^{ar}, \ldots, \varphi_k^{ar})$ **then** $E :=$ `SemanticsArithmetic`$(\mathcal{M}, \varphi, n)$
11 **else if** φ *is* $\mathbb{P}(\bigcirc \varphi')$ **then** $E :=$ `SemanticsNext`$(\mathcal{M}, \varphi, n)$
12 **else if** φ *is* $\mathbb{P}(\varphi_1 \mathcal{U} \varphi_2)$ **then** $E :=$ `SemanticsUnboundedUntil`$(\mathcal{M}, \varphi, n)$
13 **else if** φ *is* $\mathbb{P}(\varphi_1 \mathcal{U}^{[k_1,k_2]} \varphi_2)$ **then** $E :=$ `SemanticsBoundedUntil`$(\mathcal{M}, \varphi, n)$
14 **else** $E :=$ `RewSemantics`$(\mathcal{M}, \varphi, n)$
15 **return** E

To encode the reward of unbounded until formulas, we first need to encode the probability of the until formula, since this probability needs to be 1 for a defined reward value. Then we call the `RewardUnboundedUntil` method from Algorithm 16, which implements the semantics of the reward of unbounded until from Section 6.2.2. Undefinedness is covered in line 5 when the probability of the unbounded until is either not defined or not 1, and in the lines 9-10 when the probability of the unbounded until is 1, φ_2 is not true and either a successor reward is undefined, or ϕ_2 is undefined and the successor rewards are not zero. The method `RewardBoundedUntil` for reward expressions with bounded until not shown here, is similar to the unbounded case but needs additional bookkeeping about the time interval within which φ_2 needs to be satisfied.

6.5 Evaluation

We have implemented a prototype of the presented algorithm by extending our tool HyperProb [DABB21] to support rewards. The implementation has been coded in Python

Algorithm 15: SMT encoding for the meaning of reward operators

Input: $\mathcal{M} = (S, Act, P, \mathsf{AP}, L, rew)$: MDPR; φ: quantifier-free HyperPCTL formula or expression; n: number of state variables in φ.

Output: SMT encoding of the meaning of φ in n-ary self-composition of $\mathcal{M}$.

Function `RewSemantics`$(\mathcal{M}, \varphi, n)$**:**

- **if** φ *is* $\mathcal{R}_{\hat{s}_i}(\bigcirc \varphi')$ **then**
  - $E := \texttt{Semantics}(\mathcal{M}, \mathbb{P}(\bigcirc \varphi'), n)$
  - **foreach** $s = (s_1, \ldots, s_n) \in S^n$ **do**
    - $E := E \wedge ((val_{s,\mathbb{P}(\bigcirc \varphi')} \neq 1 \vee \neg def_{s,\mathbb{P}(\bigcirc \varphi')}) \leftrightarrow \neg def_{s,\varphi})$
    - **foreach** $\alpha = (\alpha_1, \ldots, \alpha_n) \in Act(s_1) \times \ldots \times Act(s_n)$ **do**
      - $E := E \wedge ([def_{s,\varphi} \wedge \bigwedge_{j=1}^{n} \sigma_{s_j} = \alpha_j] \rightarrow [val_{s,\varphi} = rew(s_i) + \sum_{s' \in supp(\alpha_1) \times \ldots \times supp(\alpha_n)} ((\prod_{j=1}^{n} P(s_j, \alpha_j, s'_j)) \cdot rew(s'_i))])$
- **else if** φ *is* $\mathcal{R}_{\hat{s}_i}(\varphi_1 \mathcal{U}^{[k_1,k_2]} \varphi_2)$ **then**
  - $E := \texttt{SemanticsBoundedUntil}(\mathcal{M}, \mathbb{P}(\varphi_1 \mathcal{U}^{[k_1,k_2]} \varphi_2), n)$
  - $E := E \wedge \texttt{RewardBoundedUntil}(\mathcal{M}, \varphi, n)$
- **else if** φ *is* $\mathcal{R}_{\hat{s}_i}(\varphi_1 \mathcal{U} \varphi_2)$ **then**
  - $E := \texttt{SemanticsUnboundedUntil}(\mathcal{M}, \mathbb{P}(\varphi_1 \mathcal{U} \varphi_2), n)$
  - $E := E \wedge \texttt{RewardUnboundedUntil}(\mathcal{M}, \varphi, n)$
- **return** E

Algorithm 16: SMT encoding for the reward of unbounded until

Input: $\mathcal{M} = (S, Act, P, \mathsf{AP}, L, rew)$: MDPR; φ: HyperPCTL unbounded until formula of the form $\mathcal{R}_{\hat{s}_i}(\varphi_1 \mathcal{U} \varphi_2)$; n: number of state variables in φ.

Output: SMT encoding of φ's meaning in the n-ary self-composition of $\mathcal{M}$.

Function `RewardUnboundedUntil`$(\mathcal{M}, \varphi = \mathcal{R}_{\hat{s}_i}(\varphi_1 \mathcal{U} \varphi_2), n)$**:**

- $\varphi' := \mathbb{P}(\varphi_1 \mathcal{U} \varphi_2)$; $\quad E := \texttt{true}$
- **foreach** $s = (s_1, \ldots, s_n) \in S^n$ **do**
  - $E := E \wedge (holds_{s,\varphi_2} = 1 \rightarrow (val_{s,\varphi} = rew(s_i) \wedge def_{s,\varphi}))$
  - $E := E \wedge ((val_{s,\varphi'} \neq 1 \vee \neg def_{s,\varphi'}) \rightarrow \neg def_{s,\varphi})$
  - **foreach** $\alpha = (\alpha_1, \ldots, \alpha_n) \in Act(s_1) \times \ldots \times Act(s_n)$ **do**
    - $E := E \wedge ((val_{s,\varphi'} = 1 \wedge def_{s,\varphi'} \wedge holds_{s,\varphi_2} \neq 1 \wedge \bigwedge_{j=1}^{n} \sigma_{s_j} = \alpha_j) \rightarrow$
    - $[val_{s,\varphi} = rew(s_i) + \sum_{s' \in supp(\alpha_1) \times \ldots \times supp(\alpha_n)} ((\prod_{i=1}^{n} P(s_j, \alpha_j, s'_j)) \cdot val_{s',\varphi}) \wedge$
    - $(\neg def_{s,\varphi} \leftrightarrow [(\bigvee_{s' \in supp(\alpha_1) \times \ldots \times supp(\alpha_n)} \neg def_{s',\varphi}) \vee$
    - $(holds_{s,\varphi_2} = \bot \wedge val_{s,\varphi} \neq rew(s_i))])])$
- **return** E

using the libraries Lark [lar] for parsing the input formula, and Stormpy [stob] for parsing the input MDPR. The generated constraints are then solved by the SMT solver Z3 [dMB08]. Our implementation cannot handle all possible cases of undefinedness. We currently do not calculate the extent of partial definedness of a property in a model. We check whether

Case study		VR	Running time (s)			#SMT variables	#sub formulas	#states	#transitions
			Encoding	**Solving**	**Total**				
TA	1-bit key	×	0.11	0.01	0.12	344	1008	8	10
	16-bit key	×	16.41	3.69	20.10	19244	49728	68	100
	30-bit key	×	143.49	44.64	188.13	62868	160160	124	184
	45-bit key	×	774.53	1304.98	2079.51	137448	348080	184	274
PC	s=(0)	✓	5.03	2.03	7.06	7281	34681	20	186
	s=(0,1,2)	✓	6.66	8.91	15.57	7281	61631	20	494
	s=(0,. . .,4)	✓	8.82	35	43.82	7281	88581	20	802
	s=(0,. . .,6)	✓	11.64	53.05	64.69	7281	115531	20	1110
RO	3x3	✓	0.87	0.05	0.92	2179	7622	18	66
	3x3	×	0.93	0.05	0.98	2179	7622	18	66
	4x4	✓	3.55	0.28	3.83	6561	21572	32	160
	4x4	×	3.43	0.25	3.68	6561	21476	32	148
	5x5	✓	13.07	0.5	13.57	15651	48302	50	250
	5x5	×	13.19	0.98	14.17	15651	48302	50	250
	6x6	✓	44.52	1.04	45.56	32041	96096	72	398
	6x6	×	44.65	7.48	52.13	32041	96096	72	398
HS	$n = 3$	✓	0.1	0.01	0.11	489	4655	8	28
	$n = 5$	✓	0.95	0.13	1.08	2369	7047	32	244
IJ	$n = 3$	✓	0.08	0.01	0.09	169	698	7	21
	$n = 4$	✓	0.24	0.04	0.28	601	2194	15	56
	$n = 5$	✓	0.89	0.33	1.22	2233	7010	31	140
	$n = 6$	✓	3.93	19.39	23.32	8569	23362	63	336

Table 6.2: Experimental results. **VR:** Verification result. **TA**: Timing attack. **PC**: Probabilistic conformance. **RO**: Robotics example. **HS**: Herman's algorithm. **IJ**: Israeli-Jaflon's algorithm. ✓: the result is true. ×: the result is false.

the states queried in the property are reachable with a probability of one and proceed in the calculation of rewards in such cases. Hence, we have evaluated case studies, where the reachability probabilities are always one.

The concept of rewards has eased the modeling of case studies with respect to counting of expected steps needed to reach a state. Hence, for timing attack and probabilistic conformance case studies, the number of transitions and states are less when compared to the models used in [ÁBBD20b]. The implementation also returns a witness/counterexample whenever possible, allowing us to synthesize schedulers. Note that, though the ensemble of schedulers in the executions (i.e. $\boldsymbol{\sigma}$ in the semantical context) define a scheduler in the self-composition, not all schedulers of the self-composition can be defined this way, posing a major difference between scheduler synthesis for PCTL and for HyperPCTL.

For the **TA** case study, we have modeled the problem with {1, 16, 30, 45}-bit encryption keys. We have verified the HyperPCTL formula described in Section 6.3.1. The property does not hold on the given model and our implementation finds this bug. Since our implementation can handle only one scheduler quantifier, we have added a second copy of the model to the input MDPR such that the single scheduler can assign different actions to the states in the two copies of the model.

For the **PC** case study, we have verified the property described in Section 6.3.2. We have started with a model with all possible transitions, represented non-deterministically, from the initial state s_0. For all other states, we allowed only the transitions that would give us a correct solution. We challenged our implementation to synthesize a scheduler that will satisfy the required probabilities within the given reward bound. We scaled the model by incrementally allowing all possible combinations of transitions using nondeterministic actions in each state and limited the expected coin tosses to be 4 for each experiment. For all the cases, our implementation was successful in finding a solution, which we verified manually as correct.

For the **RO** case study, we have verified the property described in Section 6.3.3. We have scaled the model in terms of maze size and verified both positive and negative cases of pathfinding. On self-stabilizing systems, we have verified several properties and described one of them in Section 6.3.4. This property is satisfied and we have successfully found a witness. We have reported the timing data for this property in Table 6.2. We have verified the property in models representing both Herman's (**HS**) and Israeli-Jaflon's (**IJ**) [IJ90] algorithms. Since Herman's algorithm is only valid for odd processes, we tried verification over $\{3, 5\}$ processes. For Israeli-Jaflon's, we tried it over $\{3, 4, 5, 6\}$ process.

The experiments have been performed in a Docker container running on a system with a 2.3 GHz i7 processor and 32 GB of RAM. Because of the incomplete implementation of handling of undefined values, which would add a significant number of additional constraints, the reported execution times are lower than they would normally be. From Table 6.2, it is

clear that the execution times for even relatively small MDPRs are large. This is because of the inherent complexity of the problem, to which reward operators add a new dimension of complexity.

6.6 Summary

Probabilistic logics allow us to argue about the reachability of a state or set of states. However, they cannot differentiate between paths used to reach the states as 'good' or 'bad' based on criteria like robustness, less energy or power consumption, avoidance of obstacles, etc. In this work, we focused on extending our previously proposed general language to connect it to reward models. This provided additional power to express the concept of 'best' or 'efficient' paths among all possible paths found. We have also extended our previously proposed SMT-based algorithm and demonstrated our approach in a few interesting case studies. This extension came with the challenge of undefined reward values that are caused by the basic definition of rewards being infinite for unreachable states. As part of future work for this extension, we have to modify our implementation to accommodate for undefined results and consider other forms of rewards such as transition-based results and conversion between state and transition rewards.

Chapter 7

HyperProb: A Model Checker for Probabilistic Hyperproperties

7.1 Introduction

In this chapter, we introduce the tool HyperProb, a model checker for verifying probabilistic hyperproperties expressed in the temporal logic HyperPCTL [ÁB18, ÁBBD20b] on Markov Decision Processes (MDP) given as a PRISM model [KNP11]. HyperProb reduces the model checking problem to a satisfiability modulo theory (SMT) problem, implemented by the SMT-solver Z3 [dMB08]. Along with the Boolean verdict of the model checking problem, HyperProb may provide a witness or a counterexample to the hyperproperty represented as a deterministic memoryless scheduler. A witness is provided when the probabilistic hyperproperty containing an existential quantifier, holds, and it can be used to synthesize the induced discrete-time Markov chain (DTMC) satisfying the desired probabilistic hyperproperty. A counterexample is provided when the probabilistic hyperproperty containing a universal quantifier over all possible schedulers, does not hold, and it can be exploited to synthesize an adversarial attack that may violate the desired probabilistic hyperproperty. Our implementation is available at: https://www.cse.msu.edu/tart/tools. This tool is the culmination of the different extensions to the logic and model checking algorithm we discussed in the previous chapters.

7.2 Input to the Tool

```
1  mdp
2  module basic_mdp
3  h: [0..1];  // high input
4  ℓ: [0..2]; // low output
5  [alpha] (ℓ=0 & h=0) → 3/4: (ℓ'=1) + 1/4: (ℓ'=2);
6  [alpha] (ℓ=0 & h=1) → 2/3: (ℓ'=1) + 1/3: (ℓ'=2);
7  [beta]  (h=0) → 1/2: (ℓ'=1) + 1/2: (ℓ'=2);
8  [beta]  (h=1) → 1/2: (ℓ'=1) + 1/2: (ℓ'=2);
9  [tau]   (!ℓ=0) → 1: true;
10 endmodule
11 init  (ℓ=0)  endinit   //initial states
12 label "h0" = (h=0);   //h0 label on s0
13 label "h1" = (h=1);   //h1 label on s1
14 label "ℓ1"   = (ℓ=1); //ℓ1 label on s2,s4
15 label "ℓ2"   = (ℓ=2); //ℓ2 label on s3,s5
```

Figure 7.1: PRISM model generating the MDP in Fig.7.2

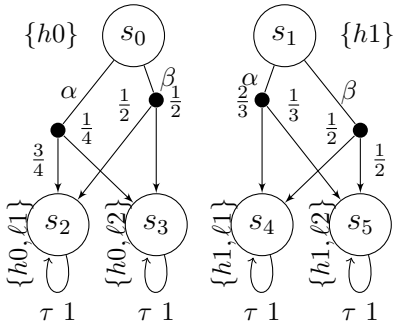


Figure 7.2: MDP of the PRISM program in Fig 7.1.

Input modeling language. The input model is provided as a high-level PRISM[1] program [KNP11]. We illustrate the language here on a simple example. This program takes a high-security input h and computes a low-security output ℓ. The execution steps are represented symbolically by probabilistic actions. For example, line 5 in Fig. 7.1 declares that action `alpha` can be executed if $\ell = 0$ and $h = 0$, and the action resets ℓ to the value 1 with

[1] https://www.prismmodelchecker.org/

$$
\begin{array}{lll}
\varphi^{sched} & ::= & \text{“AS” NAME “.” } \varphi^{state} \text{ | “ES” NAME “.” } \varphi^{state} \\
\varphi^{state} & ::= & \phi \text{ | “A” NAME “.” } \varphi^{state} \text{ | “E” NAME “.” } \varphi^{state} \\
\phi & ::= & \text{“t” | “f” | NAME “(” NAME “)” | “}\sim\text{” } \phi \text{ |} \\
 & & \phi \text{ “\&” } \phi \text{ | } \phi \text{ “|” } \phi \text{ | } \phi \text{ “=>” } \phi \text{ | } \phi \text{ “<->” } \phi \text{ | c} \\
\text{c} & ::= & \text{p “<” p | p “<=” p | p “=” p | p “>=” p | p “>” p} \\
\text{p} & ::= & \text{“P” } \psi \text{ | p “+” p | p “-” p | p “.” p | NUM} \\
\psi & ::= & \text{“(X” } \phi \text{ “)” | “(” } \phi \text{ “U” } \phi \text{ “)” |} \\
 & & \text{“(” } \phi \text{ “U[” NUM “,” NUM “]” } \phi \text{ “)” | “(F” } \phi \text{ “)”}
\end{array}
$$

Figure 7.3: Grammar defining HyperPCTL inputs to HyperProb, where the NUM token is a decimal number and NAME is a non-empty string.

probability 3/4, and to the value 2 with probability 1/4. Line 10 defines that $\ell = 0$ initially. Modelling each state explicitly yields the Markov decision process (MDP) depicted in Fig. 7.2, where the state labelling with atomic propositions is defined in the lines 11-14 in Fig. 7.1. For the given PRISM program, we use STORMpy to parse the model and to generate the underlying MDP.

Specification language: To protect the secret value h, there should be no probabilistic dependency between observations on the low variable ℓ and the value of h. For example, an attacker that chooses a scheduler that always takes action α from states s_0 and s_1 can learn whether or not $h = 0$ by observing the probability of obtaining $\ell = 1$ (or $\ell = 2$). Such properties, which compare observations on system execution when using different schedulers in different initial states are *probabilistic hyperproperties*. These properties, which should be model checked by HyperProb, can be specified in the temporal logic HyperPCTL [ÁBBD20b]. We refer to [ÁBBD20b] for a detailed description of HyperPCTL but show the tool's input grammar in Fig. 7.3. For example, the property that we expect from the MDP in Fig 7.2 is that for any scheduler (*AS sched.*) and any two initial states $s1$ and $s2$ (*A* $s1$. *A* $s2$.) with different high input values ($h0(s1) \wedge h1(s2)$), the probability to reach the low value 1 from

$s1$ (P F $\ell 1(s1)$) is the same as from $s2$ (P F $\ell 1(s2)$), and analogously for the low value 2:

$$AS\ sched.\ A\ s1.\ A\ s2.\ (h0(s1) \wedge h1(s2)) =>$$
$$(P\ F\ \ell 1(s1) = P\ F\ \ell 1(s2))\ \&\ (P\ F\ \ell 2(s1) = P\ F\ \ell 2(s2)) \tag{7.1}$$

Note that in the above formula, we use expressions of the form $a(s)$ to state that the atomic proposition a holds in the computation tree starting from the (quantified) state s. For this formula, our tool will provide a violation as output with a deterministic memoryless scheduler that would leak information to an attacker.

7.3 Tool Structure and Usage

In this section, we elaborate on the implementation details of the tool and its usage along with examples.

7.3.1 Implementation

HyperProb, is an optimized implementation of the algorithm presented in [ÁBBD20b]. Given an MDP and a HyperPCTL property, it verifies if the given hyperproperty holds in the input MDP. Depending on the scheduler quantifier in the hyperproperty, if it holds, we get a witness to the property (in the case of $\exists$) or if the hyperproperty does not hold, we get a counterexample in the model (in the case of $\forall$). The witness or counterexample is defined by the actions chosen at each state to obtain the required DTMC.

The tool has been implemented using Python3 and depends on several python packages. Computer Arithmetic and Logic (CArL) [CARa], is an open source C++ library for handling of complex mathematical computations and is needed by STORM. carl-parser [CARb] is an ANTLR based parser which is meant as an extension to CArL. pyCArL [pyc] essentially provides python bindings for CArL and is a dependency for STORM [stoa]. STORM is an academically developed probabilistic model checker. STORMpy [stob] is the python binding for STORM that we use to parse the input model. For the ease of usage, we have provided

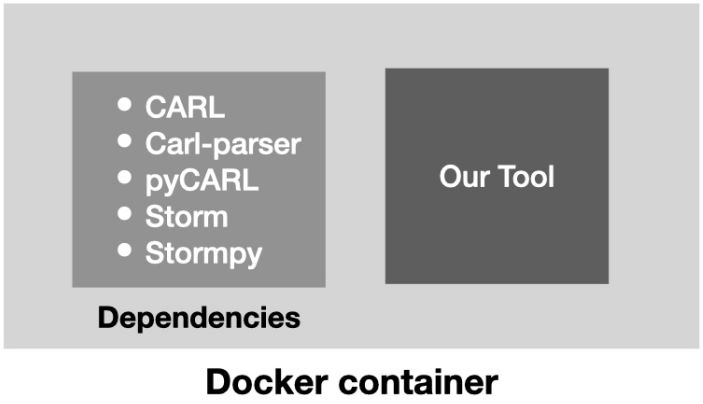


Figure 7.4: Overview of the docker container with the tool and its dependencies.

a docker image for the user. The image comes pre-installed with Ubuntu, all the required dependencies, and the tool. Docker makes it easier for the user to run the tool, as they do not have to install any of the dependencies mentioned above. The main advantage of docker is that running of the tool becomes independent of the operating system the user has. The overall view of the docker container can be seen in Fig. 7.4.

Inside the tool, as shown in Fig. 7.5 , we first parse the model using STORMpy to store them in an optimized way and for easy retrieval of the details of the model like labels, transitions, states, and actions. We, then, parse the input hyperproperty into a syntax tree that allows us to recursively encode the property in the next stages. Using the parsed model, we first encode all possible actions in each state as SMT constraints. Using the parsed model and the parsed property, we encode, as SMT constraints, both, the quantifier combinations by generating constraints that should be satisfied at all states (for $\forall$) or by at least one state (for $\exists$), and the semantic interpretation of the operators in the formula.

In the final step of the algorithm, we feed these constraints to the SMT solver Z3 [dMB08]. If our scheduler quantifier is a $\exists$, and the constraints are satisfiable, the tool outputs a witness to the property, as a set of actions that should be chosen at each state of the model. If our scheduler quantifier is a $\forall$, and the constraints are unsatisfiable, the tool outputs a counterexample to the property as a set of actions that should be chosen at each state of

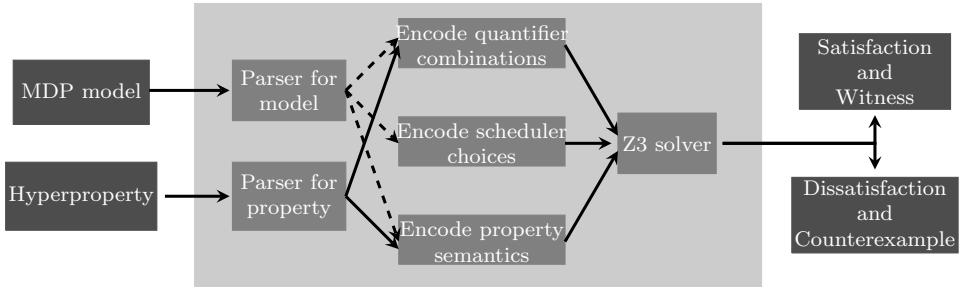


Figure 7.5: Dataflow inside the tool.

the model. For all the other combinations of the scheduler quantifier and satisfiability, the tool returns a true/false but with no sequence of actions.

HyperProb incorporates additional optimizations whose impact is reflected in Table 7.1 (columns **N** referring to original implementation in [ÁBBD20b], and **O** referring to the optimized implementation). In particular, instead of considering all possible state combinations when encoding a state formula for Z3, we consider only the relevant state combinations. For example, let us consider the scenario in Fig. **??** with six states, and its HyperPCTL specification as described in Section 7.2 with two state quantifiers. Our goal is to encode the atomic proposition $h0(s1)$. In the unoptimized version, this is encoded for 36 state combinations $((s_{0_1}, s_{0_2}), (s_{0_1}, s_{1_2}), \ldots, (s_{1_1}, s_{0_2}), \ldots, (s_{5_1}, s_{5_2}))$, since we have two copies of the MDP. However, in HyperProb, we consider the information that while encoding $h0(s1)$, only states of the second copy of the MDP are relevant. Hence, we encode it for just six state combinations, $((s_{0_1}, s_{0_2}), \ldots, (s_{0_1}, s_{5_2}))$. We keep the first state for the irrelevant copy as the first member of every state combination. This not only reduces the number of constraints generated but also relaxes the constraints Z3 needed work on.

7.3.2 Usage

In order to use the tool, we will first need to ensure that our system has docker [doc] installed in it. Then, download the *docker image* and create a container from it. Inside

the container, all dependency packages are installed in the */opt* folder under the root directory. The main tool package is located in */home/HyperProb* folder under the root directory. We can add our model file anywhere in this folder. The tool can be run by invoking the **source.py** script with appropriate inputs in the format,

python source.py **file_path_for_model hyperproperty**

Here *file_path_for_model* refers to the file path with respect to */home/HyperProb* directory as base and the hyperproperty is written according to the grammar in Section 7.2. For example, if your file is */home/HyperProb/models/mdp.nm*, your command would be,

python source.py **models/mdp.nm hyperproperty**

The commands to replicate all our case studies have been placed under */home/HyperProb/benchmark_files/Experiments.txt*.

7.4 Evaluation

In this section, we remind the readers briefly of our benchmarks and discuss our experimental results, comparing them to our previous implementations.

7.4.1 Case Studies

Side-channel timing leaks open a channel for attackers to infer the value of a secret by observing the execution time of a function. For example, the heart of the RSA public-key encryption algorithm is the modular exponentiation algorithm that computes ($a^b \mod n$), where a is an integer representing the plaintext and b is the integer encryption key. A careless implementation can leak b through a probabilistic scheduling channel (see Fig. 7.6). This program is not secure since the two branches of the *if* have different timing behaviours. Under a fair execution scheduler for parallel threads, an attacker thread can infer the value

```
1  void mexp(){
2    c = 0; d = 1; i = k;
3    while (i >= 0){
4      i = i-1; c = c*2;
5      d = (d*d) % n;
6      if (b(i) = 1)
7        c = c+1;
8        d = (d*a) % n;
9      }
10 }
11 /************/
12 t = new Thread(mexp());
13 j = 0; m = 2 * k;
14 while (j < m & !t.stop) j++;
15 /************/
```

Figure 7.6: Modular exponentiation.

of b by running in parallel to a modular exponentiation thread and iteratively incrementing a counter variable until the other thread terminates (lines 12-14). To model this program by an MDP, we can use two nondeterministic actions for the two branches of the *if* statement, such that the choice of different schedulers corresponds to the choice of different bit configurations `b(i)` for the key `b`. This algorithm should satisfy the following property: the probability of observing a concrete value in the counter `j` should be independent of the bit configuration of the secret key `b`:

$$\forall \hat{\sigma}_1.\forall \hat{\sigma}_2.\forall \hat{s}(\hat{\sigma}_1).\forall \hat{s}'(\hat{\sigma}_2).\Big(init_{\hat{s}} \wedge init_{\hat{s}'}\Big) \Rightarrow \bigwedge_{\ell=0}^{m} \Big(\mathbb{P}(\Diamond(j=\ell)_{\hat{s}}) = \mathbb{P}(\Diamond(j=\ell)_{\hat{s}'})\Big) \qquad (7.2)$$

Another example of a timing attack that can be implemented through a probabilistic scheduling side channel is password verification. It is typically implemented by comparing an input string with another confidential string (see Fig 7.7). Also here, an attacker thread can measure the time necessary to break the loop and use this information to infer the prefix of the input string matching the secret string.

```
int str_cmp(char * r){
  char * s = 'Bg\$4\0';
  i = 0;
  while (s[i] != '\0'){
    i++;
    if (s[i]!=r[i]) return 0;
    }
    return 1;
}
```

Figure 7.7: String comparison.

Scheduler-specific observational determinism policy (SSODP) [NSH13] is a confidentiality policy in multi-threaded programs that defends against an attacker choosing an appropriate scheduler to control the set of possible traces. In particular, given any scheduler and two initial states that are indistinguishable with respect to a secret input (i.e., low-equivalent), any two executions from these two states should terminate in low-equivalent states with equal probability. Formally, given a proposition h representing a secret:

$$\forall\hat{\sigma}.\forall\hat{s}(\hat{\sigma}).\forall\hat{s}'(\hat{\sigma}).\big(h_{\hat{s}} \oplus h_{\hat{s}'}\big) \Rightarrow \bigwedge_{\ell\in L}\big(\mathbb{P}(\Diamond\,\ell_{\hat{s}}) = \mathbb{P}(\Diamond\,\ell_{\hat{s}'})\big) \tag{7.3}$$

where $\ell \in L$ are atomic propositions that classify low-equivalent states and $\oplus$ is the exclusive-or operator. A stronger variation of this policy is that the executions are stepwise low-equivalent:

$$\forall\hat{\sigma}.\forall\hat{s}(\hat{\sigma}).\forall\hat{s}'(\hat{\sigma}).\big(h_{\hat{s}} \oplus h_{\hat{s}'}\big) \Rightarrow \mathbb{P}\Box\Big(\bigwedge_{\ell\in L}\big((\mathbb{P}\bigcirc\ell_{\hat{s}}) = (\mathbb{P}\bigcirc\ell_{\hat{s}'})\big)\Big) = 1. \tag{7.4}$$

Probabilistic conformance describes how well a model and an implementation conform with each other with respect to a specification. As an example, consider a six-sided die. The probability to obtain one possible side of the die is 1/6. We want to synthesize a protocol that simulates the six-sided die behaviour only by repeatedly tossing a fair coin. We know that such an implementation exists [KY76], but we aim to find such a solution automatically by modeling the die as a DTMC and by using an MDP to model all the possible

coin-implementations with a given maximum number of states, including six absorbing final states to model the outcomes. In the MDP, we associate with the states, a set of possible nondeterministic actions, each of them choosing two states as successors with equal probability 1/2. Then, each scheduler corresponds to a particular implementation. Our goal is to check whether there exists a scheduler that induces a DTMC over the MDP, such that repeatedly tossing a coin simulates die-rolling with equal probabilities for all possible outcomes:

$$\exists\hat{\sigma}.\forall\hat{s}(\hat{\sigma}).\exists\hat{s}'(\hat{\sigma}).\Big(init_{\hat{s}} \wedge init_{\hat{s}'}\Big) \Rightarrow \bigwedge_{\ell=1}^{6}\Big(\mathbb{P}(\Diamond(die=\ell)_{\hat{s}}) = \mathbb{P}(\Diamond(die=\ell)_{\hat{s}'})\Big) \tag{7.5}$$

7.4.2 Results and Discussions

All of our experiments in Section 7.4.1 were run on a MacBook Pro with a 2.3GHz quad-core i7 processor with 32GB of RAM. The results are presented in Table 7.1. For our first case study **TA**, described in subsection 7.4.1, models and analyses information leakage in the modular exponentiation algorithm. We experimented with up to four bits for the encryption key (hence, $m \in \{2,4,6,8\}$). The specification checks whether there is a timing channel for all possible schedulers, which is the case for the implementation in `modexp`.

Our second case study **PW**, described in subsection 7.4.1 handles the verification of password leakage through the string comparison algorithm. Here, we experimented with $m \in \{2,4,6,8\}$.

In our third case study **TS**, described in subsection 7.4.1, we assume two concurrent processes. The first process decrements the value of a secret h by 1 as long as the value is still positive, and after this, it sets a low variable ℓ to 1. A second process just sets the value of the same low variable ℓ to 2. The two threads run in parallel; as long as none of them terminates. A fair scheduler chooses for each CPU cycle the next executing thread. This opens a probabilistic thread scheduling channel and leaks the value of h. We compare observations for executions with different secret values h_1 and h_2 (denoted as $h = (h_1, h_2)$).

Case Study		Running time(s)						#SMT variables		#op	#st	#tr
		SE		SS		Total						
		N	O	N	O	N	O	N	O			
TA	$m = 2$	5	2	< 1	< 1	5	2	8088	2520	14	24	46
	$m = 4$	114	18	20	1	134	19	50460	14940		60	136
	$m = 6$	1721	140	865	45	2586	185	175728	51184		112	274
	$m = 8$	12585	952	**TO**	426	**TO**	1378	388980	131220		180	460
PW	$m = 2$	5	2	< 1	< 1	6	3	8088	2520	14	24	46
	$m = 4$	207	26	40	1	247	27	68670	20230		70	146
	$m = 6$	3980	331	1099	41	5079	372	274540	79660		140	302
	$m = 8$	26885	2636	**TO**	364	**TO**	3000	657306	221130		234	514
TS	$h = (0, 1)$	< 1	< 1	< 1	< 1	1	1	1379	441	28	7	13
	$h = (0, 15)$	60	8	1607	< 1	1667	8	34335	8085		35	83
	$h = (4, 8)$	12	3	17	< 1	29	3	12369	3087		21	48
	$h = (8, 15)$	60	8	1606	< 1	1666	8	34335	8085		35	83
	$h = (10, 20)$	186	19	13707	1	13893	20	52695	13095		45	108
PC	s=(0)	277	10	1996	5	2273	15	21220	6780	44	20	188
	s=(0,1)	822	13	5808	5	6630	18	21220	6780		20	340
	s=(0..2)	1690	15	**TO**	5	**TO**	20	21220	6780		20	494
	s=(0..3)	4631	16	**TO**	7	**TO**	23	21220	6780		20	648
	s=(0..4)	7353	22	**TO**	21	**TO**	43	21220	6780		20	802
	s=(0..5)	10661	19	**TO**	61	**TO**	80	21220	6780		20	956
	s=(0..6)	13320	18	**TO**	41	**TO**	59	21220	6780		20	1110

Table 7.1: Experimental results and comparison. **TA:** Timing attack. **PW:** Password leakage. **TS:** Thread scheduling. **PC:** Probabilistic conformance. **TO:** Timeout. **N:** Prototype presented in [ÁBBD20b]. **O:** HyperProb,. **SE:** SMT encoding. **SS:** SMT solving. #op: Formula size (number of operators). #st: Number of states. #tr: Number of transitions.

There is an interesting relation between the data for **TS**. Both the encoding and running time for the experiment are proportional to the higher value in the tuple h.

Our last case study **PC**, described in subsection 7.4.1, is on probabilistic conformance. The input is a DTMC modelling a fair 6-sided die as well as an MDP whose actions model single fair coin tosses with two successor states each. We are interested in finding a scheduler that induces a DTMC that simulates the die outcomes using a fair coin. Given a fixed state space, we experiment with different numbers of actions. In particular, we started from the implementation in [KY76] and for the state space of the die section of the protocol, we added all the possible nondeterministic transitions from the first state to all the other states (denoted $s = 0$), from the first and second states to all the others ($s = 0, 1$), and, similarly scaled it stepwise to include transitions from all states to all others ($s = 0 \dots 6$). Each time,

we were not only able to satisfy the formula, but also obtain the witness corresponding to the scheduler satisfying the property.

In our previous prototypical implementation [ÁBBD20b], due to the encoding of all formulas for all composed states, both the encoding and SMT solving times were significantly higher. Hence, we opted for a timeout for cases where the timing did not seem practically useful. For **TA**, **PW**, and **PC**, we used a timeout of 10000s for the SMT solving.

7.5 Summary

This work brought together the concepts defined in the last chapters into one concrete tool. Here we discussed in detail the construction and working of our tool HyperProb which is the only existing model checker for probabilistic hyperproperties. We have elaborated on the grammar used, how we have defined our models in PRISM language, and described how the tool should be run. As part of future work, we intend to build a graphical user interface for the tool to make it easy to use and accessible.

Chapter 8

Efficient Probabilistic Model Checking for Relational Reachability

Following from the previous chapters, the model checking problem for HyperPCTL is shown to be undecidable, in general, and decidable for memoryless schedulers. The exact computational complexity grows in the polynomial hierarchy with each scheduler quantifier alternation. Intuitively, the computational difficulty of HyperPCTL model checking stems from the fact that the decision problem is indeed a synthesis problem to find schedulers that satisfy relational reachability conditions (i.e., comparison between the probabilities of reachability of certain states). Our SMT-based verification procedure in HyperProb, although exhautive for memoryless and deterministic schedulers, suffers from severe scalability issues. This severely limits the scope of application of algorithms.

On close inspection, many applications, especially in the domain of information-flow security and privacy, fall within certain specific fragments and do not need the full power of the logic. Thus, in this chapter, we aim to answer the following research question:

> Is it possible to identify fragments of probabilistic hyper temporal logics that cover a wide range of applications (e.g., information-flow security) and devise efficient and effective fragment-specific model checking algorithms?

We believe such algorithms will have considerable impact on the accessibility of model check-

ing for probabilistic hyperproperties. With this motivation, we focus on fragments of HyperPCTL capable of expressing popular quantitative information-flow policies. These fragments essentially involve quantitative relational reachability reasoning to determine whether quantified policies and computations reaching certain observations under those schedulers yield equal probability. We investigate the following simple but natural fragments of HyperPCTL and propose algorithms that solve model checking decision problems for them:

- **$(1\sigma 1s)$** Does there exist a scheduler σ, such that under this scheduler, the probability of reaching a public observation a from a state s is the same as the probability of reaching a public observation b from s?
- **$(1\sigma 2s)$** Does there exist a scheduler σ, such that under this scheduler, the probability of reaching an observation a on a public channel from a state s_1 is the same as the probability of reaching an observation b on a public channel from a state s_2?
- **$(2\sigma 2s)$** Do there exist schedulers σ_1 and σ_2, such that the probability of reaching a from s_1 under σ_1 is the same as the probability of reaching b from s_2 under σ_2?

We provide a PTIME algorithm for synthesizing randomized memoryful schedulers for the fragment $(1\sigma 2s)$ or $(2\sigma 2s)$ fragment. For the $(1\sigma 1s)$ fragment, we provide a bounded model checking algorithm viewing MDPs as transformers of distributions of states. We also demonstrate the effectiveness of our algorithms by conducting several case studies, including timing side-channel attacks in the modular exponentiation algorithm, probabilistic attacks in thread scheduling, and violations of secrecy due to scheduling attacks. We observe huge performance improvement (orders of magnitude speed-up) as compared to the general SMT-based technique proposed in [DÁBB22].

8.1 Preliminaries

In addition to the general concepts used across the chapters, here we introduce a few new relevant terminologies. We use $\mathit{Distr}(V)$ to denote the set of all discrete probability

distributions (μ) over the finite set V such that $\mu\colon V \to [0,1]$, and $\sum_{v\in V}\mu(v) = 1$. For paths in MDPs, we use $\mathit{pref}(\pi, i) := s_0\alpha_0 s_1 \ldots \alpha_{i-1}s_i$ to denote the prefix of length i.

We define the *convex addition* [Qua23, p.71] of two schedulers $\sigma, \sigma' \in \Sigma^{\mathcal{M}}$ with weight $\lambda \in [0,1]$ as the scheduler $[\sigma \oplus_\lambda \sigma']$ that initially chooses to behave as σ with probability λ and with the remaining probability chooses to behave as σ'. Formally, for $\pi = s_0\alpha_0 s_1 \ldots \alpha_{n-1}s_n \in \mathit{Paths}^{\mathcal{M}}_{\mathit{fin}}$ and $\alpha \in \mathit{Act}$ we let

$$[\sigma \oplus_\lambda \sigma'](\pi)(\alpha) := \frac{\sigma(\pi)(\alpha)\cdot\lambda\cdot p_{\langle\sigma|\pi\rangle} + \sigma'(\pi)(\alpha)\cdot(1-\lambda)\cdot p_{\langle\sigma'|\pi\rangle}}{\lambda\cdot p_{\langle\sigma|\pi\rangle} + (1-\lambda)\cdot p_{\langle\sigma'|\pi\rangle}} \tag{8.1}$$

where

$$p_{\langle\sigma|\pi\rangle} := \prod_{i=0}^{n-1} \sigma(\mathit{pref}(\pi,i))(\alpha_i)$$

is the probability that σ was chosen given that we have taken path π so far.

A DTMC $\mathcal{D}$ can be viewed as a transformer of distributions. For some $i \in \mathbb{N}$, we define $\mu_i^{\mathcal{D},s_0} \in \mathit{Distr}(\mathcal{S})$ as the distribution after i steps from state $s_0 \in \mathcal{S}$. Formally,

$$\mu_i^{\mathcal{D},s_0}(s) := Pr^{\mathcal{D}}_{s_0}(\{\pi \in \mathit{Paths}^{\mathcal{D}}_{s_0} \mid \pi(i) = s\}), \text{ for } s \in \mathcal{S}. \tag{8.2}$$

One can formalize these policies in the temporal logic HyperPCTL [DÁBB22] as follows:

$(1\sigma 1s)$ $\quad \exists\hat{\sigma}.\ \forall\hat{s}(\hat{\sigma}).\ \iota_{init\,\hat{s}} \implies \mathbb{P}(\Diamond a_{\hat{s}}) = \mathbb{P}(\Diamond b_{\hat{s}})$

$(1\sigma 2s)$ $\quad \exists\hat{\sigma}.\ \forall\hat{s}_1(\hat{\sigma})\forall\hat{s}_2(\hat{\sigma}).\ (\iota^1_{init\,\hat{s}_1} \wedge \iota^2_{init\,\hat{s}_2}) \implies \mathbb{P}(\Diamond a_{\hat{s}_1}) = \mathbb{P}(\Diamond b_{\hat{s}_2})$

$(2\sigma 2s)$ $\quad \exists\hat{\sigma}_1\exists\hat{\sigma}_2.\ \forall\hat{s}_1(\hat{\sigma}_1)\forall\hat{s}_2(\hat{\sigma}_2).\ (\iota^1_{init\,\hat{s}_1} \wedge \iota^2_{init\,\hat{s}_2}) \implies \mathbb{P}(\Diamond a_{\hat{s}_1}) = \mathbb{P}(\Diamond b_{\hat{s}_2})$

In the rest of this chapter, for each fragment, we only consider MDPs where for every state variable $\hat{s}$ (or $\hat{s}_i$) there exists a unique initial state labelled with ι_{init} (or ${\iota_{init}}^i$, respectively).

8.2 Model checking algorithm for $(2\sigma 2s)$ fragment

Consider the model in Fig. 8.1. It shows a fragment of the MDP used to model a side-channel leak using the execution timing of a program. Each of the sub-MDPs represents

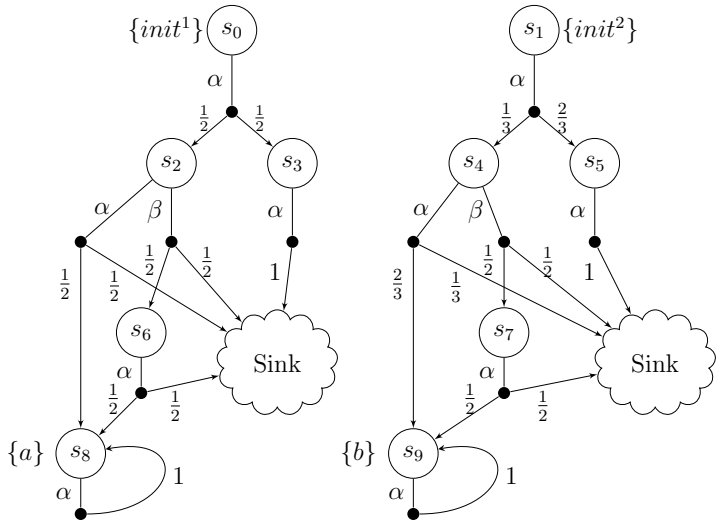


Figure 8.1: (Partial) MDP modelling information leakage via side-channel.

the scheduling between two threads - one where the actual password checking takes place, and the other is an attacker thread that runs parallel to the first thread and checks for its termination. The difference lies in the probability of choosing the threads in two different schedules. Ideally, starting from the initial states labelled ${\iota_{init}}^1$ and ${\iota_{init}}^2$, we should be able to reach the target states labelled a and b, respectively, with equal probability. Formally,

$$\exists \hat{\sigma}_1 \exists \hat{\sigma}_2.\ \forall \hat{s}_1(\hat{\sigma}_1) \forall \hat{s}_2(\hat{\sigma}_2).\ ({\iota_{init}}^1_{\hat{s}_1} \wedge {\iota_{init}}^2_{\hat{s}_2}) \implies \mathbb{P}(\Diamond a_{\hat{s}_1}) = \mathbb{P}(\Diamond b_{\hat{s}_2}) \tag{8.3}$$

In this case, neither $\sigma^{\underline{a}}$ nor $\sigma^{\overline{a}}$ achieves probability equal to $\sigma^{\underline{b}}$ or $\sigma^{\overline{b}}$. However, randomization can be introduced to combine these schedulers to satisfy the equality requirement.

Alg. 17 shows our model-checking algorithm for the $(2\sigma 2s)$ fragment. The algorithm takes as input an MDP and a $(2\sigma 2s)$ HyperPCTL formula specified by the labels of the distinct initial and target states. If a satisfying pair of schedulers exists, the algorithm generates as output a pair of (memoryful) probabilistic schedulers. The algorithm has a polynomial run time in the size of the MDP $\mathcal{M}$. More precisely, it individually model checks $\mathcal{M}$ for each target and then makes a linear pass over the resulting memoryless schedulers. The core idea revolves around combining the minimizing and maximizing schedulers for each target and is based on [Qua23, p.70-75].

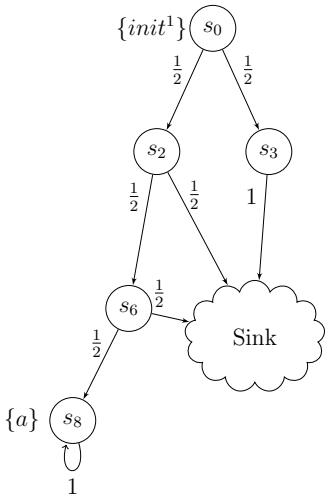
Algorithm 17: PTIME model-checking algorithm for the $(2\sigma 2s)$ fragment.

Input: $\mathcal{M}$: MDP with two initial states $s_1, s_2 \in \mathcal{S}$ with $\iota^1_{init} \in L(s_1), \iota^2_{init} \in L(s_2)$, φ: $(2\sigma 2s)$ formula
Output: Whether $\mathcal{M} \models \varphi$ and, if yes: Witnesses for $\hat{\sigma}_1$, $\hat{\sigma}_2$

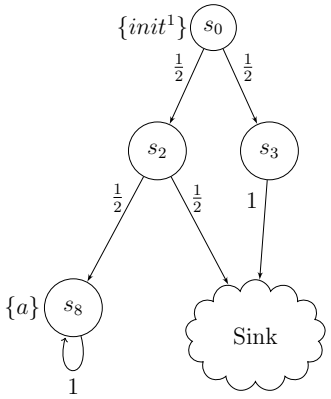
1 **Function** *Main($\mathcal{M}$, φ)*:
2 $\quad \overline{a} := \max_\sigma Pr^\sigma_{s_1}(\Diamond a)$, and $\sigma^{\overline{a}} := \arg\max_\sigma Pr^\sigma_{s_1}(\Diamond a)$
3 $\quad \underline{a}$, $\overline{b}$, $\underline{b}$ and $\sigma^{\underline{a}}$, $\sigma^{\overline{b}}$, $\sigma^{\underline{b}}$: Analogously
4 $\quad$ **if** $I := [\underline{a}, \overline{a}] \cap [\underline{b}, \overline{b}] = \emptyset$ **then**
5 $\quad\quad$ **return** *False*
6 $\quad$ **else**
7 $\quad\quad$ Choose some $c \in I$
8 $\quad\quad$ $\lambda^a := \frac{c-\underline{a}}{\overline{a}-\underline{a}}$, $\lambda^b := \frac{c-\underline{b}}{\overline{b}-\underline{b}}$
9 $\quad\quad$ $\sigma_1 := [\sigma^{\overline{a}} \oplus_{\lambda^a} \sigma^{\underline{a}}]$, $\sigma_2 := [\sigma^{\overline{b}} \oplus_{\lambda^b} \sigma^{\underline{b}}]$
10 $\quad\quad$ **return** *True*, (σ_1, σ_2)

We now describe the algorithm in detail and parallelly illustrate it with the MDP in Fig. 8.1. We intend to verify the property formalized above on $\mathcal{M}$ and, if it holds, return a pair of schedulers as witnesses. In line 2, we calculate the maximum reachability probability $\overline{a}$ for reaching states labelled with a from the relevant initial state and extract the corresponding scheduler in $\sigma^{\overline{a}}$ (this can be done by standard MDP model checking algorithms). For our example MDP, the maximum probability $\overline{a} = \frac{1}{4}$ can be obtained from the DTMC induced (shown in Fig. 8.2b) by $\sigma^{\overline{a}}$ on Fig. 8.1. We analogously calculate the maximum and minimum reachability probability ($\overline{b}$ and $\underline{b}$, respectively) to states labelled b from the other initial state. The corresponding schedulers are extracted as $\sigma^{\overline{b}}$ and $\sigma^{\underline{b}}$. For initial state $init^2$ the minimum probability to reach b, $\underline{b} = \frac{1}{12}$, can be obtained using DTMC induced (as shown in Fig. 8.2c) by $\sigma^{\underline{b}}$, and the maximum probability $\overline{b} = \frac{2}{9}$ can be obtained using DTMC induced (as shown in Fig. 8.2d) by $\sigma^{\overline{b}}$.

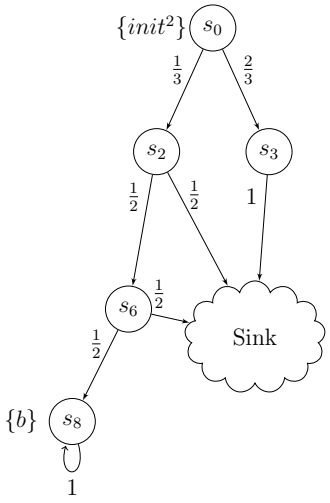
The intersection of the ranges $[\underline{a}, \overline{a}]$ and $[\underline{b}, \overline{b}]$ determines the existence of satisfying schedulers (line 4). If such schedulers can exist, we choose a random value in the intersection (line 7) and construct the schedulers using the convex addition of the respective minimal and maximal schedulers (line 9). In our example, neither of the minimum or maximum schedulers for the respective targets can satisfy the requirement of equal reachability. However,



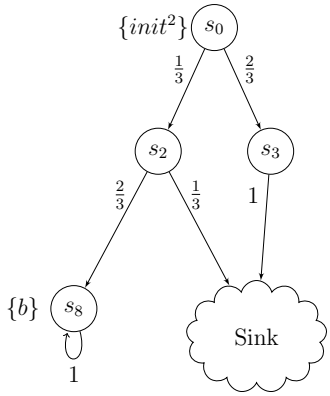
(a) DTMC induced by $\sigma^{\underline{a}}$.



(b) DTMC induced by $\sigma^{\overline{a}}$.



(c) DTMC induced by $\sigma^{\underline{b}}$.



(d) DTMC induced by $\sigma^{\overline{b}}$.

Figure 8.2: DTMCs induced by different schedulers on MDP in Fig. 8.1.

since the ranges $[\frac{1}{8}, \frac{1}{4}]$ and $[\frac{1}{12}, \frac{2}{9}]$ are intersecting, we can generate a randomized scheduler. Assuming $c = \frac{1}{6}$, σ_1 should choose $\sigma^{\overline{a}}$ with probability $\lambda^a = \frac{1}{3}$ and choose $\sigma^{\underline{a}}$ with probability $\frac{2}{3}$. Similarly, σ_2 should choose $\sigma^{\overline{b}}$ with probability $\lambda^b = \frac{3}{5}$ and choose $\sigma^{\underline{b}}$ with probability $\frac{2}{5}$.

The correctness of Alg.17 follows from [Qua23, Le. 3.8]. It runs in polynomial time since we can calculate the minimizing and maximizing schedulers in polynomial time and make a linear pass through them.

Theorem 8.2.1. The model-checking problem for the fragment $(2\sigma 2s)$ over general schedulers is in PTIME.

Remarks. This algorithm would also work if we allowed $s_1 = s_2$. Note that we focus on reachability objectives here, but the algorithm can be extended to any temporal operator since we are individually model checking each temporal objective and combining the obtained schedulers. Hence, the algorithm only needs to be adapted in line 2 to allow computing minimizing and maximizing schedulers for any temporal operator. Extending to general HyperPCTL formulas with nested probability operators is, however, not straightforward: For path formulas with nested probability operators, we cannot employ techniques like value iteration to compute the maximizing and minimizing schedulers. We can adapt Alg. 17 for the $(1\sigma 2s)$ fragment by computing a scheduler σ that behaves like σ_1 on paths starting with s_1 and like σ_2 otherwise.

8.3 Distribution-Based Approach for the $(1\sigma 1s)$ Fragment

In this section, we regard MDPs as transformers of distributions and view $(1\sigma 1s)$ as a distribution-based objective. Recall that the PTIME algorithm for the $(2\sigma 2s)$ fragment (17) does not apply to the $(1\sigma 1s)$ fragment because we have to evaluate two separate probability operators under the *same* scheduler. This is also the reason why a distribution-based view only makes sense for the $(1\sigma 1s)$ fragment but not for $(1\sigma 2s)$ or $(2\sigma 2s)$: Distribution-based objectives can only relate probabilities in a single computation tree, but in the latter fragments, we compare probabilities of two independent computation trees. Inspired by this view, we present a bounded model-checking algorithm for this fragment.

Our inputs include the MDP model $\mathcal{M}$ under consideration, the specification φ of the form $(1\sigma 1s)$ with target states labelled with a, b, and the maximum allowed size of the explored path k. As before, we have a fixed initial state for the model. The algorithm

unfolds the MDP step-by-step as a distribution transformer and checks for the equality of reachability of the targets. The algorithm can be summarized as follows:

- *Backward value iteration*: We use value iteration to track backward reachability from the target states to each state in the $\mathcal{M}$. This helps us to draw an initial conclusion about the existence or absence of the scheduler. The intersection of the minimum and maximum backward reachability values determines if all or none of the schedulers can satisfy the property at hand. In either of these cases we can exit early without further inspection.

- *Guided Forward scheduler search*: In case we cannot exit early, we begin a recursive depth-first search unrolling of $\mathcal{M}$ by exploring an action combination (a set of actions comprised of one action choice per state in the current state distribution). The choice of action combination to be explored is guided by the possible reachability along the path as calculated from the backward reachability values.

Alg. 18 describes the initial setup for the algorithm along with early exit conditions. For each target label, we separately calculate the backward transition matrix for k steps and keep track of the minimum ($V^{\underline{l}}$ and $V^{\underline{r}}$ for the left and right target label, respectively) and maximum reachability ($V^{\bar{l}}$ and $V^{\bar{r}}$ for the left and right target label, respectively) across all actions for each state. Based on these values for the initial state, we can determine if the property is satisfied for all schedulers (line 4) or no scheduler (line 6). If we cannot exit early, we create our initial state distribution to begin searching for a satisfying scheduler following Alg. 19. The state distribution consists of information about the states (hereafter, referred to as *composed* states) we are currently visiting and the probability with which we reached that state. Internally, each *composed* state stores the identifier (*name*) of the original state in $\mathcal{M}$, and the probability with which we have reached the left (P_l) and right (P_r) target state along our path used to visit this state in the current state distribution. For example, initially, this state distribution comprises only the initial state, and we assign the whole

Algorithm 18: Model-checking algorithm for $(1\sigma 1s)$ fragment

Input: $\mathcal{M} = (S, Act, \mathbf{P}, \mathsf{AP}, L)$: MDP model, k: step bound, *init*: initial state of $\mathcal{M}$, φ: $(1\sigma 1s)$ formula.
Output: Scheduler satisfying $\mathcal{M} \models \varphi$, if found, else null.

1 **Function** *Main($\mathcal{M}$, k, init, φ)*:
2 $V^{\underline{l}}, V^{\bar{l}} :=$ GetBackwardValueIteration$(\mathcal{M}, a, k)$
3 $V^{\underline{r}}, V^{\bar{r}} :=$ GetBackwardValueIteration$(\mathcal{M}, b, k)$
4 **if** $V^{\underline{l}}(init), V^{\bar{l}}(init), V^{\underline{r}}(init), V^{\bar{r}}(init)$ *are equal* **then**
5 **return** *True: All schedulers satisfy the property*
6 **else if** $[V^{\underline{l}}(init), V^{\bar{l}}(init)] \bigcap [V^{\underline{r}}(init), V^{\bar{r}}(init)] = \emptyset$ **then**
7 **return** *False: No such scheduler exists*
8 **if** $a \in L(init)$ **then**
9 $P_l := 1$
10 **else**
11 $P_l := 0$
12 **if** $b \in L(init)$ **then**
13 $P_r := 1$
14 **else**
15 $P_r := 0$
16 scheduler := ForwardSchedulerSearch($\mathcal{M}$, 1, k, V^l, V^r, $[\{init, P_l, P_r\} : 1], a, b$)
`// Assuming we combine` $V^{\underline{l}}$ `and` $V^{\bar{l}}$ `in` V^l`, and` $V^{\underline{r}}$ `and` $V^{\bar{r}}$ `in` V^r
17 **return** *scheduler*

probability mass of 1 to it as shown in line 16. We additionally remember the reachability to target states in the initial states as shown in lines 11, 15.

We begin our guided depth-first scheduler search in the recursive method shown in Alg. 19. If we have already reached path length k (line 2), we check for the equality between the total reachability of left and right targets across all states in the current distribution (this can be modified to check whether the values differ by at most ϵ). Based on this we can either conclude we have found a scheduler (line 3) and start backtracking to collect the actions taken along this path (line 13), or that no such scheduler is possible in this path (line 5) and backtrack to explore other options (line 11). For cases that have not reached the k bound, we would ideally want to generate the set of *all* combinations of actions for the states in the current distribution st_{dist} (line 6) and explore for equality. However, to optimize the process, we use Alg. 20 to filter action combinations that cannot lead to equality in reachability

Algorithm 19: Forward Scheduler Search

Input: $\mathcal{M}$: MDP model, i: current iteration, k: step bound V^l, V^r: Backward reachability to target states, st_{dist}: distribution of states in the current step along with the values for P_l, P_r reached in the last $i-1$ steps, a: left target label, b: right target label.

Output: Scheduler satisfying $\mathbb{P}(\Diamond a) = \mathbb{P}(\Diamond b)$ in k-steps, if found, else null.

1 **Function** *ForwardSchedulerSearch($\mathcal{M}$, i, k, V^l, V^r, st_{dist}, a, b)*:

2 **if** $i == k$ *and* $\sum_{st \in st_{dist}} st.P_l == \sum_{st \in st_{dist}} st.P_r$ **then**

3 | **return** *[ ]* `// empty scheduler`

4 **else**

5 | **return** *null*

6 *priority_queue* := CheckNextActions$(\mathcal{M},\ V^l,\ V^r,\ st_{dist},\ a,\ b)$

`// Naive approach:` *priority_queue* := `GetAllNextActions`$(\mathcal{M},\ st_{dist},\ a,\ b)$

7 **while** *priority_queue is not empty* **do**

8 | $curr_{act}, curr_{state}$:= *priority_queue.pop*()

9 | *scheduler* := ForwardSchedulerSearch$(\mathcal{M},\ i+1,\ k,\ V^l,\ V^r,\ curr_{state}, a, b)$

10 | **if** *scheduler is null* **then**

11 | | Go to line 7

12 | **else if** $|scheduler| \geq 0$ **then**

13 | | Append $curr_{act}$ to *scheduler*

14 | | **return** *scheduler*

15 | **return** *null*

values. We then explore each action combination in the filtered set in a depth-first manner (line 8).

In Alg. 20, we utilize the backward reachability values to determine if a specific action combination is a good candidate. For a naive exhaustive search among action combinations, line 3 would have sufficed. However, we can use the values calculated in the backward value iteration step to determine the minimum and maximum reachability values from a particular state. This can help us filter out action combinations that do not have an intersecting range of reachability values. For each new set of actions for the current state distribution (line 4), we generate its corresponding one-step next state distribution (line 5). We calculate the minimum and maximum reachability for each state in this new distribution based on whether we have already reached the target states in the history of states used to reach this current state: (1) In case we did encounter the target states, we store their value in the P_l and P_r

Algorithm 20: Check Next Action

Input: $\mathcal{M}$: MDP model, V^l, V^r: Backward reachability to left and right target states, st_{dist}: current state distribution, a: left target label, b: right target label.

Output: Prioritized list of action combinations for all states in st_{dist}.

1 **Function** *CheckNextAction*($\mathcal{M}$, V^l, V^r, st_{dist}, a, b)**:**
2 $priority\_queue := [\,]$, $flag :=$ true, $act_{old} := [\,]$, $st_{old} := [\,]$
3 $act\_dist = \times_{st \in st_{dist}} Act(st.name)$
4 **for** $\alpha_{dist} \in act\_dist$ **do**
5 $new_{dist} :=$ CreateStateDistribution($st_{dist}, st_{old}, \mathcal{M}, \alpha_{dist}, act_{old}, flag, a, b$)
6 $flag :=$ false, $v^{\underline{l}} := 0$, $v^{\bar{l}} := 0$, $v^{\underline{r}} := 0$, $v^{\bar{r}} := 0$
7 **for** $s \in new_{dist}$ **do**
8 **if** $s.P_l > 0$ **then**
9 $[v^{\underline{l}}, v^{\bar{l}}] += [s.P_l, s.P_l]$
10 **else**
11 $[v^{\underline{l}}, v^{\bar{l}}] += [st_{dist}[s] * V^{\underline{l}}(s.name), st_{dist}[s] * V^{\bar{l}}(s.name)]$
`// Set` $[v^{\underline{r}}, v^{\bar{r}}]$ `analogously`
12 **if** $s.P_l > 0$ *and* $s.P_r > 0$ *for all states in* new_{dist} **then**
13 Add $(\alpha_{dist}, new_{dist})$ to *priority_queue* with priority 1
14 **else if** $[v^{\underline{l}}, v^{\bar{l}}] \subseteq [v^{\underline{r}}, v^{\bar{r}}]$ *(or vice versa)* **then**
15 Add $(\alpha_{dist}, new_{dist})$ to *priority_queue* with priority 2
16 **else if** $[v^{\underline{l}}, v^{\bar{l}}] \subset [v^{\underline{r}}, v^{\bar{r}}]$ *(or vice versa)* **then**
17 Add $(\alpha_{dist}, new_{dist})$ to *priority_queue* with priority 3
18 $act_{old} := \alpha_{dist}$, $st_{old} := new_{dist}$
19 **return** *priority_queue*

section of the *composed* state and use them as the final reachability values along this path (line 9), (2) otherwise, we calculate the minimum and maximum values using the probability value of the state in the current distribution (st_{dist}) and the values from the backward reachability matrices (line 11). As we go through this filtering, we further prioritize these action combinations based on these values for the new state distribution produced: (1) In case we have encountered both the target states along the paths leading to all the states in the current distribution (line 12), we explore this action with the highest priority as further distribution transformation would add no more information. (2) In case the range of values for reaching one target state is a subset of the range for the other target (line 14), explore this option with second priority as there is at least one common value that we can surely

reach. (3) In case the range of value for reaching one target state is a proper subset of the range for the other target (line 16), we explore this option with the least priority.

Algorithm 21: Create State Distribution

Input: st_{dist}: current state distribution, st_{old}: state distribution created by act_{old} on st_{dist}, $\mathcal{M}$: MDP model, α_{dist}: set of actions choices we are currently processing, $flag$: denoting if we are generating a new state distribution or changing an existing one, a: left target label, b: right target label.

Output: new_st_{dist}: state distribution by applying α_{dist} on st_{dist}.

1 **Function** *CreateStateDistribution*$(st_{dist}, st_{old}, \mathcal{M}, \alpha_{dist}, act_{old}, flag, a, b)$**:**
2 **if** $flag == true$ **then**
   `// Generating new state distribution`
3 $new\_st_{dist} := [\,]$
4 **for** $i \in [\,|st_{dist}|\,]$ **do**
5 $succ_s := \{s' \mid \mathbf{P}(st_{dist}[i].name, \alpha_{dist}[i], s') > 0\}$
6 **for** $s' \in succ_s$ **do**
7 **if** $s' \in \{st.name | st \in new\_st_{dist}\}$ **then**
8 $new\_st_{dist}[s'] += st_{dist}[i][val] \ * \ \mathbf{P}(st_{dist}[i].name, \alpha_{dist}[i], s')$
9 **else**
10 $new\_st := \{name : s', P_l : 0, P_r : 0\}$
11 **if** $st_{dist}[i].P_l$ *is 0 and* $a \in L(s')$ **then**
12 $new\_st.P_l = st_{dist}[i][val] \ * \ \mathbf{P}(st_{dist}[i].name, \alpha_{dist}[i], s')$
13 **else if** $st_{dist}[i].P_l > 0$ **then**
14 $new\_state.P_l = st_{dist}[i].P_l$
   `// Set` P_r `value for the composed state analogously`
15 Append $(new\_state : val)$ to new_st_{dist}
16 **else**
   `// Modifying existing state distribution`
17 **for** $i \in \{index \mid act_{old}[index] \neq \alpha_{dist}[index]\}$ **do**
18 Remove successors of $act_{old}[i]$ and adjust val in st_{old}
19 Add successors of $\alpha_{dist}[i]$ in st_{old}
20 **return** new_st_{dist}

In Alg. 21, we elaborate on the creation of the state distribution by applying α_{dist} on st_{dist}. Note that this state distribution consists of *composed* states of the form $(name, P_l, P_r)$, and that the corresponding probability mass accumulated when reaching these states along the path is referred to as val. As denoted by the $flag$ in line 2, when we call this method for the first time on st_{dist}, we collect the successors of each state in the distribution (line 5). When creating the distribution, we either create a new corresponding *composed* state (line 10) and

update the corresponding reachability to target values along the path (lines 11-15), or we update the value of the corresponding *composed* state that already exists in the distribution (line 8). Once we have created an initial distribution for st_{dist}, we only focus on the actions that have changed between the old action combination act_{old} considered in the previous call and the currently considered α_{dist} (line 17). To this effect, we remove the states or change the value of the probability for the states associated with the action from act_{old} (line 18) and then add the new successor or modify its value in the distribution as required (line 19).

8.4 Experimental Evaluation

In this section, we give a brief overview of the different case studies and properties we use to evaluate the effectiveness of our algorithms. Our case studies are motivated by the models evaluated in [DÁBB22]. We further discuss the experimental results and compare their pros and cons with existing solutions.

8.4.1 Case Studies

Thread scheduling (TS)

$$\boxed{\textbf{while } h > 0 \textbf{ do } \{h \leftarrow h - 1\};\ l \leftarrow 2 \quad || \quad l \leftarrow 1}$$

Consider the two threads above where h is a secret input and l is the publicly observable output. Under uniform scheduling, we would be able to observe the different values of l with different probabilities ($l = 2$ with a higher probability). This difference in observation probability will increase as the value of h increases. We verify two different properties for this model.

- We allow multiple different scheduling probabilities and want to synthesize randomized schedulers such that starting from different values of h, we can observe a specific value of l with equal probability. Formally,

  $\mathbf{TS - 2\sigma 2s}$: $\exists\hat{\sigma}_1\exists\hat{\sigma}_2.\ \forall\hat{s}_1(\hat{\sigma}_1)\forall\hat{s}_2(\hat{\sigma}_2).\ (h^1_{\hat{s}_1} \neq h^2_{\hat{s}_2}) \implies \mathbb{P}(\Diamond(l = 1)_{\hat{s}_1}) = \mathbb{P}(\Diamond(l = 1)_{\hat{s}_2}$

```
void mexp(){
        c = 0; d = 1; i = k;
        while (i >= 0){
                i = i-1;
                c = c*2;
                d = (d*d) % n;
                if (b(i) = 1){
                        c = c+1;
                        d = (d*a) % n;
                }
        }
}
t = new Thread(mexp());
j = 0; m = 2 * k;
while (j < m & !t.stop)
j++;
```

Figure 8.3: Modular exponentiation.

- We allow multiple probabilistic scheduling distributions and want to synthesize a scheduler such that, starting from a specific h value we can observe $(l = 1)$ and $(l = 2)$ with equal probability. Formally,

  $\mathbf{TS-1\sigma 1s}$: $\exists\hat{\sigma}.\ \forall\hat{s}(\sigma).\ h_{\hat{s}} \implies \mathbb{P}(\Diamond(l=1)_{\hat{s}}) = \mathbb{P}(\Diamond(l=2)_{\hat{s}})$

Side-channel timing attack (TA)

For security purposes, we want to be able to avoid opening a channel for side-channel leaks based on the execution timing of a program. Consider the modular exponentiation algorithm in RSA (method *mexp* shown in Fig. 8.3). The absence of an *else* block allows for the opening of a side-channel timing leak. An attacker can run a thread parallel to the main thread to infer the execution time of the methods (loosely considered similar to the number of execution steps in which *mexp* terminates).

We aim to ensure we can verify the existence of scheduler(s) that provide the same distribution over observable outputs, independent of the secret bit value used to calculate it. This ensures similarity in the execution time of the program, thus avoiding disclosure of information about the secret input. We experiment with two different properties.

```
int str_cmp(char * r){
        char * s = 'Bg\$4\0';
        i = 0;
        while(s[i]!='\0'){
                if(r[i]=='\0' ||    s[i]!=r[i])
                return 0;
                i++;
        }
        if(r[i]=='\0')
        return 1;
        return 0;
}
```

Figure 8.4: String comparison.

- We want to synthesize a pair of schedulers (σ_1, σ_2) starting at states with different b values and terminating with the same value for a specific bit of the counter j in the parallel thread. We use Alg. 17 to generate this pair of randomized schedulers.

  TA $-$ **2**σ**2s**: $\exists\hat{\sigma}_1\exists\hat{\sigma}_2.\ \forall\hat{s}_1(\hat{\sigma}_1)\forall\hat{s}_2(\hat{\sigma}_2).\ (b^1_{\hat{s}_1} \neq b^2_{\hat{s}_2}) \implies \mathbb{P}(\Diamond(j=1)_{\hat{s}_1}) = \mathbb{P}(\Diamond(j=1)_{\hat{s}_2})$

- For the same secret b value, we want to verify if all schedulers can produce different publicly observable outputs with the same probability. We utilize Alg. 18 to verify this property.

  TA $-$ **1**σ**1s**: $\exists\hat{\sigma}.\ \forall\hat{s}(\sigma).\ b_{\hat{s}} \implies \mathbb{P}(\Diamond(j=1)_{\hat{s}}) \neq \mathbb{P}(\Diamond(j=2)_{\hat{s}})$

Password leakage (PW)

Consider the code snippet in Fig. 8.4 of a leaky string-matching implementation. The code allows a side channel leakage as it returns *false* at the first encounter with a non-matching character between the input and secret value. This can provide indirect information to the attacker about the length of the actual secret string. We can use a counter running parallel to this code (similar to TA) to observe the difference in execution time. We experiment with the following properties:

- We want to verify the existence of a scheduler such that we are to observe the same

value of the *counter* with equal probability when beginning with different secret values.

$\mathbf{PW-2\sigma 2s}$: $\exists\hat{\sigma}_1\exists\hat{\sigma}_2.\ \forall\hat{s}_1(\hat{\sigma}_1)\forall\hat{s}_2(\hat{\sigma}_2).\ (b^1_{\hat{s}_1} \neq b^2_{\hat{s}_2}) \implies$

$\mathbb{P}(\Diamond\, counter1_{\hat{s}_1}) = \mathbb{P}(\Diamond\, counter1_{\hat{s}_2})$

- We want to verify if, for a secret value, the probability of observing different bits of the counter is not the same.

  $\mathbf{PW-1\sigma 1s-1}$: $\exists\hat{\sigma}.\ \forall\hat{s}(\sigma).\ b_{\hat{s}} \implies \mathbb{P}(\Diamond\, counter1_{\hat{s}}) \neq \mathbb{P}(\Diamond\, counter0_{\hat{s}})$

- Changing **PW**-1σ1s-1 slightly, we want to verify if we can find a scheduler where the probability of reaching different counters for the same secret value is the same.

  $\mathbf{PW-1\sigma 1s-2}$: $\exists\hat{\sigma}.\ \forall\hat{s}(\sigma).\ b_{\hat{s}} \implies \mathbb{P}(\Diamond\, counter1_{\hat{s}}) = \mathbb{P}(\Diamond\, counter2_{\hat{s}})$

TaskShuffler (TF)

During task scheduling in Real-Time Operating Systems (RTOS), fixed models of scheduling can give rise to faster execution times and easy detection of fault by examining minor perturbations. However, this predictability can give rise to security issues. Determinism in the scheduling process gives rise to predictability and can provoke attackers to use the timing properties of the scheduler to design targeted attacks. Taskshuffler [YMCS16] presents an algorithm to allow randomly scheduling lower priority tasks in a queue while ensuring higher priority tasks still make their deadline.

	p_i	e_i	d_i	V_i
τ_0	5	1	5	4
τ_1	8	2	8	3
τ_2	20	3	20	4

Table 8.1: Parameters used for scheduling three tasks τ_0, τ_1, τ_2.

Consider the parameters in Table 8.1; for each of the tasks τ_0, τ_1, τ_2, e_i is the worst-case execution time, d_i is the relative deadline, p_i is the inter-arrival time of successive job releases by the task. The subscript of each task name represents its priority, i.e., τ_0 has a higher priority than τ_1 which in turn has a higher priority than τ_2. Using (p_i, e_i, d_i), the algorithm

calculates V_i, the maximum amount of time, for each task such that (1) a lower priority task can be scheduled before the current task, and (2) the current task does not miss its deadline. We refer the readers to the original paper [YMCS16] for details on the algorithm.

We experiment with the following variation of noninterference:

- We want to verify if the probability of creating two distinct schedules, where all tasks finish their executions, is equal. Here *init* refers to the initial state of the scheduling where the respective variables in the algorithm are initialized with the parameters in the table 8.1. We differentiate the schedules using $doneWithV_0$ and $doneWithV_1$ denoting that the tasks finished without using all the time permitted by V_0 and V_1 respectively.

  $\mathbf{TF-1\sigma 1s}$: $\exists\hat{\sigma}.\ \forall\hat{s}(\sigma).\ init_{\hat{s}} \implies \mathbb{P}(\Diamond\, doneWithV_{0\hat{s}}) = \mathbb{P}(\Diamond\, doneWithV_{1\hat{s}})$

- Another interesting property (although not a hyperproperty) is to verify if all jobs meet their deadline under this protocol. Formally, we can represent this as below. We set each job's counter to it e_i and count backward at each step of its execution. We use *allDone* to denote the state where all three jobs have exhausted their e_i counter.

  $\mathbf{TF-prop}$: $\forall\hat{\sigma}.\ \forall\hat{s}(\sigma).\ init_{\hat{s}} \implies \mathbb{P}(\Diamond\, allDone_{\hat{s}}) = 1$

8.4.2 Experiments and Results

We have implemented our prototypes for Alg. 17 and Alg. 18 in Python. Our input models are in the PRISM language [KNP11] and we utilize the model labels to describe our specifications (in terms of initial and target states). For Alg. 17, we have used the model checking capabilities of Stormpy [stob] and combined the minimum and maximum schedulers generated. For Alg. 18, we have utilized Stormpy to parse and store the MDP models. To evaluate the effectiveness of our algorithms, we compare our results with [DABB21]. Note that we are not necessarily generating the same type of schedulers - [DABB21] generates memoryless deterministic schedulers, and our algorithms generate memoryful randomized

Case Study	Secret Scaled on	Model Size #st	#tran	#act	Scheduler Possible?	HyperProb Time(s)	Alg. 17 Time(s)
TA-2σ2s	1 bit	24	46	30	✓	0.65	0.047
	3 bits	112	274	154	✓	18.80	0.051
	5 bits	264	694	374	✓	523.65	0.048
	8 bits	612	1684	884	✓	**TO**	0.054
	32 bits	8580	25156	12740	✓	**TO**	0.107
	50 bits	20604	60904	30704	✓	**TO**	0.152
TS-2σ2s	(h_1, h_2)=(1,0)	7	17	11	✓	0.10	0.049
	(h_1, h_2)=(15,0)	35	115	67	×	1.37	0.043
	(h_1, h_2)=(20,10)	45	150	87	✓	4.39	0.049
	(h_1, h_2)=(50,3)	106	362	208	×	22.67	0.044
PW-2σ2s	1 bit	24	46	30	✓	0.61	0.039
	3 bits	140	302	182	✓	28.31	0.046
	5 bits	352	782	462	✓	1132.56	0.057
	8 bits	850	1922	1122	✓	**TO**	0.072
	32 bits	12610	29186	16770	✓	**TO**	0.147

Table 8.2: Experimental results for scheduler generation using Alg. 17. **#st**: number of states in the model, **#tran**: number of transitions in the model, **#act**: number of actions in the model.

or memoryful deterministic schedulers. However, the comparison based on execution time seems appropriate to showcase the scalability of our algorithms. The column for *scheduler possible* notes the actual existence of a scheduler satisfying the corresponding property. All experiments were done on a MacBook Pro with a 2.3GHz i7 processor and 32GB of RAM. The execution times are in seconds. We considered a timeout (**TO**) of 5000s.

Table 8.2 shows the results for our experiments using Alg. 17 and our three case studies described earlier. We can generate (if possible) randomized schedulers efficiently and the current size of our models does not pose any challenge to this approach compared to the results of HyperProb [DABB21]. The efficiency of the implementation is mainly attributed to the strength of model checking and scheduler generation of Storm. The incorporation of randomization positively impacts the performance of model checking for this fragment by orders of magnitude. Note that the two compared approaches generate different types of schedulers, however, they always agree on the existence of a scheduler. For example, **TS**-2σ2s for $(h_1, h_2) = (15, 0)$ does not produce a scheduler for both the approaches. Evidently,

Case Study	Secret Scaled on	Model Size #st	#tran	#act	Scheduler Possible?	HyperProb Time(s)	Alg. 18 k	Alg. 18 Time(s)
TA-1σ1s	1 bit	25	48	31	✓	0.77	7	0.156
	3 bits	113	276	155	✓	18.91	11	1.059
	5 bits	265	696	375	✓	408.96	15	6.657
	8 bits	613	1686	885	✓	**TO**	21	335.000
TS-1σ1s	$h_1 = 1$	8	19	12	×	0.16	6	0.068
	$h_1 = 15$	36	117	68	×	1.66	19	0.568
	$h_1 = 30$	66	222	128	×	7.78	34	1.776
	$h_1 = 50$	106	362	208	×	53.09	54	4.684
PW-1σ1s-1	1 bit	25	48	31	✓	0.71	6	0.402
	3 bits	141	304	183	✓	30.83	10	1.297
	5 bits	353	748	463	✓	1201.00	14	3.927
	8 bits	851	1924	1123	✓	**TO**	20	12.840
PW-1σ1s-2	3 bit	141	304	183	×	43.04	11	**TO**
	5 bits	353	748	463	×	1230.00	15	**TO**
TF-1σ1s	default	54	89	77	✓	0.32	10	0.40

Table 8.3: Experimental results for scheduler generation using Alg. 18. **#st**: number of states in the model, **#tran**: number of transitions in the model, **#act**: number of actions in the model.

Alg. 17 outperforms HyperProb by orders of magnitude. In cases where a scheduler cannot be found, one does not exist, meaning that the formula is falsified by both HyperProb and Alg. 17.

Table 8.3 shows the results for our experiments using Alg. 18. Compared to [DABB21], the strength of our approach lies in two aspects: (1) An initial backward value iteration helps decide cases where all or none of the schedulers can satisfy the property as can be seen in the case of **TS**-1σ1s, and (2) the verification focuses on one set of actions at a time as a possible candidate to pursue in the distribution transformer; this makes it possible to produce a scheduler quicker than [DABB21], where the whole model has to be encoded before solving. This is evident in the cases **TA**-1σ1s and **PW**-1σ1s-1. However, due to the inherent hardness of the general problem, this might not always be effective as seen in **PW**-1σ1s-2. Both algorithms considered in this fragment are exhaustive searches across all possible schedulers of their class. [DABB21] encodes this scheduler set before attempting to solve the encoding. Thus, although time-consuming, it still produces a result. Since our

algorithm considers one action combination at a time for exploration, it can benefit from further optimization in this aspect.

We verified an aspect of the correctness of TaskShuffler stated using **TF**-*prop* in 0.22 seconds using HyperProb. Since it does not fall under the fragment we consider in this chapter, we did not execute it with the algorithms in the context here.

8.5 Summary

The general model checking problem for HyperPCTL is undecidable. Based on our exploration across different applications, fragments of the language for which we can create scalable algorithms exist. In this work, we focused on identifying specific fragments of HyperPCTL that are expressive enough to cover interesting security properties while being simple enough to find scalable algorithms for them. We presented two different such algorithms - a convex addition of minimum and maximum schedulers to produce randomized schedulers, and a distribution transformation-based algorithm to produce deterministic schedulers. We have provided comparison of our results with existing approaches to show the scalability of our approaches.

Chapter 9

Lightweight Verification of Hyperproperties

In the last few chapters, we discussed the challenges of model checking probabilistic hyperproperties. In this chapter, we focus on an approximate statistical approach to model check probabilistic hyperproperties.

9.1 Introduction

In the last decade, researchers have proposed several adaptations of classical temporal logics to specify hyperproperties in a formal and systematic way. Examples in the non-probabilistic setting are HyperLTL [CFK+14] and its asynchronous variant A-HLTL [BCBS21]. HyperLTL extends LTL [Pnu77] with explicit quantification over paths that allows to express relations among execution traces from independent system's runs. Recent works in [FRS15, HSB21, HBFS23a] provide exhaustive and bounded model-checking algorithms for HyperLTL. For probabilistic hyperproperties, there are two main specification languages: HyperPCTL [ÁB18, DÁBB22], which quantifies over schedulers and argues over computation trees, and Probabilistic HyperLogic (PHL) [DFT20] which adds quantifiers for schedulers and reasons about traces. In both contexts, these approaches face two main challenges: scalability and the need for an explicit model. Scalability is, in particular, critical: (1) HyperLTL model checking is EXSPACE-complete [BF18], (2) HyperPCTL

and A-HLTL model checking are in general undecidable with decidable fragments in EXSPACE [BCBS21, DÁBB22], (3) PHL model checking is in general undecidable with decidable fragments (reduce to HyperCTL*) in NSPACE [DFT20, FRS15]. This complexity obstacle has been a major motivation for the development of alternative approaches to handle the problem. One possible approach is to provide an approximate result with certain statistical guarantees, termed SMC. SMC is an approximate model checking method that is subject to a small probability of drawing a wrong conclusion [LLT$^+$19, LL20, LDB10]. The main idea is to simulate finitely many traces of a model and conduct *hypothesis testing* to conclude if there is enough evidence that the model satisfies or violates the property, subjecting to a small probability of drawing a wrong conclusion. Such simulation-based approaches have two main advantages: first, we can use them to approximate the probability of satisfying the desired property in a model of considerable size, which we would be otherwise unable to verify exhaustively; second, we can apply them to black-box systems for which we are unable to access the inner model. This approach is also intuitive as it can terminate early for cases where it has already found enough evidence for violation. Consider, a case where a property is required to hold for all traces. In this case, we should not be able to see a violation even if we simulate just one trace. Given these advantages, we want to study its application to verify hyperproperties. Another challenge, in terms of verification, is the handling of nondeterminism. When modeling systems, we have to take into consideration the uncertainty that can arise due to incomplete details, involvement of unknown agents, or noise, in general. From a verification perspective, we need to be able to argue that a property holds under any such possibility of nondeterministic uncertainty. Both HyperPCTL and A-HLTL model checking have the capability of reasoning over nondeterminism, however, the high complexity in their model checking solutions stems from the need for *scheduler* synthesis.

In this work, we chose to model systems as MDP to effectively express uncertainty in systems using nondeterminism and randomization. PLASMA [LS14] is a model checker that uses a memory-efficient sampling of schedulers [DLST15] to conduct simulation-based

statistical analysis. In this work, we extend PLASMA's capability to include the verification of linear, bounded hyperproperties over systems modeled as MDP. Our method orchestrates well-established methods from the SMC community for the analysis of an expressive model class in light of bounded HyperLTL properties. The result is a scalable, lightweight verification approach which is the first of its kind to handle this combination of model class and property. We have added and experimented with an extension that supports using recorded traces or requesting simulation of black-box components on-the-fly for hyperproperty verification. This opens the door to utilizing our approach for applications in cases where explicit modeling is not possible or error-prone. For evaluation, we present a diverse set of scaling benchmarks that raise the demand for this expressive model class and property type. We have selected systems that allow for verification of properties such as *noninterference*, *side-channel information leak*, *opacity*, and *anonymity*. The systems under inspection range from classical examples including dining cryptographers, to examples taken from robotics path planning and real code snippets. The state space of the resulting models varies in the order of magnitude from tens to hundreds of billions involving tens to thousands of nondeterministic actions. Our experimental evaluation indicates good performance on systems, unperturbed by the size of the state space. To summarize, our *main contributions* are:

1. To the best of our knowledge, we provide the first SMC approach for the verification of unquantified and bounded HyperLTL properties involving nondeterminism.

2. We extend the model checker PLASMA by this class of properties. Furthermore, we add capabilities to execute black-box verification.

3. We showcase the general applicability with an extensive evaluation of our method on various scalable case studies taken from different domains.

9.2 Preliminaries

In this section, we first elaborate on the syntax and semantics of the logic we use to specify our properties. This is followed by a brief description of the Sequential Probability Ratio Test which is a standard method in statistical hypothesis testing that forms the core of our approach.

9.2.1 HyperLTL

HyperLTL [CFK+14] is the extension of linear-time temporal logic (LTL) that allows the expression of temporal specifications involving relations between multiple paths. Each state in the path is observed as a set of atomic propositions that hold in that state. HyperLTL involves the evaluation of specifications over these propositions. An arbitrary path variable π is used to refer to individual paths that can be generated by the model. Contrary to LTL, each proposition a^π is associated with a path variable π denoting the path on which it should be evaluated.

Syntax

We focus on unquantified and bounded HyperLTL defined by the grammar below.

$$\varphi := \ \mathsf{a}^\pi \mid \neg\varphi \mid \varphi \wedge \varphi \mid \bigcirc\varphi \mid \varphi \, \mathcal{U}^{\leq k} \varphi$$

- $\mathsf{a} \in \mathsf{AP}$ is an atomic proposition that evaluates to *true* or *false* in a state;
- π is a *random path variable* from an infinite supply of such variables Π;
- $\bigcirc$, $\Diamond^{\leq k}$, $\Box^{\leq k}$, and $\mathcal{U}^{\leq k}$ are the 'next', 'finally', 'global', and 'until' temporal operators, respectively,
- $k \in \mathbb{N}$ is the path length within which the operator has to be evaluated.

Following are the connectives defined as syntactic sugar:
$\mathtt{true} \equiv \mathsf{a}^\pi \vee \neg\mathsf{a}^\pi$, $\varphi \vee \varphi' \equiv \neg(\neg\varphi \wedge \neg\varphi')$, $\varphi \Rightarrow \varphi' \equiv \neg\varphi \vee \varphi'$, $\Diamond^{\leq k}\varphi \equiv \mathtt{true}\mathcal{U}^{\leq k}\varphi$, $\Box^{\leq k}\varphi \equiv$

$\neg \Diamond^{\leq k} \neg\varphi$. We denote $\mathcal{U}^{\leq\infty}$, $\Diamond^{\leq\infty}$, and $\Box^{\leq\infty}$ or the unbounded temporal operators by $\mathcal{U}$, $\Diamond$, and $\Box$, respectively. In our work, we consider only the bounded fragment of HyperLTL such that for all temporal operators (except $\bigcirc$), we evaluate the result on finite fragments of the simulated traces.

Semantics

The path evaluation function $V : \Pi \to \mathcal{S}^\omega$ assigns each path variable π, a concrete path of the labeled DTMC Below we consider the semantics of HyperLTL,

$$
\begin{aligned}
V &\models \mathsf{a}^\pi && \text{iff } \mathsf{a} \in L\big(V(\pi)[0]\big) \\
V &\models \neg\varphi && \text{iff } V \not\models \varphi \\
V &\models \varphi_1 \wedge \varphi_2 && \text{iff } V \models \varphi_1 \text{ and } V \models \varphi_2 \\
V &\models \bigcirc \varphi && \text{iff } V^{(1)} \models \varphi \\
V &\models \varphi_1 \mathcal{U}^{\leq k} \varphi_2 && \text{iff there exists } i \in [0,k], V^{(i)} \models \varphi_2 \\
& && \quad \text{and for all } j \in [0,i),\ V^{(j)} \models \varphi_1
\end{aligned}
$$

where $V^{(i)}$ is the i-shift of path assignment V defined by $V^{(i)}(\pi) = (V(\pi))^{(i)}$. For example, the formula $V \models \mathsf{a}_1^{\pi_1} \mathcal{U}^k \mathsf{a}_2^{\pi_2}$ means that a_1 holds on the path $V(\pi_1)$ until a_2 holds on the path $V(\pi_2)$ in k steps.

9.2.2 Sequential Probability Ratio Test

We use Wald's SPRT [Wal45]. The idea is to continue sampling until we are either able to conclude or we have exhausted a user-provided sampling budget. Assuming we want to verify if a property φ holds on our model $\mathcal{M}$ with probability greater than and equal to θ, i.e., $\mathbb{P}_{\mathcal{M}}(\varphi) \geq \theta$. To use SPRT in this case, we add an indifference region around our bound to create two distinct and flexible hypothesis tests [AP18]. For a given indifference region ϵ, we define $p_0 = \theta + \epsilon$ and $p_1 = \theta - \epsilon$. Our resultant hypotheses are,

$$H_0 : \mathbb{P}_{\mathcal{M}}(\varphi) \geq p_0 \quad H_1 : \mathbb{P}_{\mathcal{M}}(\varphi) \leq p_1 \tag{9.1}$$

Using these newly created bounds, we define the following probability ratios,

$$ratio_t = \frac{p_1}{p_0} \quad ratio_f = \frac{1-p_1}{1-p_0} \tag{9.2}$$

We define an indicator function $\mathbf{1}(T \models \varphi) \in \{0, 1\}$ that returns 1 if the trace T satisfies the property φ, and returns 0 otherwise. When evaluating φ on a set of sampled traces $\{T_1, \ldots, T_n\}$, we accumulate $ratio_t$ if $\mathbf{1}(T \models \varphi) = 1$ and $ratio_f$ otherwise. Assuming, we have sampled N traces, the final product of the truth value corresponds to

$$p_{ratio} = \prod_{i=1}^{N} \frac{(p_1)^{\mathbf{1}(T_i \models \varphi)}(1-p_1)^{\mathbf{1}(T_i \models \neg\varphi)}}{(p_0)^{\mathbf{1}(T_i \models \phi)}(1-p_0)^{\mathbf{1}(T_i \models \neg\phi)}} \tag{9.3}$$

We iteratively calculate this ratio until the exit condition is met. To restrict the error in the estimation of the probability θ, we specify error probabilities α as the maximum acceptable probability of incorrectly rejecting a true H_0, and β as the maximum acceptable probability of accepting a false H_0. The boundary error ratios can be defined as $A = \beta/(1-\alpha)$ and $B = (1-\beta)/\alpha$. To reach a conclusion, we accept H_0 if $p_{ratio} \leq A$, and accept H_1 if $p_{ratio} \geq B$. The case for specifications with $\mathbb{P}_{\mathcal{M}}(\varphi) \leq \theta$ is similar except we use the reciprocals of $ratio_t$ and $ratio_f$.

9.3 Problem Formulation

HyperLTL allows explicit quantification over traces, allowing the user to express whether they want their specification to hold across all paths associated with a path variable or in at least one of those paths. Along with the added expressiveness, this formulation distends existing challenges - (1) While checking a specification across all possible sets of paths provides a robust verification result, it is considerably expensive, making it impractical as we scale to models with larger state spaces. (2) Most real-life systems involve uncertainties in the form of randomization, nondeterminism, or partial observability. Consequently, this raises a need to express that, for instance, a fraction of the paths of the system satisfy the specification.

To handle the above challenges, we propose a practical formulation for expressing unquantified and bounded HyperLTL formulas for models that involve both probabilistic choices and nondeterminism. We quantify over the path variables by associating a probabilistic bound denoting the proportion of the set of traces that should satisfy a given specification. We can express that a specification is *almost always likely* or *highly unlikely* by adjusting the bound of the probability p to $p \geq 1$ or $p \leq 0$, respectively. Intuitively, *almost always likely* can be considered as a weaker counterpart of $\forall$ (forall) quantification, and *highly unlikely* can be considered as a weaker counterpart of $\neg\exists$ (existential) quantification over path variables. Note that these limits our expression of HyperLTL formulas with quantifier alternation in any capacity, and we leave that as an aspect worth exploring in future works.

Consider an MDP $\mathtt{M}$ and an unquantified, bounded HyperLTL formula φ that contains path variables $(\pi_1, \ldots, \pi_m)$. We consider tuples of m schedulers to simulate m traces assigned to these path variables, i.e., we have a one-to-one correspondence between schedulers and path variables. We are interested in checking if there *exists* a combination of schedulers that can satisfy the HyperLTL specification φ on our model within a given probability bound. Formally, this can be expressed as,

$$\exists\sigma_1 \exists\sigma_2 \ldots \exists\sigma_m \; \mathbb{P}_{\mathtt{M}}(V \models \varphi) \sim \theta \tag{9.4}$$

where $\theta \in [\epsilon, 1-\epsilon]$ to allow an indifference region for hypothesis testing (see 9.2.2), σ_i are schedulers of $\mathtt{M}$, $V(\pi_i)$ is the path drawn from the DTMC $\mathtt{M}_{\sigma_i}$ for $i \in [n]$ which is induced by σ_i on MDP $\mathtt{M}$, and $\sim \in \{\geq, \leq\}$. Note that we can involve multiple models to yield paths for each scheduler σ_i. For properties where we want to check if a given specification holds across all scheduler combinations, we negate our specification to re-formulate the problem as in Eq. 9.4. Since we adopt a statistical model checking algorithm, it is worth noting that we cannot directly observe if a specification holds *for all* cases, thus, we utilize this approach to check if we can satisfy its negation. We elaborate on this in Section 9.4.

9.4 Approach

We utilize the advantages of SMC to verify hyperproperties by answering our model checking problem using hypothesis testing, specifically SPRT, as described in Sec. 9.2.2. The overall approach involves the sampling of schedulers and traces from one (or more) given MDP, monitoring the satisfaction of the property on these traces, and determining if we have gathered enough evidence to reach a concrete verdict for the property. In this section, we explain the concepts and parameters involved in finding the result of this test such that we can directly use it to answer our model checking question.

9.4.1 Scheduler sampling

One of the main challenges when verifying MDP is the generation of schedulers for verification. It stems from the complexity involved in the storage of history to resolve non-determinism in the current state. We utilize the lightweight scheduler sampling feature of PLASMA [LS14]. This approach avoids the explicit storage of schedulers by using *uniform* PRNG to resolve non-determinism and hashes as seeds for the PRNG. In the following, we will give an intuition of the approach inbuilt in PLASMA and how we have extended it to argue about hyperproperties.

PRNG forms the core of the smart sampling algorithm of PLASMA. Given a set of possible action choices and sufficient runs of the number generator, they allow the generation of statistically independent numbers that are uniformly distributed across a specified range. They are uniquely mapped to their seed values int_{sch}, ensuring the reproduction of the same value when the generator is provided with the same seed. Note that we can use PRNG to identify individual schedulers but cannot identify specific schedulers. Furthermore, since the seeds only initialize the PRNG, using problem-specific information (e.g., about the property) during the generation of the seed does not allow related schedulers.

Each state of the MDP is internally represented as a concatenation of the bits representing the values of the atomic propositions that are true at that state. A sequence of states can

be represented by concatenating their bit sequences. The sum of the bits of such a sequence of numbers int_t, which is an integer, represents a trace. Concatenation of int_{sch} and int_t can be then used to uniquely identify both a scheduler and a trace. PLASMA generates a hash with this concatenated value which represents the history of both the scheduler and the trace and is used as a seed to resolve the next nondeterministic choice. PLASMA uses an efficient iterative hash using modular arithmetic that ensures efficient storage of the possible schedulers mapping the comparatively large set of schedulers to a smaller set of integers with a low probability of collision. For more details on this, we refer the reader to [DLST15]. Once the nondeterministic choice is resolved at a state, PLASMA uses an independent PRNG to uniformly choose a successor state from the ones available under the chosen action. This is concatenated with the trace before generating the next hash for the nondeterministic resolution.

When working with hyperproperties, we would need to consider a tuple of schedulers and traces. In this aspect, we can either simulate traces from the same MDP using different schedulers, use different schedulers for each MDP, or a combination of both. We define a *scheduler tuple* $\underline{\sigma} \subset \Sigma^m$ as a tuple of schedulers sampled from the set of possible schedulers allowed by our MDP and m is the number of scheduler quantifiers in our specification as shown in Eq. 9.4. We define a *trace tuple* as a tuple of traces sampled from our model based on the tuple of schedulers. Thus, $\omega^{\underline{\sigma}}$ represents the trace tuple ω, sampled from the DTMC induced by the scheduler tuple $\underline{\sigma}$. For simplicity, we consider a one-to-one correspondence between our schedulers and MDP. We define an indicator function $\mathbf{1}(\omega^{\underline{\sigma}} \models \varphi) \in \{0, 1\}$ that returns 1 if the trace tuple $\omega^{\underline{\sigma}}$ satisfies the hyperproperty φ, and returns 0 otherwise.

The aim is to verify the satisfaction of the given specification under all or some combination of nondeterministic choices in our system. Since a scheduler represents a concrete resolution of nondeterminism across the system, our problem is transformed to that of finding a scheduler tuple that satisfies our specification in the form of the described hypothesis in Eq. 9.4. Intuitively, SMC considers the proportion of the sampled trace tuples that in-

dividually satisfy the property to estimate the true satisfaction probability in the overall model. To bind the errors in the estimation, the algorithm uses precision and user-provided error margins.

For the case where we want to conclude all scheduler tuples satisfy the property, we negate the property and try to find a scheduler tuple that satisfies this negated property. The falsity of this property makes our original property true. For the case where we want to search if there exists a scheduler tuple, we pose the hypothesis directly. However, in this case, a false result does not necessarily guarantee the absence of a witness to the specification; it suggests that our algorithm was unable to find such a scheduler tuple within the given budget, error, and precision bound. Note that we cannot derive the exact scheduler tuple (we get the traces generated but not the reduced DTMC) due to the black-box nature of our sampling. We can only reason about its existence or absence within the given budget.

9.4.2 Implementation

In this section, we discuss the handling of the hypothesis testing of H_0 as shown in Eq. 9.1 in detail. The case for H_1 (also shown in Eq. 9.1) is similar except we use the reciprocals of $ratio_t$ and $ratio_f$. As shown in Alg. 22, we begin by initializing the necessary parameters (line 2). For conducting sequential hypothesis testing on large systems, we need an additional bound to represent the maximum limit of resources we want to spend on this verification. To this end, PLASMA utilizes the concept of a user-provided *budget*. Following the idea described in [DLST15], the algorithm automatically distributes the budget to determine the number of schedulers and trace tuples the verification should consider as described in the previous section. We generate a set of scheduler tuples $\sum$ and create a mapping to store which scheduler should be used to produce which trace (deriving this information from the input specification). In lines 3-4, for each scheduler tuple, we use the internal simulator to simulate the traces as specified by the mapping. In the case of multiple initial states, we allow the choice of traces with the same or different initial states. This reduces extra

Algorithm 22: Hypothesis testing on Hyperproperties

Input : MDP model: $\mathtt{M}$, spec: φ, Hypothesis $H_0 \colon \mathbb{P}_{\mathtt{M}}(V \models \varphi) \geq \theta$
α, β: desired type I, type II errors,
$\mathcal{N}_{max}$: simulation budget, ϵ: indifference region.

Output: Success: There exists a satisfying scheduler tuple,
No success: Could not find a satisfying scheduler tuple,
Inconclusive: No *conclusive* scheduler tuple was found

1 **Function** *Main(*$\mathtt{M}$, φ, α, β, $\mathcal{N}_{max}$, ϵ*)*
2 initialize() ▷ Initializes $\mathcal{N}, \mathcal{M}, p_0, p_1, A, B, k, ratio_t, ratio_f$
3 $\underline{\Sigma} \leftarrow \{\mathcal{M}$ tuples of k randomly chosen seeds$\}$
4 $\forall \underline{\sigma} \in \underline{\Sigma}, \forall i \in \{1, ..., \mathcal{N}\} : \omega_i^{\underline{\sigma}} \leftarrow \text{simulate}(\mathtt{M}, \varphi, \underline{\sigma})$
5 $R \leftarrow \{(\underline{\sigma}, n) | \underline{\sigma} \in \underline{\Sigma} \wedge \mathbb{N} \ni n = \sum_{i=1}^{\mathcal{N}} \mathbf{1}(\omega_i^{\underline{\sigma}} \models \varphi)\}$
6 **if** *canEarlyAccept(R)* **then**
7 Accept H_0 and exit
8 $\underline{\Sigma} \leftarrow \{\underline{\sigma} \in \underline{\Sigma} | R(\underline{\sigma}) > 0\}, \mathcal{M} \leftarrow |\underline{\Sigma}| + 1$ ▷ Remove null schedulers
9 **if** $|\underline{\Sigma}| = 0$ **then**
10 Quit: No suitable scheduler-tuple found
11 **while** $|\underline{\Sigma}| > 1$ **do**
12 initializeSchedulerBasedBounds() ▷ Initializes $\alpha_M, \beta_M, A_M, B_M$
13 **foreach** $\underline{\sigma} \in \underline{\Sigma}$, $i \in \{1, \ldots, |\underline{\Sigma}|\}$ **do**
14 $ratio_i \leftarrow 1$
15 **foreach** $j \in \{1, \ldots, \mathcal{N}\}$ **do**
16 **if** *simulate*$(\mathtt{M}, \varphi, \underline{\sigma}) \models \varphi$ **then**
17 $ratio \leftarrow ratio \cdot ratio_T$; $ratio_i \leftarrow ratio \cdot ratio_T$
18 **else**
19 $ratio \leftarrow ratio \cdot ratio_F$; $ratio_i \leftarrow ratio \cdot ratio_F$
20 **if** $ratio_i \leq A_M$ *or* $ratio \leq A$ **then**
21 Accept H_0 and quit: scheduler found
22 **else if** $ratio_i \geq B_M$ **then**
23 Quit iteration for $\underline{\sigma}$: Scheduler tuple rejected
24 **if** *All schedulers were rejected* **then**
25 Quit: No scheduler found in given budget
26 $\underline{\Sigma} \leftarrow \text{filter}(\underline{\Sigma})$ ▷ Keep only the best-performing scheduler tuples
27 Inconclusive: There exists a scheduler that was neither accepted nor rejected

sub-formulas on the property to decide on initial states and allows us to verify the property only on relevant trace samples. In line 5, we use a custom model checker that we have implemented in PLASMA to verify linear, bounded HyperLTL properties on sets of traces sent as input. We further allow n-ary boolean operations by extending the general idea of AND, OR, XOR, etc., to reduce the length of input property the user has to provide.

In line 6 of the algorithm, we compare the ratio generated using Eq. 9.3 against error boundary A to check if we have already found enough witnesses to accept our null hypothesis H_0. We do not check against boundary B because the absence of a satisfying scheduler in this initial phase does not ensure that the possibility of finding such a scheduler is zero. It hints at the need for further sampling. In line 8, we filter out the null schedulers, i.e., for which none of the trace tuples satisfied the property. Since we are looking for a scheduler tuple to satisfy the property with positive probability, null schedulers cannot definitely be our best search options. For each scheduler tuple in this filtered set, we again sample $\mathcal{N}$ trace tuples. We essentially conduct multiple independent hypothesis tests, one for each scheduler tuple. Hence, similar to [DLST15], we modify the error for each scheduler to $\alpha_M = 1 - \sqrt[\mathcal{M}]{1-\alpha}$, $\beta_M = 1 - \sqrt[\mathcal{M}]{1-\beta}$ to account for the error correction needed. In the initial phase (lines 3-10), the idea was to check if we can satisfy the boundaries A, B using the truth value of all trace tuples sampled, irrespective of its scheduler. In the rest if the algorithm, we check if we can individually accept or reject any scheduler tuple, alongside the global check for satisfaction across all sampled trace tuples. Since our trace tuples return an overall true/false for the whole tuple, the error bound for each scheduler tuple would not change when we are working with alternation-free hyperproperties instead of trace properties.

In lines 17 and 19, we re-calculate p_{ratio} (as in Eq. 9.3), both for each scheduler tuple and for all the sampled trace tuples. As we encounter a satisfying tuple of traces, our overall p_{ratio} decreases as $ratio_t$ is a value less than one in this case and with each non-satisfying trace tuple, it increases. If the ratio obtained over all sampled traces across all schedulers is reduced below A or its scheduler counterpart is below A_M, we either have found a scheduler tuple that satisfies the property or over all the sampled trace tuples, we have found enough evidence to conclude that our hypothesis H_0 is satisfied.

At the end of the iteration over the scheduler tuples, we can quit the test if all our scheduler tuples are rejected, or proceed to the next iteration with only the *best* scheduler tuples. We rearrange our scheduler tuples in an ascending order based on the number of

trace tuples that satisfied φ. Since we are aiming to find a scheduler tuple that satisfies φ with a probability greater than θ, we only keep the first half of rearranged scheduler tuples, ensuring that we are looking only at the schedulers that have a higher chance of exceeding the bound. If our evaluation reaches line 27, the set Σ would contain one scheduler which we were neither able to accept nor reject, reaching an inconclusive decision about H_0 within the given budget and precision margins. This inconclusive result would indicate we have to retry the experiment with a higher simulation budget and/or different precision and error margins for further scrutiny.

Convergence

The algorithm will always terminate in a finite number of iterations as we eliminate half of our candidate scheduler tuples at each iteration. However, it may not have found a satisfying scheduler tuple within that boundary. For an MDP `M` and property φ, we want to find a *good* scheduler tuple, i.e., one that satisfies φ with probability $p \geq \theta - \epsilon$. Assuming we have $|\S|$ possible scheduler tuples, and $|\S_g|$ *good* schedulers, we use $\mathbb{P}\colon \S \to [0,1]$ to denote the probability with which a scheduler tuple satisfies φ. If we sample $\mathcal{M}$ scheduler tuples and $\mathcal{N}$ trace tuples per scheduler tuple, the probability of sampling a trace tuple from a *good* scheduler tuple that satisfies φ is,

$$\underbrace{\left(1 - \left(1 - \frac{|\S_g|}{|\S|}\right)^{\mathcal{M}}\right)}_{\text{good scheduler tuple}} \underbrace{\left(1 - \left(1 - \frac{\sum_{\sigma \in \S_g} \mathbb{P}_\sigma}{|\S_g|}\right)^{\mathcal{N}}\right)}_{\text{trace satisfies } \varphi} \tag{9.5}$$

We aim to maximize the value of this probability by optimizing the values of $\mathcal{M}$ and $\mathcal{N}$, across which the budget $\mathcal{N}_{max}$ is the total number of sampling we want to permit. Since we need to find schedulers that satisfy the property with probability at least θ, we set $\mathcal{N} = \lceil \frac{1}{\theta} \rceil$. This ensures that we spend our sampling budget verifying scheduler tuples that have a higher probability of satisfying our property. For example, if our θ is 0.25, $\mathcal{N} = 4$. If we want to

check for our specification to be $\geq \theta$, any scheduler that satisfies at least 1 of the 4 sampled traces should be a good candidate for a *good* scheduler. In case we want to check for our specification to be $\leq \theta$, finding such *good* schedulers would help us reject the hypothesis easily. We allocate the rest of the budget (such that $\mathcal{N} \cdot \mathcal{M} \sim \mathcal{N}_{max}$) to sample scheduler tuples, thus, we set $\mathcal{M} = \lceil \theta \mathcal{N}_{max} \rceil$. We have experimented with various values of budget, adjusting them based on the expected accuracy of our results.

9.5 Case Studies

In this section, we discuss case studies to show the applicability and scalability of our approach. One of the main advantages of statistical model checking lies in the fact that we do not necessarily need access to the underlying model to verify a system. This allowed us to utilize our approach on sets of traces generated from black-box sources. We have separated our case studies into two sections elaborating on the models of the grey-box (where we have access to the underlying model) and black-box (where we just have access to a set of traces generated by different schedulers) examples.

9.5.1 Grey-box verification

Group Anonymity in Dining Cryptographers (DC)

We explored the dining cryptographers problem [Cha88] from the perspective of how it is designed to maintain anonymity. In this model, three cryptographers go out for dinner and at the end, want to figure out who paid the bill (their manager or one of them) while respecting each other's privacy. The protocol proceeds in two stages:(1) each cryptographer flips a coin and only informs the cryptographer on their right of the outcome (head or tail), (2) the cryptographers consider both the coin tosses that they know of, to declare *agree* in case the tosses were the same, or *disagree* otherwise. However, in the case of the cryptographer who actually paid, they would declare the opposite conclusion.

Given an odd number of cryptographers, we should have an odd number of agrees if the

manager pays the bill, an even number of agrees if one of the cryptographers paid, and vice versa for an even number of cryptographers. We want to verify if there is any information leakage depending on which cryptographer pays. In the model, we nondeterministically choose who pays the bill and the order in which the cryptographers toss their coin. In case one of the cryptographers pays in both traces, we expect the parity of coins at the end to be the same. As described in [BP09], the order of coin toss should not affect the anonymity in the protocol. This good behaviour can be expressed as a hyperproperty,

$$\varphi_{DC} = \Big(\bigvee_{i\in(1,2,3)} Cpay_{i\pi_1} \wedge \bigvee_{i\in(1,2,3)} Cpay_{i\pi_2} \Big) \implies$$
$$\Diamond(done \wedge (c1 \oplus c2 \oplus c3))_{\pi_1} \bigwedge \Diamond(done \wedge (c1 \oplus c2 \oplus c3))_{\pi_2} \tag{9.6}$$

For the correctness of the model, we should not be able to find a scheduler that satisfies the bad behaviour $\neg\varphi_{DC}$ with positive probability, thus, we design the hypothesis as,

$$\exists\sigma_1.\exists\sigma_2.\ \mathbb{P}_{\mathtt{M}}(V \not\models \varphi_{DC}) > 0 \tag{9.7}$$

We expect this property to be false for an odd number of cryptographers and true for even, as our model should ensure anonymity. We experimented with both unbiased and biased coins in the model to check if that affects the parity of agreement. The main challenge for this study was the size of the models as shown in Table 9.1. Existing exhaustive approaches would take considerable memory and execution time to verify this model. Hence, an approximate approach like SMC has its utility here.

Noninterference in path planning (RNI)

Consider the grid in Fig. 9.1 which represents two robots moving across a two-dimensional plane subdivided (discretized) into $n \times n$ cells. The robots can nondeterministically choose to move to their neighbouring cells unrestricted (up, down, left, or right) unless it is blocked by the grid boundaries. However, with a certain error probability, the chosen target cell is

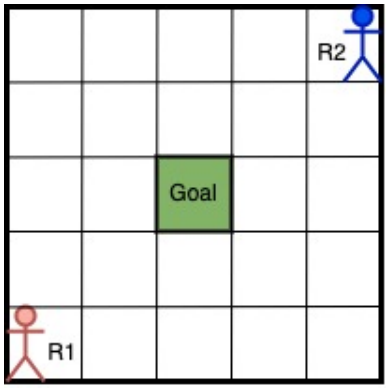


Figure 9.1: Two robots attempting to reach the same goal.

not reached and instead, the robot stays in its current cell. The grid can hence, be modeled as an MDP where each state models a grid cell. Note that we do not restrict or force any specific strategy for the movement of these robots. Thus, each scheduler corresponds to a specific strategy that defines how the robot moves across the grid. We consider the case where two robots (R1 and R2) are placed in opposite corners of the grid and aim to reach the goal state at the center of the grid. Assume $R1$ is our robot of interest and $R2$ is an intruder. Motivated by the idea in [DFT20], a specification of interest would be to check if the plan of $R1$ to reach the goal is affected by the plans of $R2$.

We design the hypothesis as the negation of the required property. Hence, we want to determine if there exists any such scheduler tuple where the movement of $R1$ would be similar but $R1$ fails to reach the goal in one of them. The unquantified HyperLTL formula is as follows,

$$\varphi_{RNI} = \square \left(actR1_{\pi 1} = actR1_{\pi 2} \right) \bigwedge (\neg goalR1_{\pi 1} \mathcal{U} goalR2_{\pi 1}) \oplus (\neg goalR1_{\pi 2} \mathcal{U} goalR2_{\pi 2}) \quad (9.8)$$

For any arbitrary probability p, we design our specification as,

$$\exists \sigma_1 . \exists \sigma_2 .\ \mathbb{P}_{\mathtt{M}}(V \vDash \varphi_{RNI}) > p \quad (9.9)$$

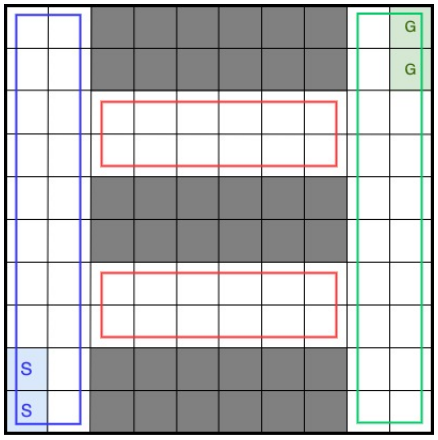


Figure 9.2: Grid divided into regions to ensure opacity.

Current state opacity (CSO)

Consider the grid in Fig. 9.2 where we use only one robot on the grid, which starts from any of the starting states labeled S and aims to reach the opposite corner labeled G. The gray boxes represent obstacles. Instead of analysing reachability, we are interested in analysing opacity similar to [WNP20]. Opacity requires that an unauthorized user should not be able to realize the current state of the system. In the context of a robot, opacity ensures privacy is preserved as the robot moves across the grid. An observer gets an observation corresponding to each movement of the robot. Note that we have divided the grid into three regions (blue: *near initial*, red: *between obstacles*, green: *near goal*) which would generate the same observation even when the robot is in a different position.

Current state opacity specifically states that while starting from the same initial state (here: either of the lower left corners marked in blue), it is still feasible to move across the grid using different paths that can produce the same observation. By different paths, we refer to cases where the actual positions of the robot are different due to the execution of different actions (up, down, left, right). This would mean that an intruder should not be able to guess the exact location by merely gathering observations about the movement of

the robot. We can express this formally as,

$$\varphi_{CSO} = \big(start_{\pi_1} \wedge start_{\pi_2}\big) \bigwedge \neg \Box_{\leq k}(act_{\pi_1} = act_{\pi_2}) \bigwedge \Box_{\leq k}(region_{\pi_1} = region_{\pi_2}) \tag{9.10}$$

where *act* encodes the action taken by the robot on the grid and *region* denotes the corresponding region observed. We want to check if any such combination of schedulers exists that satisfies the current state opacity with respect to a given threshold. This is expressed as,

$$\exists \sigma_1 . \exists \sigma_2 .\ \mathbb{P}_{\mathtt{M}}(V \models \varphi_{CSO}) > p \tag{9.11}$$

9.5.2 Black-box verification

We use the example of a side-channel timing attack on a password checker as a black-box case study. We consider several password checkers that vary in the expected amount of information leaked by observing the execution time, resulting from different input guesses. We ran our password checkers on a microcontroller and considered numerical passwords of length 10 as input.

Following the approach described in Section 9.4.1, a scheduler is represented as a seed for a pseudo-random number generator. For a black-box model, the model checker calls a `python` script with one parameter (the scheduler seed) as an input. This seed is used by the model to resolve nondeterminism internally via a pseudo-random number generator. Internally, the script uses the scheduler seed to create a random password guess. Here, we assume the password guess and the actual password is of the same length to simplify code run on the microcontroller. The number of correct digits of the generated password guess is saved and the password is forwarded to the microcontroller via the serial interface over USB. The execution time of the microcontroller together with the number of correct bits are returned by the Python script and the out-stream is parsed and interpreted by the model checker.

We convert numerical return values (rounded to a predefined level of precision), e.g., the number of correct digits (cd) or the execution time (et) to traces whose length of consecutive symbols of a type reflects those values. For instance, the returned pair of values `execution_time=4, correct_digits=1` would be converted into the trace

$$\{et, cd\} \rightarrow \{et\} \rightarrow \{et\} \rightarrow \{et\} \rightarrow \{\} \rightarrow \dots$$

Leakage of information from an unsafe password checker can be obtained by observing the execution times for several inputs. Intuitively, if the checking of a password with more consecutive correct digits (in the front) takes longer than a password with fewer correct digits, observing the execution time for multiple guesses should allow guessing the correct password. To formalize this, we use the specification of unwanted behavior

$$\varphi_{TAM} = (\Diamond(cd_{\pi_1} \wedge \neg cd_{\pi_2}) \wedge \Diamond(et_{\pi_1} \wedge \neg et_{\pi_2})) \oplus (\Diamond(cd_{\pi_2} \wedge \neg cd_{\pi_1}) \wedge \Diamond(et_{\pi_2} \wedge \neg et_{\pi_1})) \tag{9.12}$$

Consider the example of a password checker that leaks information (BB-L) in Listing 9.1. In contrast, a simple, safe approach (BB-S) checks the whole password without the option of an early return as in Line 6 and thus always produces the same execution time regardless of the correctness of the guess g.

```
1 bool checkPassword(String g){
2         int i;
3         for(i=0; i < g.length(); ++i)
4         {
5                 if(g[i]!=secret[i])
6                 return false;
7         }
8         return true;
9 }
```

Listing 9.1: Possible leaky password checker (BB-L).

Additionally, we can also add padding to obfuscate actual execution timing. In our experiments, we consider a random delay (BB-*R) between 0 and 10 microseconds or a fixed

delay (BB-*F) of 2 microseconds. We want to check, for an arbitrary probability p, whether a combination of schedulers exists, such that bad behaviour, i.e., information leakage can be derived. This is expressed as,

$$\exists\sigma_1.\exists\sigma_2.\ \mathbb{P}_{\mathtt{M}}(V \models \varphi_{TAM}) > p \tag{9.13}$$

9.6 Experimentation/Evaluation

The model details of our grey-box case studies have been reported in Table 9.1. Experimental results for our case studies have been summarized in Table 9.2. The parameters in Table 9.2 refer to the number of schedulers ($\#sch$) and traces ($\#tr$) that were sampled as determined by our algorithm, and the length of the trace (k) as determined by the user based on knowledge about the model. We separately report the time required for the sampling of the scheduler (Sim) and trace tuples and the time required to verify (Ver) the hyperproperty on them. Reported timing data is the average over 10 runs. Note that in our evaluation we do not compare our results to the existing model checkers for linear hyperproperties as they cannot handle probabilistic models with non-determinism.

9.6.1 Black-box verification

Experiments were run on an Intel® Core™ i7 (6x3.30 GHz) with 32Gb RAM, the password checkers ran on an `esp32` micro-controller to alleviate variance in timing due to process scheduling. To obtain results with higher precision, we execute using multiple parameter configurations - the size of the indifference region (ϵ), the satisfying probability (θ), and sampling budget ($\mathcal{N}_{max}$). The error probabilities α, β were kept at 0.01 for the whole experiment. Results and running times for the most accurate runs are shown in Table 9.2, where different variants of password checkers (see also Table 9.5) are referenced by their labels.

In total, we have run over 480 combinations of parameters to synthesize accurate results. Table 9.2 lists the results of parameter configurations that maximize the probability of sat-

isfying the property without being inconclusive to give an estimate on a worst-case scenario. In case the property could be satisfied in the majority of the 10 runs, we show results for two configurations: one leading to a large observed probability and a second one that used a higher budget and smaller indifference region which, thus, can be expected to be more precise.

From the results, we can observe that for safe password checking the tested variants with no padding (S), fixed padding (SF), and random padding (SR) do not allow information leakage about the correctness of the password guess via correlation of the observed execution time. In contrast to this, the experiments with a leaky password generator with a fixed or no padding scheme (LF, L) allow correlating execution time and correctness of passwords. Note that in most cases the created guesses had only zero to one correct digit, as we did not implement adversarial strategies to guess larger parts of the password.

9.6.2 Grey-box verification

Experiments were run on an Intel® Core™ i7 (4x2.3 GHz) with 32Gb RAM. We ran experiments on each of the described case studies by scaling them across the different parameters involved. However, due to space constraints, we report cases that are sufficient to show the scalability and robustness of our approach.

The DC component in the tables 9.2 and 9.1 corresponds to the verification of the dining cryptographers protocol described in Section 9.5.1. Our specification φ_{DC} should not hold for an odd number of cryptographers and should hold for even ones. We have scaled the model over $\#c = \{4, 7, 8, 15\}$ and witnessed the expected results. We used a constant budget of 5000 for all the cases reported. We used models directly from PRISM [KNP11] and were able to verify them without alterations. We experimented with both biased and unbiased coins. The result produced was the same proving that the biases of the coins do not affect the outcome of the protocol. The experiment for the biased coin scenario used the exact same parameters as reported and yielded similar execution times for both scenarios.

CS	Param.	#States	#Transitions	#Actions
DC	$c = 4$	2598	6864	5448
	$c = 7$	328760	1499472	1186040
	$c = 8$	1687113	8790480	6952248
	$c = 15$	10^{11}	10^{12}	9×10^{11}
RB Grid Fig.9.1	$n = 3$	1034	4888	2444
	$n = 5$	12346	77152	38576
	$n = 10$	256926	1852972	926486
RB Grid Fig.9.2	$n = 10$	200	1440	720
	$n = 20$	800	6080	3040
	$n = 30$	1800	13920	6960

Table 9.1: Model details of grey-box case studies.

The RNI section in the tables 9.2 and 9.1 corresponds to noninterference case study. We have scaled our grid for $N = \{3, 5, 10\}$. We verify the existence of scheduler tuples that fail to satisfy noninterference with probability bounds of $\{0.1, 0.2, 0.5\}$ with a budget of 2000. The trace lengths have been increased in proportion to the grid sizes. We have experimented on arbitrary trace lengths which have been adjusted as we increase the grid size. As we do not specify any smart movement strategy for the robots, these results are based on possible random walks the robots can make on the grid. The interesting observation here is the difference in execution time based on the parameters. The cases for $\theta \leq 0.1$ seem to be challenging, given the current grid and budget, resulting in an inconclusive result; for $n = 10$ our experiment ran for more than 2 days hinting at an inconclusive result within the given budget. This is expected as we are challenging the algorithm to find a scheduler with a very low probability (between 0 and 0.1) given the large search space. For $\theta \geq 0.2$, we are often able to find our target scheduler tuples in the initial sampling phase, leading to short execution time due to early exit. This is mainly because we are looking for a scheduler across a wider range of probability (between 0.2 and 1). Using $\theta \geq 0.5$ becomes challenging when scaling the model (with the same budget for comparison) due to the growing number of possible scheduler tuples, and the lack of any specific strategy that finds traces where both the robots are aiming to reach the goal. Thus, finding a scheduler with probability on

Case Study	Specification		Result	Parameters			Time [sec]	
				#sch	*#tr*	*k*	Sim.	Ver.
BB	$Pr(V \vDash \varphi_{TAM}) \geq 0.1 \pm 0.01$	# S	False	400	10	80	108.00	0.09
	$Pr(V \vDash \varphi_{TAM}) \geq 0.1 \pm 0.01$	# SF	False	400	10	80	93.10	0.09
	$Pr(V \vDash \varphi_{TAM}) \geq 0.1 \pm 0.01$	# SR	False	400	10	80	92.80	0.07
	$Pr(V \vDash \varphi_{TAM}) \geq 0.3 \pm 0.1$	# L	True	1201	4	80	102.00	0.10
	$Pr(V \vDash \varphi_{TAM}) \geq 0.25 \pm 0.01$	# L	True	1001	4	80	85.50	0.01
	$Pr(V \vDash \varphi_{TAM}) \geq 0.15 \pm 0.1$	# LF	True	601	7	80	92.00	0.09
	$Pr(V \vDash \varphi_{TAM}) \geq 0.1 \pm 0.01$	# LF	Undec	400	10	80	90.00	0.08
	$Pr(V \vDash \varphi_{TAM}) \geq 0.1 \pm 0.01$	# LR	False	400	10	80	88.70	0.08
DC	$Pr(V \nvDash \varphi_{DC}) \geq 0.1 \pm 0.01$	$(\#c = 4)$	True	500	10	20	1.60	0.50
	$Pr(V \nvDash \varphi_{DC}) \geq 0.01 \pm 0.001$	$(\#c = 4)$	True	50	100	20	1.40	0.40
	$Pr(V \nvDash \varphi_{DC}) \geq 0.1 \pm 0.01$	$(\#c = 7)$	False	500	10	25	2.70	0.30
	$Pr(V \nvDash \varphi_{DC}) \geq 0.01 \pm 0.001$	$(\#c = 7)$	False	50	100	25	2.60	0.60
	$Pr(V \nvDash \varphi_{DC}) \geq 0.1 \pm 0.01$	$(\#c = 8)$	True	500	10	30	2.60	0.80
	$Pr(V \nvDash \varphi_{DC}) \geq 0.01 \pm 0.001$	$(\#c = 8)$	True	50	100	30	2.70	0.70
	$Pr(V \nvDash \varphi_{DC}) \geq 0.1 \pm 0.01$	$(\#c = 15)$	False	500	10	65	4.50	1.80
	$Pr(V \nvDash \varphi_{DC}) \geq 0.01 \pm 0.001$	$(\#c = 15)$	False	50	100	65	5.10	1.90
RNI	$Pr(V \vDash \varphi_{RNI}) \leq 0.1 \pm 0.01$	$(n = 3)$	Undec	200	10	10	385.00	0.30
	$Pr(V \vDash \varphi_{RNI}) \geq 0.2 \pm 0.01$	$(n = 3)$	True	400	5	10	3.80	0.20
	$Pr(V \vDash \varphi_{RNI}) \geq 0.5 \pm 0.01$	$(n = 3)$	True	1000	2	10	210.00	0.15
	$Pr(V \vDash \varphi_{RNI}) \leq 0.1 \pm 0.01$	$(n = 5)$	Undec	200	10	26	2999.00	0.19
	$Pr(V \vDash \varphi_{RNI}) \geq 0.2 \pm 0.01$	$(n = 5)$	True	400	5	26	38.17	0.33
	$Pr(V \vDash \varphi_{RNI}) \geq 0.5 \pm 0.01$	$(n = 5)$	Undec	1000	2	26	1243.00	0.67
	$Pr(V \vDash \varphi_{RNI}) \geq 0.2 \pm 0.01$	$(n = 10)$	True	400	5	80	173.65	0.87
	$Pr(V \vDash \varphi_{RNI}) \geq 0.5 \pm 0.01$	$(n = 10)$	Undec	1000	2	80	10.4k	1.38
CSO	$Pr(V \vDash \varphi_{CSO}) \leq 0.05 \pm 0.001$	$(n = 10)$	Undec	150	20	30	84.00	0.82
	$Pr(V \vDash \varphi_{CSO}) \geq 0.3 \pm 0.01$	$(n = 10)$	True	900	4	30	0.70	0.17
	$Pr(V \vDash \varphi_{CSO}) \leq 0.7 \pm 0.01$	$(n = 10)$	True	2100	2	30	0.93	0.25
	$Pr(V \vDash \varphi_{CSO}) \leq 0.05 \pm 0.001$	$(n = 20)$	Undec	150	20	45	376.00	0.41
	$Pr(V \vDash \varphi_{CSO}) \geq 0.3 \pm 0.01$	$(n = 20)$	True	900	4	45	2.41	0.34
	$Pr(V \vDash \varphi_{CSO}) \leq 0.7 \pm 0.01$	$(n = 20)$	True	2100	2	45	1.74	0.41
	$Pr(V \vDash \varphi_{CSO}) \leq 0.05 \pm 0.001$	$(n = 30)$	True	150	20	55	511.00	0.35
	$Pr(V \vDash \varphi_{CSO}) \geq 0.3 \pm 0.01$	$(n = 30)$	True	900	4	55	7.97	0.29
	$Pr(V \vDash \varphi_{CSO}) \leq 0.7 \pm 0.01$	$(n = 30)$	True	2100	2	55	2.45	0.32

Table 9.2: Data from experimentation. #sch: number of scheduler-tuples sampled, #tr: number of trace tuples sampled per scheduler tuple, k: length of traces sampled. $\alpha = \beta = 0.01$.

the higher end (between 0.5 - 1) is not always possible in the given budget and indifference regions.

During our experiments on the opacity case study (CSO), we added a sub-formula to

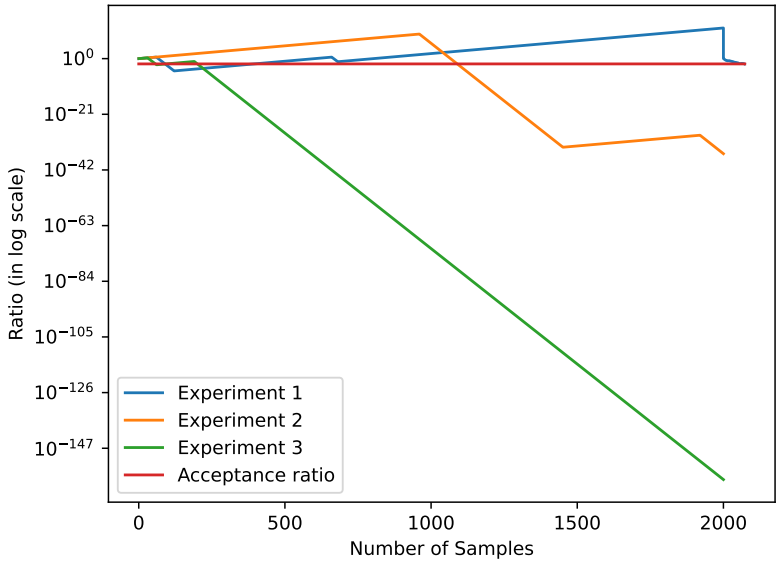
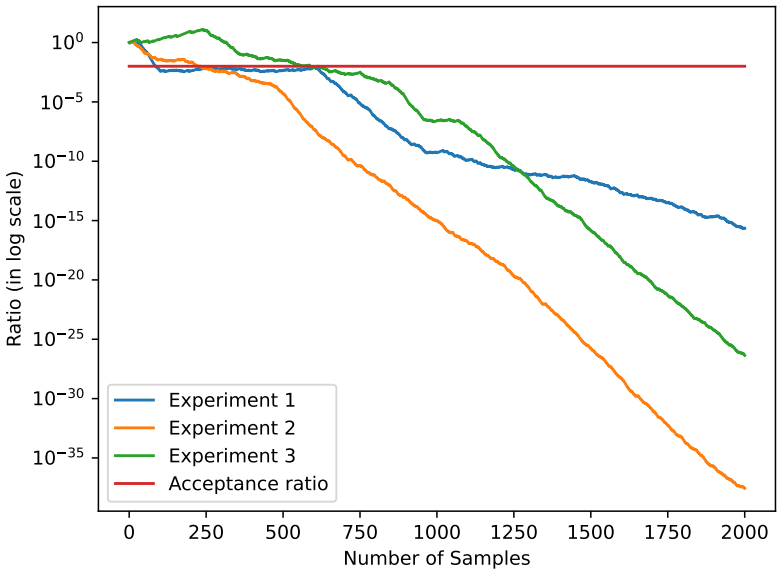


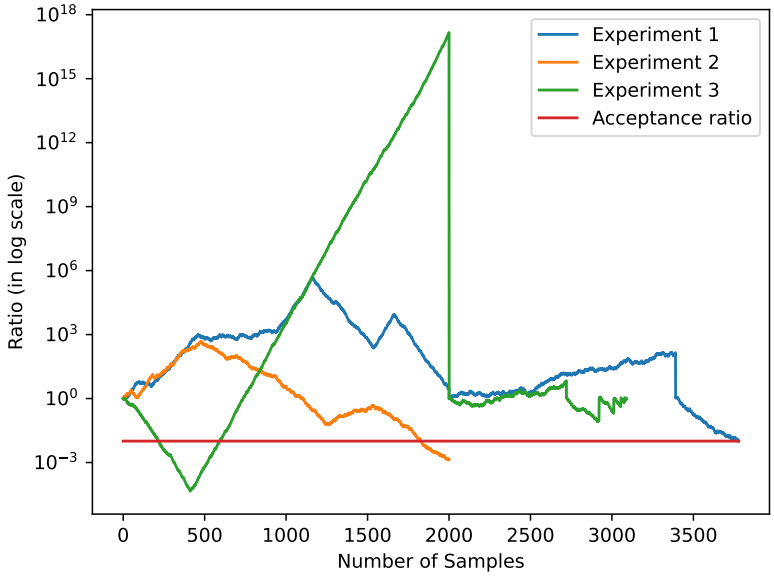
Figure 9.3: DC with $n = 4$ ($Pr \geq 0.1 \pm 0.01$).

φ_{CSO} to check if the robot reaches the goal in both traces. Given that we do not enforce any smart movement strategy on the movement of the robot, it usually makes a random walk in the grid often looping in a few states for a long time. Consequently, the probability of the robot reaching the goal is highly unlikely. We checked the probability of satisfying our specification against $\{0.05, 0.3, 0.7\}$. We used a budget of 3000 for all versions of this experiment and increased the trace length in proportion to the increase in the grid size.

The plots in Figs. 9.3, 9.4 depict convergence results, where each line in a graph shows how the value of *ratio* changes across a single algorithm run. In each of the plots, the red line represents the ratio A in Alg. 22 which serves as our exit condition. In Fig. 9.3, we use the sampling budget of 2000 to calculate the *ratio*. At the end of this phase, if the *ratio* is below A, we can declare that we have found enough evidence for a concrete result of the specification being satisfied as shown in lines labeled experiment 2 and 3. For the case of experiment 1, we could not reach a concrete conclusion in the initial round, as the line can be seen to be well above A. We were required to enter the main algorithm loop and required a few more samples (~ 75) to reach the same concrete conclusion.



(a) RNI with $n = 4 (Pr \geq 0.2 \pm 0.01)$.



(b) RNI with $n = 4$ $(Pr \geq 0.5 \pm 0.01)$.

Figure 9.4: Plots showing the change in ratio based on sampling across schedulers.

The robotics case plotted in Fig. 9.4a shows that we were able to get a concrete result in the initial sampling for all three cases. We plotted an undecidable case in Fig. 9.4b. Note that in experiment 2 we were able to get a concrete result in the initial sampling round;

in experiment 1, we were able to reach a concrete result in the main algorithm loop after intensive sampling within the chosen schedulers; and in experiment 3, we could neither find an accepting scheduler nor reject all schedulers, leading to an inconclusive result. This supports the results of undecidability that the algorithm returned. The main reason can be traced back to the fact that we did not specify any strategy for the robots, thus, sampling across random walks of the robot.

9.7 Summary

We presented a probabilistic formulation of bounded, unquantified HyperLTL and provided an SMC approach to verify them over MDP. To handle nondeterminism, our approach leverages the smart sampling algorithm presented in [DLST15], extending it to reason about hyperproperties. We have implemented our approach as an extension of PLASMA [LS14] adding new capabilities to perform black-box verification and demonstrating the scalability of our approach in several case studies with large state spaces. This work aimed to showcase that SMC is a feasible solution for cases where exhaustive or bounded model checking is unable to provide us with any insight. In future directions, we would like to extend support for quantifier alternations for paths (as in HyperLTL) and scheduler tuples, as the current approach can only handle existential scheduler tuples and limits our applicability to a wider variety of security properties.

Chapter 10

Related Work

In this chapter, we summarize the inexhaustive line of work that has influenced our work. We begin at the origins of the trace logic, their evolution to accommodate different types of requirements, and the model checking methods used for them. Thereafter, we discuss related works pertaining to the specific problems handled in each specific chapter in this thesis.

10.1 Discrete and Continuous Time Logics

Temporal logic essentially introduces an ordering among events or a 'sense' of time in the system without actually including time. In Linear Temporal Logic (LTL) [Pnu77] argues over all linear time finite traces. It is the extension of propositional logic with temporal operators. An orthogonal attempt was developed as Computation Tree Logic (CTL) [BAMP81] to argue about branching time logic. Later CTL* [EH83] was developed to subsume both LTL and CTL. LTL and CTL have become standards in the model checking community.

Initially, to verify properties on probabilistic, concurrent finite-state programs, LTL was used. The aim was to check if an LTL property holds in a Markov chain 'almost always'. The initial work on model checking of probabilistic programs and their optimizations was presented in [HSP83], [Var85], [CY88]. This motivated the development of Probabilistic Computation Tree Logic (PCTL) [HJ94]. The idea was to ensure soft bounds into properties instead of ensuring something always happens. For e.g., 'The system is up 98% of the time'

cannot be expressed in LTL or CTL. So, PCTL includes an explicit inclusion of probabilistic operators in the logic and can be used for both quantitative verification along with the qualitative answer provided by LTL or CTL. This was followed by [CY95], [DA98] where the complexity of probabilistic logics for both linear and branching time logics have been explored in terms of DTMCs. In [BdA95], the authors extend the logic to include the concept of satisfaction of probability operators to accommodate non-deterministic choices and describe model checking algorithms for such systems. The complexity of their algorithm is linear in the size of the state-space of the model.

In the context of continuous time systems, [ASSB96] defined the logic Continuous Stochastic Logic (CSL) to describe properties and proved that the verification problem is decidable. In [BKH99], the authors provide a model checking algorithm for CSL by reducing the problem to solving set of linear equations and Volterra integrals. They provide an approximate symbolic method for solving the integrals using MTDDs (multi-terminal decision diagrams).

Binary Decision Diagrams Taking a step back, there have been several works on using Binary Decision Diagrams and they brought in a major breakthrough in efficiently solving model checking problems. In [Bry86], Bryant came up with the idea of using acyclic graphs to represent Boolean functions. This brought a major breakthrough as a range of works utilized this procedure for model checking. The complexity of the binary operations on functions was proportional to the product of the graphs. The author describes the utility as "... the performance of a program based on our algorithms when processing a sequence of operations degrades slowly, if at all" [Bry86]. However, the major shortcoming of the seminal work was that the size of the graph depended on the ordering in which the input variables from the systems are used to build the graph. And finding the ordering is in itself a co-NP problem. BDDs brought in the idea of symbolic model checking. The main idea of symbolic model checking [BCM$^+$92] [McM93] involves conversion of the state space into their symbolic

representations like BDDs which, when given a property, can be traversed and labeled to check for inconsistencies. One of the main shortcomings of model checking has always been the state-explosion problem. BDD representation has been a huge progress in handling this problem. However, BDD was designed to handle Boolean representations. Multi-Terminal BDD (MTBDD) [CFZ96] was introduced to handled real values. In [BCHG+97], the authors used BDDs to represent properties, MTBDDs to represent Markov Chains, and utilized the algorithm in [HJ94]. In [KNP00] the authors published the first experience of working with the model checker PRISM [KNP01] that used MTBDDs. PRISM has been widely used across domains to exhibit the power of probabilistic model checking to verify large models with probabilistic properties.

10.2 Hyperproperties

In a series of work [FGM04], [McL96], [TA05], several important security properties were hinted to not fall under the general category of trace properties. In [CS08], the authors formalized the concept of *Hyperproperties*. The paper elaborated on the difference between trace and hyperproperties, possible application fields, and their topological characterization. This brought on a whole new field of research as the different logics available for trace properties had to be re-imagined as their hyper-components. In [CFK+14], the authors introduced HyperLTL and HyperCTL* to express linear and branching time hyperlogics, and in [FRS15] the authors introduced the model checking algorithm for the same.

There has been a lot of recent progress in verifying [FMSZ17, FHT18, CFST19, Fin21, PT18] and monitoring [AB16, FHST19, BSB17, BSS18, FHST18a, SSSB19, HST19] HyperLTL specifications. A growing set of tools support HyperLTL, including the model checker MCHyper [FRS15, CFST19], the satisfiability checkers EAHyper [FHS17], MGHyper [FHH18], explicit-state chceker for arbitrary quantifier called AutoHyper [BF23]. The model checking problem of HyperLTL is in general undecidable. Hence, work has been done on fragment specific approximate model checking solutions for the same. In [HSB21] the authors provided

a bounded model checking solution by reducing the problem to quantified Boolean formula and solving it for a solution. HyperQB [HBS24] is the QBF-encoding based model checker applying this theory. In [AMTZ21] the authors have explored augmented barrier certificates for verifying HyperLTL for cyber-physical systems. In [WNP20], the authors specifically use HyperLTL to express optimal robotics path planning requirements in HyperLTL. In the context of run-time verification, in [FHST17] the authors studied the monitoring problem for HyperLTL under different combination of quantifiers involved, in [AB16] the authors studied the verification of k-safety in HyperLTL, in [BSS18] the authors combined static analysis and runtime verification to monitor HyperLTL properties, in [CFH$^+$21] the authors studied the enforcement of universally quantified HyperLTL, and in [FHST18b] the authors revealed the development of the tool RVHyper for runtime-verification of HyperLTL hyperproperties. For real-valued signals, [NKJ$^+$17] introduced HyperSTL and [BDF$^+$22] introduced HyperSTL*.

Additionally, all of the above approaches reason about their logic in synchronous settings. Recently, there has been efforts to extend these languages to accommodate for asynchronous settings [BPS21], [BCB$^+$21], [GMOO21], [HBFS23b], [BBST24], [GMOO24] that adds more expressivity along with additional challenges in their model checking solutions. This can be interpreted as a way to handle nondeterminism in non-probabilistic systems.

10.3 Parameter Synthesis

A parametric DTMC [Daw04, LMST07] is a special class of Markov models, where some the transition probabilities (or rates) are not known a-priori and are parameter-dependent. These models can be adopted to analyze systems with stochastic uncertainty due to the impossibility to measure certain quantities (e.g., fault rates, packet loss ratios, etc.). In [Daw04], Daws proposed an approach to express the probability to reach the target state as a rational function with the domain in the parameter space. This symbolic approach has been exploited in several model checking algorithms for parametric probabilistic Markov chains [HHZ11, BGK$^+$11, JCV$^+$14, PÁJ$^+$15, DJJ$^+$15, QDJ$^+$16] and efficiently imple-

mented in PARAM1 and PARAM2 tools [HHZ11]. There are also *type-theoretic* approaches (e.g., [SA19]) for synthesizing protocols for differential privacy, which is out of scope of our target problem. The *parameter synthesis* problem consists in exploiting the generated rational function to find the parameter values that would maximize or minimize the probability to reach the target state [BGK+11]. The price to pay for these techniques is the increasing complexity of the rational functions in the presence of large models [LMST07], causing the parameter synthesis to be also very computationally expensive. However, the introduction of new efficient heuristics [JCV+14, PÁJ+15, DJJ+15, QDJ+16, CDP+17, ABC+18, SJK19] has helped to alleviate this problem by supporting the parameter synthesis for quite large models. For example, PROPhESY [DJJ+15] supports incremental automatic parameter synthesis for parametric Markov chains w.r.t. reachability properties expressed in PCTL and it exploits SMT techniques to determine *safe* and *unsafe* regions of the parameter space. In contrast with PROPhESY, our approach enables the parameter synthesis for the richer class of formal specifications defined by ReachHyperPCTL, a fragment of HyperPCTL. It is worth pointing out that other approaches based on *parameter lifting* [QDJ+16, CDP+17], although scalable, cannot be adapted to our framework. For example, the work in [QDJ+16] propose to: (a) relax the dependency among the parameters by introducing free variables, (b) replace parametric transitions by nondeterministic choices of extremal values, and (c) analyze the resulting parameter-free Markov decision process by computing lower and upper bounds on probabilities of regions in the parameter space. This approach is restricted to work only for probabilistic properties with an *eventually* operator, while our ReachHyperPCTL supports other operators such as the *until*. Furthermore, ReachHyperPCTL allows to compare the value of two probabilistic operators and this feature makes it infeasible to use parameter lifting that provides probability intervals. In [JJK22] the authors provide a detailed comparison of the state-of-the-art concepts and solutions in the field of parameter synthesis. In [JÁH+19] the authors focus on research questions related to coverage of the parameter space i.e., if all values in a region satisfy the property, finding regions and do or do not

satisfy the property, and finding approximate solution to the previous question to cover a larger fraction of possible values.

10.4 Model Checking of Probabilistic Hyperproperties

In this context, it is important to note that the work in [DFT20] independently addresses the problem of incorporating reasoning over nondeterminism similar to chapter 4.The authors propose the temporal logic PHL. Similar to HyperPCTL, PHL also allows quantification over schedulers, but path quantification of the induced DTMC is achieved by using HyperCTL*. Both the works show that the model checking problem is undecidable for the respective logics. The difference, however, is in our approaches to deal with the undecidability result, which leads to two complementary and orthogonal techniques. For both logics the problem is decidable for non-probabilistic memoryless schedulers. We provide an SMT-based verification procedure for HyperPCTL for this class of schedulers. The work in [DFT20] presents two methods for proving and for refuting formulas from a fragment of PHL for general memoryful schedulers. The authors propose an over-approximate and another under-approximate automata-based model checking algorithms for the alternation-free n-safety fragment of their logic PHL on n self-composed systems. The scheduler synthesis step is the main challenge in this work.The two papers offer disjoint case studies for evaluation. In [WNBP21], the authors provide a scalable solution to tackle probabilistic hyperproperties using statistical model checking focusing on HyperPCTL*. The idea is to use sequential probability ratio tests with multi-dimensional indifference regions. This allows verification of nested probability operators. However, this work does not allow non-determinism in the models. In asynchronous settings [GDÁ$^+$23] extends HyperPCTL to argue over novel stutter schedulers which allows scheduler synthesis in systems that allows stuttering in states. The general complexity of the model checking problem for probabilistic hyperproperties has motivated numerical solutions for DTMCs [ZCÁB22], and deductive pruning of strategies generated by abstraction refinement [ABČ$^+$23] for MDPs. Both these methods handle their respective fragments of

the logic efficiently.

There has been tremendous effort in language-based techniques for dealing with probabilistic information-flow, specifically differential privacy (e.g., [BEH+17, BGHP16, BKOB13, BGG+16]), but those techniques are not in the same context as the problem we deal with across this dissertation. An interesting direction was pursed in [KVAK10] where the authors viewed MDPs as *distribution transformers* which was capable of expressing properties involving comparison of values in a set of states with another set of states at the same time. This view is intuitively connected to hyperproperties. Recently, [ACMZ23, AGV18] have used this view of MDPs as distribution transformers and worked with distribution-based objectives for MDPs. For safety properties that can be specified as closed convex stochastic polytopes, the authors show that the existential safety problem for MDPs is PTIME-complete and that the universal safety problem is coNP-complete [AGV18]. This perspective has motivated our algorithm idea in chapter 8.

10.5 Statistical Model Checking

The hardness of the general model checking problem for quantitative properties has motivated exploration of approximate solutions that find a balance between scalability and accuracy within a specified confidence bound. The literature mentioned for hyperproperties above suffer from the challenges of scalability, inability to handle probabilistic systems, or lack of support for nondeterminism.

To verify hyperproperties in probabilistic systems there are two main families of approaches proposed in the literature: exact methods [ÁB18, ÁBBD20b, DÁBB22, DABB21] and approximated ones [WNBP21, DP22]. Note that the specification language used in these works differs from our specification language in chapter 9. Exact methods exploit the underlying Markov chain structure of the probabilistic system to be verified for computing precisely (numerically) and for comparing the probabilities of satisfying temporal logic formulas of multiple and independent sequences of sets of states.

SMC has been explored to solve problems across different domains for analysing dynamic software architectures [CQT+16], performing security risk assessments using attack-defence trees [GHL+16], verifying cyber-physical systems [CZ11], validation of biochemical reaction models [Zul15], etc. Verification of bounded LTL for MDP has been proposed using SMC [HMZ+12] and has shown promising results. Extensive tool support exists for SMC on trace properties with respect to discrete-event modelling [BCLS13], priced timed automata [BDL+12], probabilistic model checking [KNP11, KZH+11, You05], black-box systems [GD08]. Statistical verification of probabilistic hyperlogics has been proposed for HyperPCTL* [WNBP21, DP22], for continuous Markov chains [WZBP19], and for real-valued signals [AHL+22]. However, none of these works reason about models involving nondeterminism.

Chapter 11

Conclusion and Future Work

In the previous chapters, we have defined a generalized logic to express probabilistic hyperproperties on Markov models, discussed the complexity involved in their model checking problem, and explored approximate solutions to model check fragments of the logic that can still be used to express important information flow and security related properties.

11.1 Summary

In Chapter 3, the *parameter synthesis* problem takes as input a parametric discrete-time Markov chain and a probabilistic hyperproperty, and asks for valid parameter values for which the induced discrete-time Markov chain satisfies the probabilistic hyperproperty. Our synthesis algorithm works in two steps. In the first step, it computes symbolic conditions for satisfying the formula, involving rational functions on the set of parameters. In the second step, it identifies regions of satisfying parameter configurations by decomposing the domain of parameter configurations and exploring smaller regions in which either all or none of the configurations lead to the satisfaction of the input formula. We demonstrated how our algorithm works on several examples: randomized response, probabilistic conformance, probabilistic noninterference, and dining cryptographers.

In Chapter 4, we investigated the problem of specifying and model checking probabilistic hyperproperties of Markov decision processes (MDPs). Our study is motivated by the fact

that many systems have a probabilistic nature and are influenced by nondeterministic actions of their environment. We extended the temporal logic HyperPCTL for DTMCs [ÁB18] to the context of MDPs by allowing formulas to quantify over schedulers. This additional expressive power leads to the undecidability of the HyperPCTL model checking problem on MDPs, but we also showed that the undecidable fragment becomes decidable for non-probabilistic memoryless schedulers. Indeed, all applications discussed here only require this type of scheduler.

In Chapter 5, we focus on the decidable fragment of our logic and provide proof of decidability by reducing the SAT problem to our model checking problem. We propose an SMT-encoding based algorithm to model check the single-scheduler logic. The algorithm is sound and complete and can provide a counterexample or witness to the specified hyperproperty. We provide evaluation results on a range of case studies involving security and conformance.

In Chapter 6, we studied probabilistic hyperproperties with rewards. To this end, we extended the temporal hyperlogic HyperPCTL with reward operators that associate quantified computation trees with interrelated accumulated rewards. We also proposed an SMT-based algorithm for model checking these formulas for MDPRs. We have created a prototypical implementation and used it to analyse a few case studies.

In Chapter 7, we introduced HyperProb, a fully automated tool for model checking probabilistic hyperproperties expressed in the temporal logic HyperPCTL for DTMCs and as well as MDPs. We accumulated our algorithms from the previous chapters and combined them into a push-button automated model checker for the decidable fragment of HyperPCTL with rewards. We allow additional optimizations which scale better when compared to our previous implementations. We provide details of the inner workings of HyperProb along with details on how to use the tool.

In Chapter 8, we concentrated on the problem of model checking fragments of the temporal logic HyperPCTL for verification of probabilistic hyperproperties. We showed that these

fragments can specify interesting and useful probabilistic information-flow security policies through various case studies. These fragments essentially involve quantitative relational reachability reasoning to determine whether quantified policies and computations reach certain observations with equal probability. We proposed two algorithms to solve the model checking problem for different fragments and scheduler classes. We also demonstrated the effectiveness of our algorithms by conducting several case studies and showing orders of magnitude speed-up, compared to the state of the art.

In Chapter 9, we explored a statistical model checking (SMC) approach for universally quantified probabilistic hyperproperties for a fragment of HyperPLTL, allowing argument over nondeterminism. The algorithm uses a sequential probability ratio test to provide model checking results that scale well with a reduction in accuracy within a user-defined confidence bound. We extended the prominent SMC tool PLASMA to handle this fragment of hyperproperties. We showcased the effectiveness of our approach on case studies of huge state space size, well beyond the capacity of existing solutions.

11.2 Future Work

Based on the current line of research my future work can be considered in two main directions:

- Depth-wise, I want to explore further techniques to make more scalable tool support for probabilistic hyperproperties available to the community. We have explored distinct approaches to the model checking problem across different fragments and I intend to incorporate them in HyperProb to enable it to run multiple model checking solutions based on the type of specification. In the next section, I discuss possible future extensions or optimizations that can be incorporated into our current algorithms.

- Breadth-wise, I want to explore more complex applications where our logic might be helpful in specifying requirements for verification or synthesis.

11.2.1 Improvements in logic and model checking algorithms

HyperPCTL, with its extensions mentioned in this dissertation can be used to express complex system properties. In terms of logic, it still has gaps such as lack of support for structural constraints over schedulers and underlying DTMCs, and its inability to handle partial observability. Structural constraints can be used to filter out or limit the 'good candidates' for schedulers of specific during synthesis. Additionally, it lacks support for reasoning over partially observable systems. In reality, most systems are partially observable due to the lack of range of observability or restricted permissibility of the observer. There exists a long line of work in probabilistic model checking under partial observation [NPZ15], [AC20], [BPQR15], [WJW$^+$17], etc. When dealing with such systems, we require synthesis of observation-based schedulers that agree on choosing the same action in similarly observed states. Intuitively, this can be expressed as a hyperproperty where based on the comparison of observable outputs of two states, we restrict the choice of actions allowed in such paths. The prominent model checkers for quantitative properties PRISM [KNP11] and STORM [HJK$^+$22] come with support for partially observable models.

With reference to chap. 8, there are some specific open problems. First, the complexity of model checking for memoryful deterministic schedulers for all considered fragments remains open. It would also be interesting to further examine the relationship between $(1\sigma 1s)$ and distribution-based objectives, both from an algorithmic and a complexity perspective: Can we employ algorithmic approaches for distribution-based objectives to solve our queries, and can we adapt complexity results to our setting? We also plan to explore more complex fragments of HyperPCTL like ones that need the until operator along with the comparison of probabilities. Such formulas appear in reasoning about probabilistic causality with application in information-flow security.

From the algorithm perspective, when reasoning over multiple traces, we often utilize self-composition in cases of universally quantified specifications. This can be optimized by utilizing the symmetry occurring in the two models. On the other hand, one of the main

bottlenecks in the algorithm is the controller synthesis step. Our statistical-based approach (in chap. 9) was scalable essentially because we were able to handle the resolution of non-determinism efficiently. Abstraction refinement-based approach [ABČ+23] has shown great promise in this regard. Our current work in chap. 8 has utilized a guided exhaustive search. Although encouraging, there is ample scope for improvement in this aspect by exploiting structural intricacies in the model. We currently do not gather any further information when exploring a scheduler that is not a good candidate for our specification. We can reduce the number of explored actions here by eliminating the possibility of exploring redundant action choices based on feedback from an already explored action combination. For statistical model checking, the limitation currently is in its inability to handle the alternation of quantifiers. But this extension is not trivial. For example, for a $\forall$ `schedulers`. $\exists$ `scheduler` property, the result of the SMC algorithm might not be accurate since we are not in reality exploring all possibilities. However, we still might be able to use over-and-under approximation over the set of schedulers to provide a result within a specified confidence bound.

11.2.2 Applications

On another front, the latest fast-track progress in the field of machine learning has brought into focus the need for augmenting reasoning capabilities to large-scale machine learning models. Probabilistic programming [vdMPYW21], [BKS20] has surfaced as the general language represented in machine learning. Although the idea might not be new [SPH84], [LMOW08], [PZ93], there has been a renewed interest in expanding reasoning over probabilistic programming languages [Kat15], [DLHM18], [TT23], [SBK+23] by extending existing program verification concepts to accommodate for probabilistic languages. This poses a scope for probabilistic hyperproperties to expand on these works to express specifications beyond functional correctness in programs. Recently, the core program correctness concept of Hoare triples has been extended to Hyper-Hoare triples [DM23] to argue about security and privacy-related hyperproperties like fairness, robustness, noninterference, etc., in

program semantics. Equivalence checking, which in itself is a form of conformance property and hence, a hyperproperty, of programs among different probabilistic languages can help in defining methods for easy and fast translation of solutions. Note that these extensions might not be trivial as the models cannot be always represented as Markov models due to their complexity. Hence, verification of probabilistic programs would require verification using theorem proving instead of solely model checking.

Another important application is causality. It is the relationship between cause and effect. Due to the randomness involved in practical systems, it makes sense to interpret the relationship in terms of probability. Trace properties in terms of PCTL have been used [KM09] to find causes of events. However, using this process we can only argue about each possible cause individually. Introducing the concept of hyperproperties into causality would help us take this argument a step further to explore how relationships between causes affect the occurrences of events. This would have applications in early diagnosis of diseases using gene patterns, understanding financial markets, figuring out sustainable steps to slow down climate change, etc. An associated challenge when exploring causality is the size of the models used. Realistic models of the systems we aim to verify (human body or gene structure, financial markets, climate cycle, etc.) are huge and cannot be limited to small-size models as they will take away important aspects, causing a loss of the actual relationships we want to observe. Our current exhaustive solutions explore all possible cases in a model to prove its correctness. The general complexity of the problem hints at the need to find better approximate solutions, based on our current theory, to find a balance between the accuracy of our solutions and the time taken to find them.

BIBLIOGRAPHY

[AB16] Shreya Agrawal and Borzoo Bonakdarpour. Runtime verification of k-safety hyperproperties in HyperLTL. In *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*, pages 239–252, Lisbon, June 2016. IEEE.

[ÁB18] E. Ábrahám and B. Bonakdarpour. HyperPCTL: A temporal logic for probabilistic hyperproperties. In *Proceedings of the 15th International Conference on Quantitative Evaluation of Systems (QEST)*, pages 20–35, 2018.

[ÁBBD20a] E. Ábrahám, E. Bartocci, B. Bonakdarpour, and O. Dobe. Parameter synthesis for probabilistic hyperproperties. In *Proceedings of the 23rd International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, pages 12–31, 2020.

[ÁBBD20b] Erika Ábrahám, Ezio Bartocci, Borzoo Bonakdarpour, and Oyendrila Dobe. Probabilistic hyperproperties with nondeterminism. In Dang Van Hung and Oleg Sokolsky, editors, *Automated Technology for Verification and Analysis*, Lecture Notes in Computer Science, pages 518–534, Cham, 2020. Springer International Publishing.

[ABC+18] Sebastian Arming, Ezio Bartocci, Krishnendu Chatterjee, Joost-Pieter Katoen, and Ana Sokolova. Parameter-independent strategies for pmdps via pomdps. In *Proc. of the 15th Int. Conf. on Quantitative Evaluation of Systems (QEST'18)*, volume 11024 of *LNCS*, pages 53–70. Springer, 2018.

[ABČ+23] Roman Andriushchenko, Ezio Bartocci, Milan Češka, Francesco Pontiggia, and Sarah Sallinger. Deductive controller synthesis for probabilistic hyperproperties. In Nils Jansen and Mirco Tribastone, editors, *Quantitative Evaluation of Systems*, pages 288–306, Cham, 2023. Springer Nature Switzerland.

[AC20] Eric Atkinson and Michael Carbin. Programming and reasoning with partial observability. *Proc. ACM Program. Lang.*, 4(OOPSLA), nov 2020.

[ACMZ23] S. Akshay, Krishnendu Chatterjee, Tobias Meggendorfer, and Dorde Zikelic. MDPs as Distribution Transformers: Affine Invariant Synthesis for Safety Objectives. In Constantin Enea and Akash Lal, editors, *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part III*, volume 13966 of *Lecture Notes in Computer Science*, pages 86–112. Springer, 2023.

[ADDN17] Aws Albarghouthi, Loris D'Antoni, Samuel Drews, and Aditya V. Nori. Fairsquare: Probabilistic verification of program fairness. *Proc. ACM Program. Lang.*, 1(OOPSLA), oct 2017.

[AGV18] S. Akshay, Blaise Genest, and Nikhil Vyas. Distribution-based objectives for Markov Decision Processes. *CoRR*, abs/1804.09341, 2018.

[AHL+22] Shiraj Arora, René Rydhof Hansen, Kim Guldstrand Larsen, Axel Legay, and Danny Bøgsted Poulsen. Statistical model checking for probabilistic hyperproperties of real-valued signals. In Owolabi Legunsen and Grigore Rosu, editors, *Model Checking Software*, page 61–78, Cham, 2022. Springer International Publishing.

[AMTZ21] Mahathi Anand, Vishnu Murali, Ashutosh Trivedi, and Majid Zamani. Formal verification of hyperproperties for control systems. In *Proceedings of the Workshop on Computation-Aware Algorithmic Design for Cyber-Physical Systems*, CAADCPS '21, page 29–30, New York, NY, USA, 2021. Association for Computing Machinery.

[AP18] Gul Agha and Karl Palmskog. A survey of statistical model checking. *ACM Trans. Model. Comput. Simul.*, 28(1), jan 2018.

[ASSB96] Adnan Aziz, Kumud Sanwal, Vigyan Singhal, and Robert Brayton. Verifying continuous time markov chains. page 269–276, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[BAMP81] Mordechai Ben-Ari, Zohar Manna, and Amir Pnueli. The temporal logic of branching time. In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '81, page 164–176, New York, NY, USA, 1981. Association for Computing Machinery.

[BBG08] C. Baier, N. Bertrand, and M. Größer. On decision problems for probabilistic Büchi automata. In *Proc. of FOSSACS'08*, pages 287–301, 2008.

[BBGK12] Christel Baier, Tomás Brázdil, Marcus Größer, and Antonín Kucera. Stochastic game logic. *Acta Informatica*, 49(4):203–224, 2012.

[BBST24] Alberto Bombardelli, Laura Bozzelli, César Sánchez, and Stefano Tonetta. Unifying asynchronous logics for hyperproperties, 2024.

[BCB+21] Jan Baumeister, Norine Coenen, Borzoo Bonakdarpour, Bernd Finkbeiner, and César Sánchez. A temporal logic for asynchronous hyperproperties. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification*, pages 694–717, Cham, 2021. Springer International Publishing.

[BCBS21] J. Baumeister, N. Coenen, B. Bonakdarpour, and B. Finkbeinerand C. Sánchez. A temporal logic for asynchronous hyperproperties. In *Proceedings of the 33rd International Conference on Computer-Aided Verification (CAV)*, pages 694–717, 2021.

[BCHG+97] C. Baier, E. Clarke, V. Hartonas-Garmhausen, M. Kwiatkowska, and M. Ryan. Symbolic model checking for probabilistic processes. In P. Degano, R. Gorrieri, and A. Marchetti-Spaccamela, editors, *Proc. 24th International Colloquium on Automata, Languages and Programming (ICALP'97)*, volume 1256 of *LNCS*, page 430–440. Springer, 1997.

[BCLS13] Benoît Boyer, Kevin Corre, Axel Legay, and Sean Sedwards. Plasma-lab: A flexible, distributable statistical model checking library. In *Quantitative Evaluation of Systems: 10th International Conference, QEST 2013, Buenos Aires, Argentina, August 27-30, 2013. Proceedings 10*, pages 160–164. Springer, 2013.

[BCM+92] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 1020 states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[BdA95] Andrea Bianco and Luca de Alfaro. Model checking of probabilistic and nondeterministic systems. In P. S. Thiagarajan, editor, *Foundations of Software Technology and Theoretical Computer Science*, page 499–513, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[BDF+22] Sebastian Biewer, Rayna Dimitrova, Michael Fries, Maciej Gazda, Thomas Heinze, Holger Hermanns, and Mohammad Reza Mousavi. Conformance Relations and Hyperproperties for Doping Detection in Time and Space. *Logical Methods in Computer Science*, Volume 18, Issue 1, January 2022.

[BDL+12] Peter Bulychev, Alexandre David, Kim Gulstrand Larsen, Marius Mikučionis, Danny Bøgsted Poulsen, Axel Legay, and Zheng Wang. Uppaal-smc: Statistical model checking for priced timed automata. *arXiv preprint arXiv:1207.1272*, 2012.

[BEH+17] G. Barthe, T. Espitau, J. Hsu, T. Sato, and P.-Y. Strub. *-liftings for differential privacy. In *Proceedings of the 44th International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 102:1–102:12, 2017.

[BF18] Borzoo Bonakdarpour and Bernd Finkbeiner. The complexity of monitoring hyperproperties. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 162–174, 2018.

[BF23] Raven Beutner and Bernd Finkbeiner. Autohyper: Explicit-state model checking for hyperltl. In Sriram Sankaranarayanan and Natasha Sharygina, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 145–163, Cham, 2023. Springer Nature Switzerland.

[BGG+16] G. Barthe, M. Gaboardi, B. Grégoire, J. Hsu, and P.-Y. Strub. Proving differential privacy via probabilistic couplings. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 749–758, 2016.

[BGHP16] G. Barthe, M. Gaboardi, J. Hsu, and B. C. Pierce. Programming language techniques for differential privacy. *SIGLOG News*, 3(1):34–53, 2016.

[BGK+11] Ezio Bartocci, Radu Grosu, Panagiotis Katsaros, C. R. Ramakrishnan, and Scott A. Smolka. Model repair for probabilistic systems. In *Proc. TACAS 2011*, volume 6605 of *LNCS*, pages 326–340, 2011.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[BKH99] Christel Baier, Joost-Pieter Katoen, and Holger Hermanns. Approximative symbolic model checking of continuous-time markov chains. In Jos C. M. Baeten and Sjouke Mauw, editors, *CONCUR'99 Concurrency Theory*, page 146–161, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[BKOB13] G. Barthe, B. Köpf, F. Olmedo, and S. Z. Béguelin. Probabilistic relational reasoning for differential privacy. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 35(3):9:1–9:49, 2013.

[BKS20] Gilles Barthe, Joost-Pieter Katoen, and Alexandra Silva. *Foundations of probabilistic programming*. Cambridge University Press, 2020.

[BP09] Romain Beauxis and Catuscia Palamidessi. Probabilistic and nondeterministic aspects of anonymity. *Theoretical Computer Science*, 410(41):4006–4025, 2009.

[BPQR15] Simon Busard, Charles Pecheur, Hongyang Qu, and Franco Raimondi. Reasoning about memoryless strategies under partial observability and unconditional fairness constraints. *Information and Computation*, 242:128–156, 2015.

[BPS21] Laura Bozzelli, Adriano Peron, and César Sánchez. Asynchronous extensions of hyperltl. In *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–13, 2021.

[Bry86] Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[BSB17] N. Brett, U. Siddique, and B. Bonakdarpour. Rewriting-based runtime verification for alternation-free HyperLTL. In *Proc. of the 23rd Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'17)*, volume 10206 of *LNCS*, pages 77–93. Springer, 2017.

[BSS18] B. Bonakdarpour, C. Sánchez, and G. Schneider. Monitoring hyperproperties by combining static analysis and runtime verification. In *Proc. of the 8th Leveraging Applications of Formal Methods, Verification and Validation (ISoLA'18)*, volume 11245 of *LNCS*, pages 8–27. Springer, 2018.

[CARa] CARL. https://ths-rwth.github.io/carl/.

[CARb] CARL PARSER. https://github.com/ths-rwth/carl-parser/.

[CDP+17] M. Ceska, F. Dannenberg, N. Paoletti, M. Kwiatkowska, and L. Brim. Precise parameter synthesis for stochastic biochemical systems. *Acta Informatica*, 54(6):589–623, 2017.

[CFH+21] Norine Coenen, Bernd Finkbeiner, Christopher Hahn, Jana Hofmann, and Yannick Schillo. Runtime enforcement of hyperproperties. In Zhe Hou and Vijay Ganesh, editors, *Automated Technology for Verification and Analysis*, page 283–299, Cham, 2021. Springer International Publishing.

[CFK+14] Michael R. Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K. Micinski, Markus N. Rabe, and César Sánchez. Temporal logics for hyperproperties. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Martín Abadi, and Steve Kremer, editors, *Principles of Security and Trust*, volume 8414, pages 265–284. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[CFST19] N. Coenen, B. Finkbeiner, C. Sánchez, and L. Tentrup. Verifying hyperliveness. In *Proc. of CAV'19*, pages 121–139, 2019.

[CFZ96] Edmund M Clarke, Masahiro Fujita, and Xudong Zhao. Multi-terminal binary decision diagrams and hybrid decision diagrams. In *Representations of discrete functions*, pages 93–108. Springer, 1996.

[Cha88] David Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *Journal of Cryptology*, 1(1):65–75, Jan 1988.

[Chu67] Kai Lai Chung. Markov chains. *Springer-Verlag, New York*, 1967.

[CQT+16] Everton Cavalcante, Jean Quilbeuf, Louis-Marie Traonouez, Flavio Oquendo, Thais Batista, and Axel Legay. Statistical model checking of dynamic software architectures. In *Software Architecture: 10th European Conference, ECSA 2016, Copenhagen, Denmark, November 28–December 2, 2016, Proceedings 10*, pages 185–200. Springer, 2016.

[CS08] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. In *2008 21st IEEE Computer Security Foundations Symposium*, page 51–65, Jun 2008.

[CS10] M. R. Clarkson and F. B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010.

[CY88] C. Courcoubetis and M. Yannakakis. Verifying temporal properties of finite-state probabilistic programs. In *[Proceedings 1988] 29th Annual Symposium on Foundations of Computer Science*, pages 338–345, 1988.

[CY95] Costas Courcoubetis and Mihalis Yannakakis. The complexity of probabilistic verification. *J. ACM*, 42(4):857–907, jul 1995.

[CZ11] Edmund M. Clarke and Paolo Zuliani. Statistical model checking for cyber-physical systems. In Tevfik Bultan and Pao-Ann Hsiung, editors, *Automated Technology for Verification and Analysis*, page 1–12, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[DA98] Luca De Alfaro. Formal verification of probabilistic systems, 1998.

[DABB21] Oyendrila Dobe, Erika Abraham, Ezio Bartocci, and Borzoo Bonakdarpour. Hyperprob: a model checker for probabilistic hyperproperties. In *Formal Methods: 24th International Symposium, FM 2021, Virtual Event, November 20–26, 2021, Proceedings 24*, pages 657–666. Springer, 2021.

[DÁBB22] Oyendrila Dobe, Erika Ábrahám, Ezio Bartocci, and Borzoo Bonakdarpour. Model checking hyperproperties for markov decision processes. *Information and Computation*, 289:104978, 2022.

[Daw04] C. Daws. Symbolic and parametric model checking of discrete-time Markov chains. In *Proc. of the First Int. Conf. on Theoretical Aspects of Computing (ICTAC'04)*, volume 3407 of *LNCS*, pages 280–294. Springer, 2004.

[DFT20] Rayna Dimitrova, Bernd Finkbeiner, and Hazem Torfah. Probabilistic hyperproperties of markov decision processes. In *Proc. of ATVA 2020: the 18th International Symposium on Automated Technology for Verification and Analysis*, volume 12302 of *LNCS*, pages 484–500. Springer, 2020.

[Dij75] E. J Dijkstra. Guarded commands, nondeterminancy and formal derivations of programs. 1975.

[DJJ+15] Christian Dehnert, Sebastian Junges, Nils Jansen, Florian Corzilius, Matthias Volk, Harold Bruintjes, Joost-Pieter Katoen, and Erika Ábrahám. Prophesy: A probabilistic parameter synthesis tool. In *Proceedings of the 27th International Conference on Computer-Aided Verification (CAV)*, pages 214–231, 2015.

[DJKV17] C. Dehnert, S. Junges, J.-P. Katoen, and M. Volk. A Storm is coming: A modern probabilistic model checker. In *Proc. of the 29th Int. Conf. on Computer Aided Verification (CAV'17)*, volume 10427 of *LNCS*, pages 592–600. Springer, 2017.

[DLHM18] Saikat Dutta, Owolabi Legunsen, Zixin Huang, and Sasa Misailovic. Testing probabilistic programming systems. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, page 574–586, New York, NY, USA, 2018. Association for Computing Machinery.

[DLST15] Pedro D'argenio, Axel Legay, Sean Sedwards, and Louis-Marie Traonouez. Smart sampling for lightweight verification of markov decision processes. *Int. J. Softw. Tools Technol. Transf.*, 17(4):469–484, aug 2015.

[DM23] Thibault Dardinier and Peter Müller. Hyper hoare logic:(dis-) proving program hyperproperties (extended version). *arXiv preprint arXiv:2301.10037*, 2023.

[dMB08] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 337–340, 2008.

[DMNS16] C. Dwork, F. McSherry, K. Nissim, and A. D. Smith. Calibrating noise to sensitivity in private data analysis. *Journal of Privacy Confidentiality*, 7(3):17–51, 2016.

[DN11] Rocco De Nicola. *Process Algebras*, pages 1624–1636. Springer US, Boston, MA, 2011.

[doc] docker. Docker. https://www.docker.com/get-started.

[DP22] Spandan Das and Pavithra Prabhakar. Bayesian statistical model checking for multi-agent systems using hyperpctl*, 2022.

[DR14] Cynthia Dwork and Aaron Roth. The algorithmic foundations of differential privacy. *Found. Trends Theor. Comput. Sci.*, 9(3–4):211–407, aug 2014.

[DSB+23] Oyendrila Dobe, Stefan Schupp, Ezio Bartocci, Borzoo Bonakdarpour, Axel Legay, Miroslav Pajic, and Yu Wang. Lightweight verification of hyperproperties. In Étienne André and Jun Sun, editors, *Automated Technology for Verification and Analysis*, pages 3–25, Cham, 2023. Springer Nature Switzerland.

[DWÁ+22] Oyendrila Dobe, Lukas Wilke, Erika Ábrahám, Ezio Bartocci, and Borzoo Bonakdarpour. Probabilistic hyperproperties with rewards. In *NASA Formal Methods: 14th International Symposium, NFM 2022, Pasadena, CA, USA, May 24–27, 2022, Proceedings*, pages 656–673. Springer, 2022.

[EH83] E. Allen Emerson and Joseph Y. Halpern. "sometimes" and "not never" revisited: On branching versus linear time (preliminary report). In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '83, page 127–140, New York, NY, USA, 1983. Association for Computing Machinery.

[FBT13] Narges Fallahi, Borzoo Bonakdarpour, and Sébastien Tixeuil. Rigorous performance evaluation of self-stabilization using probabilistic model checking. In *Proc. of SRDS'13: the 32nd IEEE Int. Conf. on Reliable Distributed Systems*, pages 153–162. IEEE Computer Society, 2013.

[FGM04] Riccardo Focardi, Roberto Gorrieri, and Fabio Martinelli. Classification of security properties (part ii: Network security). 2004.

[FHH18] B. Finkbeiner, C. Hahn, and T. Hans. MGHyper: Checking satisfiability of HyperLTL formulas beyond the $\exists^*\forall^*$ fragment. In *Proc. of the 16th Int. Symp. on Automated Technology for Verification and Analysis (ATVA'18)*, volume 11138 of *LNCS*, pages 521–527. Springer, 2018.

[FHS17] B. Finkbeiner, C. Hahn, and M. Stenger. EAHyper: Satisfiability, implication, and equivalence checking of hyperproperties. In *Proc. of the 29th Int. Conf. on Computer Aided Verification (CAV'17)*, volume 10427 of *LNCS*, pages 564–570. Springer, 2017.

[FHST17] Bernd Finkbeiner, Christopher Hahn, Marvin Stenger, and Leander Tentrup. Monitoring hyperproperties. In Shuvendu Lahiri and Giles Reger, editors, *Runtime Verification*, page 190–207, Cham, 2017. Springer International Publishing.

[FHST18a] B. Finkbeiner, C. Hahn, M. Stenger, and L. Tentrup. RVHyper: A runtime verification tool for temporal hyperproperties. In *Proc. of the 24th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'18)*, volume 10806 of *LNCS*, pages 194–200. Springer, 2018.

[FHST18b] Bernd Finkbeiner, Christopher Hahn, Marvin Stenger, and Leander Tentrup. Rvhyper: A runtime verification tool for temporal hyperproperties. In Dirk Beyer and Marieke Huisman, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, page 194–200, Cham, 2018. Springer International Publishing.

[FHST19] B. Finkbeiner, C. Hahn, M. Stenger, and L. Tentrup. Monitoring hyperproperties. *Formal Methods in System Design (FMSD)*, 54(3):336–363, 2019.

[FHT18] B. Finkbeiner, C. Hahn, and H. Torfah. Model checking quantitative hyperproperties. In *Proc. of the 30th Int. Conf. on Computer Aided Verification (CAV'18)*, volume 10981 of *LNCS*, pages 144–163. Springer, 2018.

[Fin21] Bernd Finkbeiner. Model checking algorithms for hyperproperties (invited paper). In Fritz Henglein, Sharon Shoham, and Yakir Vizel, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 3–16, Cham, 2021. Springer International Publishing.

[FMSZ17] B. Finkbeiner, Ch. Müller, H. Seidl, and E. Zalinescu. Verifying Security Policies in Multi-agent Workflows with Loops. In *Proc. of CCS'17*, 2017.

[FRS15] Bernd Finkbeiner, Markus N. Rabe, and César Sánchez. Algorithms for model checking HyperLTL and HyperCTL*. In Daniel Kroening and Corina S. Păsăreanu, editors, *Computer Aided Verification*, Lecture Notes in Computer Science, pages 30–48. Springer International Publishing, 2015.

[GD08] David R. Gilbert and Robin Donaldson. A monte carlo model checker for probabilistic ltl with numerical constraints. 2008.

[GDÁ+23] Lina Gerlach, Oyendrila Dobe, Erika Ábrahám, Ezio Bartocci, and Borzoo Bonakdarpour. Introducing asynchronicity to probabilistic hyperproperties. In Nils Jansen and Mirco Tribastone, editors, *Quantitative Evaluation of Systems*, pages 47–64, Cham, 2023. Springer Nature Switzerland.

[GHL+16] Olga Gadyatskaya, René Rydhof Hansen, Kim Guldstrand Larsen, Axel Legay, Mads Chr Olesen, and Danny Bøgsted Poulsen. Modelling attack-defense trees using timed automata. In *Formal Modeling and Analysis of Timed Systems: 14th International Conference, FORMATS 2016, Quebec, QC, Canada, August 24-26, 2016, Proceedings 14*, pages 35–50. Springer, 2016.

[GM82] J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symp. on Security and Privacy*, pages 11–20, 1982.

[GMB17] M. Guarnieri, S. Marinovic, and D. Basin. Securing databases from probabilistic inference. In *Proc. of CSF'17*, pages 343–359, 2017.

[GMOO21] Jens Oliver Gutsfeld, Markus Müller-Olm, and Christoph Ohrem. Automata and fixpoints for asynchronous hyperproperties. *Proc. ACM Program. Lang.*, 5(POPL), jan 2021.

[GMOO24] Jens Oliver Gutsfeld, Markus Müller-Olm, and Christoph Ohrem. Deciding asynchronous hyperproperties for recursive programs. *Proc. ACM Program. Lang.*, 8(POPL), jan 2024.

[HBFS23a] Tzu-Han Hsu, Borzoo Bonakdarpour, Bernd Finkbeiner, and César Sánchez. Bounded model checking for asynchronous hyperproperties. *CoRR*, abs/2301.07208, 2023.

[HBFS23b] Tzu-Han Hsu, Borzoo Bonakdarpour, Bernd Finkbeiner, and César Sánchez. Bounded model checking for asynchronous hyperproperties. In Sriram Sankaranarayanan and Natasha Sharygina, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 29–46, Cham, 2023. Springer Nature Switzerland.

[HBS24] Tzu-Han Hsu, Borzoo Bonakdarpour, and César Sánchez. Hyperqb: A qbf-based bounded model checker for hyperproperties, 2024.

[Her90] Ted Herman. Probabilistic self-stabilization. *Information Processing Letters*, 35(2):63–67, 1990.

[HHZ11] Ernst Moritz Hahn, Tingting Han, and Lijun Zhang. Probabilistic reachability for parametric Markov models. *STTT*, 13(1):3–19, 2011.

[Hit21] Christopher Hitchcock. Probabilistic Causation. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Spring 2021 edition, 2021.

[HJ94] Hans Hansson and Bengt Jonsson. A logic for reasoning about time and reliability. *Formal Aspects of Computing*, 6(5):512–535, Sep 1994.

[HJK+22] Christian Hensel, Sebastian Junges, Joost-Pieter Katoen, Tim Quatmann, and Matthias Volk. The probabilistic model checker storm. *Int. J. Softw. Tools Technol. Transf.*, 24(4):589–610, aug 2022.

[HMZ+12] David Henriques, João G. Martins, Paolo Zuliani, André Platzer, and Edmund M. Clarke. Statistical model checking for markov decision processes. In *2012 Ninth International Conference on Quantitative Evaluation of Systems*, pages 84–93, 2012.

[HSB21] Tzu-Han Hsu, César Sánchez, and Borzoo Bonakdarpour. Bounded model checking for hyperproperties. In Jan Friso Groote and Kim Guldstrand Larsen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, page 94–112, Cham, 2021. Springer International Publishing.

[HSP83] Sergiu Hart, Micha Sharir, and Amir Pnueli. Termination of probabilistic concurrent program. *ACM Trans. Program. Lang. Syst.*, 5(3):356–380, jul 1983.

[HST19] C. Hahn, M. Stenger, and L. Tentrup. Constraint-based monitoring of hyperproperties. In *Proc. of the 25th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'19)*, volume 11428 of *LNCS*, pages 115–131. Springer, 2019.

[III92] J. W Gray III. Toward a mathematical foundation for information flow security. *Journal of Computer Security*, 1(3-4):255–294, May 1992.

[IJ90] A. Israeli and M. Jalfon. Token management schemes and random walks yield self-stabilizing mutual exclusion. In Cynthia Dwork, editor, *Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 119–131, 1990.

[JÁH+19] Sebastian Junges, Erika Ábrahám, Christian Hensel, Nils Jansen, Joost-Pieter Katoen, Tim Quatmann, and Matthias Volk. Parameter synthesis for markov models. *CoRR*, abs/1903.07993, 2019.

[JCV+14] Nils Jansen, Florian Corzilius, Matthias Volk, Ralf Wimmer, Erika Ábrahám, Joost-Pieter Katoen, and Bernd Becker. Accelerating parametric probabilistic verification. In *Proc. QEST 2014*, volume 8657 of *LNCS*, pages 404–420, 2014.

[JJK22] Nils Jansen, Sebastian Junges, and Joost-Pieter Katoen. *Parameter Synthesis in Markov Models: A Gentle Survey*, pages 407–437. Springer Nature Switzerland, Cham, 2022.

[Kat15] Joost-Pieter Katoen. Probabilistic programming: A true verification challenge. In Bernd Finkbeiner, Geguang Pu, and Lijun Zhang, editors, *Automated Technology for Verification and Analysis*, pages 1–3, Cham, 2015. Springer International Publishing.

[KM09] Samantha Kleinberg and Bud Mishra. The Temporal Logic of Causal Structures. In *Proceedings of the 25th Conference on Uncertainty in Artificial Intelligence (UAI)*, Montreal, Quebec, June 2009.

[KNP00] M. Kwiatkowska, G. Norman, and D. Parker. Verifying randomized distributed algorithms with prism. In *Proc. Workshop on Advances in Verification (Wave'2000)*, Jul 2000.

[KNP01] M. Kwiatkowska, G. Norman, and D. Parker. Prism: Probabilistic symbolic model checker. In P. Kemper, editor, *Proc. Tools Session of Aachen 2001 International Multiconference on Measurement, Modelling and Evaluation of Computer-Communication Systems*, page 7–12, Sep 2001.

[KNP11] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In G. Gopalakrishnan and S. Qadeer, editors, *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011.

[Kri63] S. Kripke. *Semantical Consideration on Modal Logic*, page 83–94. 1963.

[KVAK10] Vijay Anand Korthikanti, Mahesh Viswanathan, Gul Agha, and YoungMin Kwon. Reasoning about mdps as transformers of probability distributions. In *2010 Seventh International Conference on the Quantitative Evaluation of Systems*, pages 199–208, 2010.

[KY76] D. Knuth and A. Yao. *Algorithms and Complexity: New Directions and Recent Results*, chapter The complexity of nonuniform random number generation. Academic Press, 1976.

[KZH+11] Joost-Pieter Katoen, Ivan S Zapreev, Ernst Moritz Hahn, Holger Hermanns, and David N Jansen. The ins and outs of the probabilistic model checker mrmc. *Performance evaluation*, 68(2):90–104, 2011.

[lar] lark. LARK. https://lark-parser.readthedocs.io/.

[LDB10] Axel Legay, Benoît Delahaye, and Saddek Bensalem. Statistical model checking: An overview. In Howard Barringer, Ylies Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon Pace, Grigore Roşu, Oleg Sokolsky, and Nikolai Tillmann, editors, *Runtime Verification*, page 122–135, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[LL20] Kim G. Larsen and Axel Legay. 30 years of statistical model checking. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part I*, volume 12476 of *Lecture Notes in Computer Science*, pages 325–330. Springer, 2020.

[LLT+19] Axel Legay, Anna Lukina, Louis Marie Traonouez, Junxing Yang, Scott A Smolka, and Radu Grosu. Statistical model checking. In *Computing and software science: state of the art and perspectives*, pages 478–504. Springer, 2019.

[LMOW08] Axel Legay, Andrzej S Murawski, Joël Ouaknine, and James Worrell. On automated verification of probabilistic programs. In *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings 14*, pages 173–187. Springer, 2008.

[LMST07] R. Lanotte, A. Maggiolo-Schettini, and A. Troina. Parametric probabilistic transition systems for system design and analysis. *Formal Aspects of Computing*, 19(1):93–109, 2007.

[LS14] Axel Legay and Sean Sedwards. On statistical model checking with plasma. In *The 8th International Symposium on Theoretical Aspects of Software Engineering*, 2014.

[McL96] John McLean. A general theory of composition for a class of" possibilistic" properties. *IEEE Transactions on Software Engineering*, 22(1):53–67, 1996.

[McM93] Kenneth L McMillan. Symbolic model checking. In *Symbolic Model Checking*, pages 25–60. Springer, 1993.

[NKJ+17] 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 '17, page 104–113, New York, NY, USA, 2017. Association for Computing Machinery.

[NPZ15] Gethin Norman, D. Parker, and Xueyi Zou. Verification and control of partially observable probabilistic real-time systems. In *International Conference on Formal Modeling and Analysis of Timed Systems*, 2015.

[NSH13] T. M. Ngo, M. Stoelinga, and M. Huisman. Confidentiality for probabilistic multi-threaded programs and its verification. In *Proc. of ESSoS'13*, pages 107–122, 2013.

[PÁJ+15] S. Pathak, E. Ábrahám, N. Jansen, A. Tacchella, and J.-P. Katoen. A greedy approach for the efficient repair of stochastic models. In *Proc. of the 7th Int. NASA Formal Methods Symp. (NFM'15)*, volume 9058 of *LNCS*, pages 295–309. Springer, 2015.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 46–57, 1977.

[PT18] Adrien Pommellet and Tayssir Touili. Model-checking hyperltl for pushdown systems. In María del Mar Gallardo and Pedro Merino, editors, *Model Checking Software*, pages 133–152, Cham, 2018. Springer International Publishing.

[Put90] Martin L Puterman. Markov decision processes. *Handbooks in operations research and management science*, 2:331–434, 1990.

[pyc] pycarl. pycarl. https://moves-rwth.github.io/pycarl/.

[PZ93] Amir Pnueli and Lenore D Zuck. Probabilistic verification. *Information and computation*, 103(1):1–29, 1993.

[QDJ+16] T. Quatmann, C. Dehnert, N. Jansen, S. Junges, and J.-P. Katoen. Parameter synthesis for Markov models: Faster than ever. In *Proc. of the 14th Int. Symp. on Automated Technology for Verification and Analysis (ATVA'16)*, volume 9938 of *LNCS*, pages 50–67. Springer, 2016.

[Qua23] Tim Quatmann. *Verification of Multi-Objective Markov Models*. PhD thesis, RWTH Aachen University, 2023.

[RD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[Rei85] W. Reisig. *Petri Nets: An Introduction*, volume 4. Springer, 1985.

[SA19] C. Smith and A. Albarghouthi. Synthesizing differentially private programs. *Proc. of the ACM on Programming Languages (PACMPL'19)*, 3(ICFP):94:1–94:29, 2019.

[SBK+23] Philipp Schröer, Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. A deductive verification infrastructure for probabilistic programs. *Proc. ACM Program. Lang.*, 7(OOPSLA2), oct 2023.

[SJK19] J. Spel, S. Junges, and J.-P. Katoen. Are parametric Markov chains monotonic? In *Proc. of the 17th Int. Symp. on Automated Technology for Verification and Analysis (ATVA'19)*, volume 11781 of *LNCS*, pages 479–496. Springer, 2019.

[Smi03] Geoffrey Smith. Probabilistic noninterference through weak probabilistic bisimulation. In *Proceedings of the 16th IEEE Computer Security Foundations Workshop (CSF)*, pages 3–13, 2003.

[SPH84] Micha Sharir, Amir Pnueli, and Sergiu Hart. Verification of probabilistic programs. *SIAM Journal on Computing*, 13(2):292–314, 1984.

[SSSB19] S. Stucki, C. Sánchez, G. Schneider, and B. Bonakdarpour. Gray-box monitoring of hyperproperties. In *Proc. of the 3rd World Congress on Formal Methods (FM'19)*, volume 11800 of *LNCS*, pages 406–424. Springer, 2019.

[stoa] STORM: A tool for the analysis of systems involving random or probabilistic phenomena. http://www.stormchecker.org/index.html.

[stob] stormpy. STORMPY. https://moves-rwth.github.io/stormpy/.

[TA05] Tachio Terauchi and Alex Aiken. Secure information flow as a safety problem. In *International Static Analysis Symposium*, pages 352–367. Springer, 2005.

[TT23] Joseph Tassarotti and Jean-Baptiste Tristan. Verified density compilation for a probabilistic programming language. *Proc. ACM Program. Lang.*, 7(PLDI), jun 2023.

[Uck12] S. L. Uckelman. Arthur prior and medieval logic. 188(3):349–366, 2012.

[Var85] Moshe Y. Vardi. Automatic verification of probabilistic concurrent finite state programs. *26th Annual Symposium on Foundations of Computer Science (sfcs 1985)*, pages 327–338, 1985.

[vdMPYW21] Jan-Willem van de Meent, Brooks Paige, Hongseok Yang, and Frank Wood. An introduction to probabilistic programming, 2021.

[Wal45] A. Wald. Sequential Tests of Statistical Hypotheses. *The Annals of Mathematical Statistics*, 16(2):117 – 186, 1945.

[Way20] Hillel Wayne. Hypermodeling hyperproperties, Jan 2020.

[WJW+17] Leonore Winterer, Sebastian Junges, Ralf Wimmer, Nils Jansen, Ufuk Topcu, Joost-Pieter Katoen, and Bernd Becker. Motion planning under partial observability using game-based abstraction. In *2017 IEEE 56th Annual Conference on Decision and Control (CDC)*, pages 2201–2208, 2017.

[WNBP21] Yu Wang, Siddhartha Nalluri, Borzoo Bonakdarpour, and Miroslav Pajic. Statistical model checking for hyperproperties. In *IEEE Computer Security Foundations Symposium*, pages 1–16, Dubrovnik, Croatia, 2021.

[WNP20] Yu Wang, Siddhartha Nalluri, and Miroslav Pajic. Hyperproperties for robotics: Planning via hyperltl. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 8462–8468, 2020.

[WZBP19] Yu Wang, Mojtaba Zarei, Borzoo Bonakdarpour, and Miroslav Pajic. Statistical verification of hyperproperties for cyber-physical systems. *ACM Transactions in Embedded Computing Systems*, 18(5):92–, 2019.

[WZBP20] Y. Wang, M. Zarei, B. Bonakdarpour, and Miroslav Pajic. Probabilistic conformance for cyber-physical systems. *CoRR*, abs/2008.01135, 2020.

[YMCS16] Man-Ki Yoon, Sibin Mohan, Chien-Ying Chen, and Lui Sha. Taskshuffler: A schedule randomization protocol for obfuscation against timing inference attacks in real-time systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–12, 2016.

[You05] Håkan LS Younes. Ymer: A statistical model checker. In *Computer Aided Verification: 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005. Proceedings 17*, pages 429–433. Springer, 2005.

[ZCÁB22] Eshita Zaman, Gianfranco Ciardo, Erika Ábrahám, and Borzoo Bonakdarpour. Hyperpctl model checking by probabilistic decomposition. In Maurice H. ter Beek and Rosemary Monahan, editors, *Integrated Formal Methods*, pages 209–226, Cham, 2022. Springer International Publishing.

[ZM03] 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*, page 29. IEEE Computer Society, 2003.

[Zul15] Paolo Zuliani. Statistical model checking for biological applications. *International Journal on Software Tools for Technology Transfer*, 17:527–536, 2015.