UNIVERSITÄT
DES
SAARLANDES

Model Counting for Reactive Systems

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

by

Hazem Torfah

Saarbrücken, 2019

Dean of the faculty	Prof. Dr. Sebastian Hack
Date of the colloquium	Dec. 5th, 2019
Chair of the committee	Prof. Dr. Sebastian Hack
Reviewers	Prof. Bernd Finkbeiner, Ph.D.
	Prof. Ruzica Piskac, Ph.D.
	Prof. Sanjit Seshia, Ph.D.
Academic Assistant	Dr. Roland Leißa

Abstract

Model counting is the problem of computing the number of solutions for a logical formula. In the last few years, it has been primarily studied for propositional logic and has been shown to be useful in many applications. In planning, for example, propositional model counting has been used to compute the robustness of a plan in an incomplete domain. In information-flow control, model counting has been applied to measure the amount of information leaked by a security-critical system.

In this thesis, we introduce the model counting problem for linear-time properties and show its applications in formal verification. In the same way propositional model counting generalizes the satisfiability problem for propositional logic, counting models for linear-time properties generalizes the emptiness problem for languages over infinite words to a problem that asks for the number of words in a language. The model counting problem, thus, provides a foundation for quantitative extensions of model checking, where not only the existence of computations that violate the specification is determined, but also the number of such violations.

We solve the model counting problem for the prominent class of omega-regular properties. We present algorithms for solving the problem for different classes of properties and show the advantages of our algorithms in comparison to indirect approaches based on encodings into propositional logic. We further show how model counting can be used for solving a variety of quantitative problems in formal verification, including probabilistic model checking, quantitative information-flow in security-critical systems, and the synthesis of approximate implementations for reactive systems.

Zusammenfassung

Das Modellzählproblem fragt nach der Anzahl der Lösungen einer logischen Formel, und wurde in den letzten Jahren hauptsächlich für Aussagenlogik untersucht. Das Zählen von Modellen aussagenlogischer Formeln hat sich in vielen Anwendungen als nützlich erwiesen. Im Bereich der künstlichen Intelligenz wurde das Zählen von Modellen beispielsweise verwendet, um die Robustheit eines Plans in einem unvollständigen Weltmodell zu bewerten. Das Zählen von Modellen kann auch verwendet werden, um in sicherheitskritischen Systemen die Menge an enthüllten vertraulichen Daten zu messen.

Diese Dissertation stellt das Modellzählproblem für Linearzeiteigenschaften vor, und untersucht dessen Rolle in der Welt der formalen Verifikation. Das Zählen von Modellen für Linearzeiteigenschaften führt zu neuen quantitativen Erweiterungen klassischer Verifikationsprobleme, bei denen nicht nur die Existenz eines Fehlers in einem System zu überprüfen ist, sondern auch die Anzahl solcher Fehler.

Wir präsentieren Algorithmen zur Lösung des Modellzählproblems für verschiedene Klassen von Linearzeiteigenschaften und zeigen die Vorteile unserer Algorithmen im Vergleich zu indirekten Ansätzen, die auf Kodierungen der untersuchten Probleme in Aussagenlogik basieren. Darüberhinaus zeigen wir wie das Zählen von Modellen zur Lösung einer Vielzahl quantitativer Probleme in der formalen Verifikation verwendet werden kann. Dies beinhaltet unter anderem die Analyse probabilistischer Modelle, die Kontrolle quantitativen Informationsflusses in sicherheitskritischen Systemen, und die Synthese von approximativen Implementierungen für reaktive Systeme.

Acknowledgments

I want to express my immense gratitude to my advisor, Bernd Finkbeiner, for adopting me to his group and for his continuous support throughout my years as his doctoral student. Thank you, Bernd, for believing in me, for your patience, and your commitment. Thank you for appreciating my crazy ideas and for encouraging me to pursue them. Thank you for sharing your knowledge with me, for celebrating my achievements with me, and riding along in adventurous taxi drives through Madrid. Thank you for setting the way. I am excited to continue my journey.

I would like to thank my friends Andi, Andre, Caro, Connie, Florian, Elisa, Jule, Julia, Marco, Nathalie, Nora, Patrick, Pia, Pia, Sina, Stefan, Stephan, Sven, and Verena for wonderful times in Saarbrücken, for the Wohnheim-E evenings, for the crazy parties at Heuduckstraße, for the late-night snow hikes in Sweden, and for reconquering my true home Hazemburg. You are awesome. Julia, Andi, Pia, Pia, Marco, Nora, and Andre thank you for being there for me in the last months.

I am grateful to Manuel, Angela, Elias, and Lukas for making me part of their family. I am glad to have you in my life.

I want to thank Mai for being there for me over all the years. Thank you for understanding me, for believing in me, and for being my friend.

I want to thank Norine for her unconditional support, as a friend and colleague. Thank you for sharing your tea with me and for reminding me to breathe.

I want to thank my colleagues Jan Baumeister, Tom Baumeister, Rüdiger Ehlers, Michael Gerke, Sebastian Gerling, Jesko Hecking-Harbusch, Jana Hofmann, Swen Jacobs, Felix Klein, Florian Kohn, Andrey Kupriyanov, Sabine Lessel, Niklas Metzger, Noemi Passing, Hans-Jörg Peter, Markus Rabe, Mouhammad Sakr, Christa Schäfer, Malte Schledjewski, Maximilian Schwenger, Alexander Weinert, and Martin Zimmermann for productive collaborations, interesting discussions, crossword puzzles, delicious cakes, tennis matches, and the wonderful memories at Saarland University. Special thanks go to Rayna Dimitrova, Anna Marie and Peter Faymonville, Christopher Hahn, and Leander Tentrup for their continuous support during my time at the reactive systems group.

I want to thank my doctoral committee for their time and feedback: the external reviewers, Ruzica Piskac and Sanjit A. Seshia, the committee chair Sebastian Hack, and the academic staff committee member Roland Leißa.

I want to thank the Deutsche Telekom Foundation for supporting my doctoral studies. Special thanks go to Christiane Frense-Heck, Pascal Cerfontaine, Martin Heining, Sarah Meisenheimer, and Helen Nöding.

I want to thank my Aunt Nahla and my Uncle Haitham for opening their home for me, my Uncle Maher and Carmen for all their support and care, and Brigitte and Kurt for their love, you are like parents to me.

Above all, I want to thank my parents for their love, devotion, and sacrifice. You made me the person I am today. My sister Lisa, and my brother Rani, thank you for having my back. I dedicate this thesis to you. I love you all.

Contents

Chapter 1

Introduction

Reactive systems are computer systems that continuously interact with their surrounding environment [81]. They constitute the majority of modern IT-systems and play a crucial role in many applications in transport, telecommunications, medicine, and several other safety- and security-critical areas. The development of reactive systems can be very challenging. Constructing correct implementations for such systems requires developers to anticipate every possible behavior of the environment, which is in many cases not possible. This has increased the demand for advanced automated tools that support developers in the construction of safe and secure reactive systems and has made formal verification indispensable for computer-aided analysis tools.

A major breakthrough in the formal verification of reactive systems was made with the introduction of *model checking* [43, 128]. Model checking is a powerful push-button technique that automatically checks the correctness of a system with respect to a formal specification. A formal specification is a formula, given in some logical formalism, that defines the correct behavior of a system by posing constraints on its interaction with the surrounding environment [91]. The most common logical formalism for the specification of reactive systems are based on the linear-time paradigm [44, 121, 126, 143], where the behavior of the system is captured by its set of *traces*, i.e., the infinite alternating sequences of input and output values representing the values received from the environment and those produced in reaction by the system. A *linear-time property* [13] defines

the correct behavior of the system by defining the set of traces that are allowed to be produced by this system. In this setting, model checking is then defined as the problem of verifying that all traces of the system lie within the set of traces defined by linear-time property. Solving the model checking problem thus boils down to solving the underlying emptiness problem for languages over infinite words, where we check whether the intersection of the language defined by the set of traces of the system and the complement of the language defined by the linear-time property, is empty.

In this thesis, we generalize the emptiness problem to a counting problem for languages over infinite words, identified as the *model counting problem* for linear-time properties. Given a language over infinite words, the model counting problem asks for the number of words, the *models*, that are defined by the language. For model checking, this means that we are not only interested in the existence of a trace in the system that violates a given linear-time property, but also in the number of such traces. We show that model counting opens up new applications in formal verification, and lays a new foundation for solving a variety of quantitative verification problems.

In practice, constructing an implementation of a system that fully satisfies all the requirements posed by the specification is mostly not possible. This calls for analysis methods that reflect the degree of satisfaction and its impact on the safety and security of the system. Model counting can be used to determine the levels of satisfaction in a system by computing the number of system traces on which a requirement is violated. The number of traces can be used to determine the likelihood of satisfying a requirement. For example, in cases where we only have a partial model of the environment, the number of traces can be used to determine the number of unanticipated environment scenarios, that may cause the system to violate the specification [119]. Model counting can also be used for evaluating the level of security of security-critical systems. The leakage of information in a system is in general unavoidable, as some leakage of information is necessary to maintain a certain level of functionality (A password checker needs to inform a user whether an entered phrase is correct or wrong). Nevertheless, bounding the amount of information leaked by a system is necessary to keep the uncertainty of an adversary at levels, high enough to prevent the adversary from guessing the secrets in the system. One way to measure the leakage of information in a system is by relating the number of outputs observable by the adversary to the number of secret inputs to the system, which determines the likelihood of guessing the secret data by an adversary [139].

We solve the model counting problem for a variety of logical formalisms for the specification of reactive systems, covering automata-based formalisms [143] and temporal logics [126, 44]. Here, we distinguish between two types of model counting problems. The *infinite-trace* counting problem, where the models are the infinite traces of the system, and *bounded-trace* counting problems, where the models are defined as bounded finite traces of the system. This distinction is made with respect to the class of the specification under scrutiny. Specifications that define *safety* properties [108] for the system, i.e., properties whose violations can be observed over finite traces of the system, can be solved by considering finite traces of the system with increasing lengths. Specifications that define *liveness* properties [1] for the system, i.e., properties that define the progress of the system, can only be checked by looking at the infinite behavior of the system. In this case, the model counting problem can be either defined over infinite traces, or finite representations of those, for example by considering ultimately periodic traces with periods of bound size. We present both automata-theoretic and symbolic approaches for solving model counting problems for the different specification formalisms and the different types of specification classes, and show how these new algorithms can be used in solving quantitative verification problems such as probabilistic model checking [104, 94], model checking quantitative information-flow policies [139], and the synthesis of approximate implementations for reactive systems [58], where implementations are approximated by the number of input sequences for which the specification is satifisfied.

The model counting problem presented in this thesis follows a long history of model counting problems investigated for propositional logic. Introduced by Valiant in 1979 [147], the propositional model counting problem has been the most studied counting problem for logical formalisms [77]. Propositional model counting generalizes the satisfiability problem for propositional formulas, to a problem that asks for the number of assignments that satisfy a propositional formula. The model counting problem for linear-time properties is a generalization of the propositional model counting problem, where we not only ask for the number of valuations of propositions but the number of such valuations over time. We start with a brief introduction to the different model counting problems defined for propositional logic and show how we can generalize these problems to new counting problems for linear-time properties.

1.1 Propositional Model Counting

1.1.1 Problem definition

Propositional model counting is the problem of computing the number of satisfying assignments for a given propositional formula [148]. Given a propositional formula ϕ over a set of propositional variables P, counting the models of ϕ is the problem of computing the size of the set

$$\{\nu : P \to \{0,1\} \mid \nu \text{ satisfies } \phi\}.$$

Besides its theoretical importance as the canonical counting problem for the complexity class #P [147], with the rise of efficient and scalable SAT solvers, model counting has recently gained a lot of attention on the algorithmic side [77]. Prominent examples of problems solved using propositional model counting are bounded adversarial planning, where model counting has been used to evaluate plans in incomplete domains [119], probabilistic reasoning, where DPLL-based counting algorithms[1] can be easily adapted to solving the Bayesian inference problem by replacing clause functions (that evaluate to 0 or 1) with probability functions over values between 0 and 1 [9], and quantitative information flow, where, using the notion of min-entropy [139], computing the amount of information leakage in security-critical systems boils down to computing the number of possible reachable states of a program, which in turn can be bounded by the number of possible outputs produced by the system [99, 154].

1.1.2 Generalizations of propositional model counting

Two prominent generalizations of propositional model counting are the problems of *projected model counting* and *maximum model counting*.

In projected model counting, we count the satisfying assignments of a propositional formula projected to a given set of propositional variables [8]. Formally, for a propositional formula ϕ over a set of propositional variables P, and for a set of propositional variables $X \subseteq P$, projected model counting asks for computing the size of the set

$$\{\nu \colon X \to \{0,1\} \mid \exists \nu' \colon P \to \{0,1\}, \nu \preceq \nu', \text{ and } \nu' \text{ satisfies } \phi\}$$

where $v \preceq v'$ holds if ν and ν' agree on the valuations of the variables in X, i.e., for all $x \in X$ it holds that $\nu(x) = \nu'(x)$.

[1] The Davis-Putnam-Logemann-Loveland (DPLL) algorithm for solving the satisfiability problem of propositional formulas can be adapted as a counting algorithm, where the solving procedure continues looking for satisfying assignments as long as new ones can be found.

Projected model counting can be used for checking the robustness of non-deterministic plans with respect to initial configurations [8]. For a set of initial states and a set of goal states, one would like to know for how many initial states there are plans that succeed in reaching a goal state. Independent of the actions of the plan and the reached goal state, projected model counting allows us to compute the number of such initial states by computing the number of assignment projections projected to the variables defining the initial states.

In maximum model counting [73], one maximizes the number of satisfying assignments for a given set of propositional variables, called the counting variables, over another set of propositional variables, known as the maximization variables. For a propositional formula ϕ over a set of propositional variables P, a set $X \subseteq P$, and a set $Y \subseteq P$, maximum model counting is the problem of computing the value

$$\max_{\nu: Y \to \{0,1\}} |\{\nu' \colon X \to \{0,1\} \mid \exists \nu'' \colon P \to \{0,1\}, \nu, \nu' \preceq \nu'', \text{ and } \nu'' \text{ satisfies } \phi\}|.$$

Here, the sets Y and X represent the sets of maximization and counting variables, respectively.

Maximum model counting can be used for model checking quantitative information-flow policies. In security-critical systems, we can distinguish between two types of inputs: secret and public inputs. An external observer of the system should not deduce any information about the secret inputs by observing the outputs of the system. A quantitative information-flow policy may require that the amount of information leaked to the external observer about the secret inputs is bounded. By bounding the number of observable outputs, we are able to limit the leakage of information to the external observer. Maximum model counting can be used to check whether the system respects this bound by maximizing the number of observable outputs over all possible public inputs to the system[2].

1.2 Model Counting for Reactive Systems

1.2.1 Problem definition

Traces of reactive systems can be represented by sequences of evaluations over a set of atomic propositions $AP = I \cup O$ defining the input values received from the environment, I, and the output values produced by the system, O. Propositional model counting can thus be lifted to model counting for linear-time prop-

[2] In channel capacity terms one would compute the entropy over this maximum value [99].

erties by lifting the satisfaction from a set of propositions P to a sequence of valuations over a set of atomic propositions AP. Formally, a linear-time property φ over AP is defined as the set $\varphi \subseteq (2^{AP})^{\omega}$ [13]. Given φ, the model counting problem asks for computing the number of words $\sigma \in \varphi$, i.e., the size of the set defined by φ. Regarding model checking, computing the number of violations in a system T, with respect to a linear-time property φ, is done by solving the model counting problem for the set defined by $\mathit{Traces}(T) \cap \overline{\varphi}$, where $\overline{\varphi}$ is the complement language of φ, and $\mathit{Traces}(T)$ is the set of traces of T.

Generalizations of the model counting problem for linear-time properties can be presented in a similar fashion as for propositional logic. Projected model counting is the problem, where given a linear-time property φ over a set of atomic propositions AP, and a set $X \subseteq AP$, we want to compute the number of different models of φ projected to atomic propositions in X. Maximum model counting defines the problem where given sets $X, Y \subseteq AP$, we want to maximize the number of models associated with a unique evaluation of the propositions in Y, after projecting them to the values of propositions in X. Projected model counting and maximum model counting are relevant for solving quantitative verification problems for security-critical reactive systems. In such systems, an adversary who is observing the sequences of outputs of the system should not deduce anything about the sequences of secret inputs that lead to producing these outputs. Projected model counting can be used to determine the degree of deniability by counting the number of different output sequences produced by the system, and comparing this number to the total number of secret inputs. A system is secure if the number of secret inputs is much higher than the number of secret outputs. Maximum model counting can further be used to check that the number of different outputs for each public input does not exceed a certain threshold, thus bounding the amount of information leaking from the secret inputs to the public outputs with respect to the individual public inputs to the system [44, 99].

Excursion 1.1 (Branching-time properties)**.** Specifications for reactive systems can be also defined using branching-time logics. Branching-time logical formalisms allow us to write properties of states by existentially or universally quantifying over executions that start in a specific state. For example, a branching-time specification could state that "at every position in an execution, there is always a continuation that can lead back to the initial state" [13]. Such a property cannot be defined by any linear-time logic as

we only require some continuation from a state to lead back to the initial state, and not all of them.

Prominent examples of branching-time logics include the logics CTL [19, 42, 60] and CTL* [61]. The semantics of these two logics is defined over infinite trees, instead of infinite words. Trees represent the full infinite behavior of an implementation of a reactive system, where branches of the trees represent the input from the environment and nodes of the tree represent a state of the implementation.

The counting problem for branching-time logical formalisms is out of the scope of this thesis. We will nevertheless give a glimpse of a possible definition of this problem in our final discussion.

1.2.2 Bounded model counting

In addition to counting the number of traces violating the specification, we are also interested in determining the degree of violation up to a certain time point, or under restrictions on the behavior of the environment. Motivated by techniques such as bounded model checking [22], and bounded synthesis [67, 101], where violations in a system are restricted to bounded traces, we also look into bounded model counting problems for linear-time properties.

Different types of properties allow for the definition of different types of bounded models. In general, we can classify linear-time properties according to their types of violations [13]. A linear-time property can be a combination of safety and liveness properties [1, 2, 108], distinguishing between violations that can be detected by observing a finite execution of the system, and violations that can only be detected by inspecting the infinite behavior of the system. In the following, we give an overview of these different types of properties along with their associated finite models.

Safety properties. A safety property is a linear-time property that requires that *"a bad thing never happens"* [108]. To determine whether an implementation of a system violates a safety property we need to check whether the *"bad thing"* is observable on one of its traces. For example, consider the safety property that requires an autonomous car to never cross a red light. A violation of the property is a trace where the car passes a red light. Notice that the action of the car is irreparable. Once the car has passed the red light, the safety property is violated forever. We call the finite trace that includes "the bad thing", in the case

of the car the action of passing the red light, a *bad prefix* for the safety property, which is a prefix of a trace that proves that the system violated the property.

By bounding the length of bad prefixes to some bound n, we can count the number of bad prefixes for a safety property in a system that are of length n. We refer to this problem as the bounded model counting problem for bad prefixes. Counting bad prefixes can be used to determine the likelihood of an error occurring in a system within n steps after the system has been deployed.

Liveness properties. A *liveness* property is a linear-time property that requires that *"a good thing happens"* [1]. An example liveness property is one that requires a traffic light to eventually turn green, every time a car approaches the traffic light. In contrast to safety properties, a violation of a liveness property cannot be detected by considering finite traces of the system. In order to determine whether a liveness property is violated, we need to look at the full traces of the system. A violation of a liveness property in the system can be detected by finding a trace where the good thing is prevented from ever happening again. This is usually done by searching for a loop in the implementation of the system, a repeated sequence of inputs that keeps the system from reaching the good thing. An iterative approach for the detection of violations of a liveness property is one that looks for increasing sequences of inputs, if repeated forever, will block the system from satisfying the liveness property.

By bounding the length of finite traces leading to such loops and bounding the size of the loops themselves, we define bounded finite representations for violations of liveness properties. We call such finite representations *lassos*. A lasso represents a trace of a system where, after performing a finite number of steps, the system enters a periodic behavior that is performed forever. The size of the lasso is the joint size of its stem (the execution leading to the loop) and the size of its loop. Given a linear-time property, the bounded model counting problem for lassos is defined as the problem of computing the number of lassos up to a certain size that represent traces that violate (or satisfy) the linear-time property. Counting the number of lassos can be used to determine the likelihood of a system ending up in a loop that prevents the system from fulfilling its tasks.

***Remark* 1.1.** *Liveness [1, 108] includes properties that require "a good thing to eventually happen" as well as properties that require that "the good thing happens an infinite number of times". We will investigate these two types of liveness properties separately. The first type is defined by the class of co-safety properties. In contrast to other liveness properties, co-safety properties can be checked by looking for finite traces where the "good thing" has already occurred. Such finite traces are called good prefixes.*

1.3 Contributions

The contributions of this thesis can be divided into two parts. In the first part, we introduce the model counting problem for linear-time properties and provide algorithms for solving the problem for properties given in LTL [126], and as automata over infinite words [143]. Building on that, we establish the relation between model counting and probabilistic model checking [104]. In the second part, we show the role of model counting algorithms in solving quantitative verification problems such as model checking quantitative information-flow policies [139], and approximate synthesis for reactive systems [63, 125].

1.3.1 Part I: The model counting problem

We introduce four model counting problems for linear-time properties: The *infinite trace counting problem*, where we count the infinite traces defined by a given linear-time property, and the *bounded model counting problems* for bad-prefixes, good-prefixes, and lassos, where we compute the number of bad prefixes, good prefixes of bounded length and lassos of bounded size for a linear-time property.

For the prominent subclass of omega-regular properties [121], we introduce algorithms for solving the model counting problems for properties given as automata over infinite words [78, 143], and as formulas in linear-time temporal logic (LTL) [126]. We introduce automata-based algorithms for each problem as well as symbolic encodings of these problems in propositional logic and give a thorough complexity analysis determining a lower and upper bound for each problem. Our complexity analysis is done in terms of both the traditional decision complexity classes and the counting complexity classes [124]. For the latter, we show that the traditional definitions of counting complexity classes [147, 148] do not suffice for capturing all model counting problems presented in this thesis.

We further establish the relation between the probabilistic model checking problem for discrete-time Markov chains [95] and each of the counting problems introduced in this thesis. Probabilistic model checking is a formal verification technique for verifying properties of systems with probabilistic behavior [13, 104]. Probabilistic modeling is essential, for reasoning about the reliability of systems. For example, we can use probabilistic reasoning to quantify the likelihood that a system fulfills its task given the probability for the different environment scenarios that the system may encounter while it is running. We revisit probabilistic model checking for linear-time properties, and show how model counting can be used to solve this problem.

1.3.2 Part II: Applications of model counting in formal verification

We demonstrate the role of model counting in two different applications. The first application is related to security-critical systems. We introduce new techniques based on model counting for solving model checking problems of quantitative information-flow policies for reactive systems. The second application involves the synthesis of reactive systems. We show that model counting can be used for the automatic construction of approximate implementations for reactive systems, allowing us to synthesize implementation even when the specification is unrealizable. In the following, we briefly elaborate on each of these applications.

Model checking quantitative hyperproperties. We show that model counting can be used for model checking quantitative information-flow policies for security-critical systems. In security-critical systems, we distinguish between public and secret inputs. In quantitative information-flow, the goal is to bound the amount of information leaked from the secret inputs to the public outputs.

Verifying information-flow policies requires reasoning about multiple executions in the system. *Hyperproperties* [46] allow for such reasoning by defining sets of sets of executions that are allowed to coexists in the system (one can see hyperproperties as a set of linear-time properties). We study the model checking problem for a certain type of hyperproperties, which we define as counting hyperproperties. Counting hyperproperties define hyperproperties that express a bound on the number of executions that may appear in a certain relation. For example, quantitative noninterference limits the amount of information about certain secret inputs that is leaked through the observable outputs of a system. Quantitative noninterference thus bounds the number of executions that have the same observable input but a different observable output. We study counting hyperproperties in the setting of the logic HYPERLTL [44], a temporal logic for hyperproperties. We show that, while quantitative hyperproperties can be expressed in HYPERLTL, the running time of the traditional HYPERLTL model checking algorithm is, depending on the type of property, exponential or even doubly exponential in the quantitative bound [66]. We improve this complexity with a new model checking algorithm based on maximum model counting. The new algorithm needs only **logarithmic** space in the bound and therefore improves, depending on the property, exponentially or even doubly exponentially over the traditional model checking algorithm of HYPERLTL [66].

Synthesis of approximate implementations. In synthesis, an implementation of a system is automatically constructed from its formal specification [125]. The advantage of synthesis is that the constructed implementation is inherently correct with respect to the given specification and no further debugging is needed in that respect.

Synthesis shifts the task of developers from implementing the system to writing specifications for the system. The latter task comes however with its own challenges, as writing correct specification is not always that simple. Specifications tend to quickly become unrealizable, i.e., there is no implementation that fulfills the specification on all environmental behavior.

The unrealizability of a specification is often due to the assumption that the behavior of the environment is unrestricted. In this thesis, we use model counting to construct *approximate* implementations for reactive systems. The approximation is based on fulfilling the specification for restricted environments, in our case, these are bounded environments, where the environment can only generate input sequences that are ultimately periodic words (lassos) with finite representations of bounded size. The synthesis problem is thus transformed into an optimization problem, where we search for an implementation that fulfills the specification over a maximum number of environment inputs. We provide automata-theoretic and symbolic approaches for solving this synthesis problem, which use model counting to approximate the behavior of an implementation under the given bounded environment.

1.4 Related Work

Quantitative approaches to formal verification. Quantitative approaches have a long tradition in formal verification [85, 86, 88, 90, 94, 105, 139, 141]. In contrast to the classical view, where a specification is either satisfied or violated by a system, quantitative approaches give a more informative perspective on up to what extent the specification is fulfilled or violated by the system, and can thus be used to evaluate certain tradeoffs in the implementation of systems.

Shortly after the introduction of model checking in the early eighties [43, 128], several works on quantitative extensions of model checking have started to follow. Most of the proposed approaches where based on probabilistic notinons [14, 49, 82, 127, 149], ones that were later expanded to newer notions with cost and reward functions [11, 12, 71, 94], that allowed to tackle robustness problems, where we are interested in the stability of the system against unanticipated perturbations in the environment [26, 72, 141, 142].

With the development of reactive systems, quantitative notions from other fields have found their way to formal verification. These include prominent notions like entropy [47, 137][3], which has closely become associated with quantitative verification problems for security-critical systems [139]. The notion of entropy, as used in computer science, was originally introduced by Claude Shannon as a measure of the uncertainty of information received in an event sent over a communication channel [137]. The more information the event carries, the more the uncertainty about this information is. In information-flow control, entropy can be used to measure the amount of information leaked out of the system [97, 139].

The probabilistic model checking problem for linear-time properties can be approximated by bounded model counting in all variants presented above. Furthermore, solving quantitative verification problems based on specific entropy notions can also be done using model counting. Such notions include min-entropy [139], which quantifies the amount of information about a secret gained by an adversary by a single guess, or notions like language entropy [7, 38], which defines the growth rate of a language and can be used to determine the uncertainty of approximating a language with another language.

The model counting problem represents the class of quantitative verification problem, where measures computed over the individual traces of a system are aggregated to a measure for evaluating the whole system. Another class of counting problems in formal verification includes those that are concerned with counting the satisfaction of a property along the individual executions of the system. Examples of such counting problems include verification problems for counting logics, where the frequency of violations can be measured along an execution in a system system [28, 53, 110]. In contrast to the model counting problem, such approaches do not present quantitative measures for the evaluation of a system beyond the simple binary satisfaction a property along each execution of the system. Model counting may be applied on top of these approaches to determine the satisfaction of the quantitative properties over the whole system.

Counting words of regular languages. The model counting problem has also been studied for regular languages. Given a finite automaton, the problem is to compute the number of words of a certain length accepted by the automaton [93]. Counting words of finite automata can be used for random generation

[3]The notions of entropy goes back to Rudolf Clausius, who used the term to describe the content of a transformation during a change of a state in a physical system [47]. Here we use the term entropy for information entropy as introduced by Claude Shannon [137].

algorithms [21]. In contrast, the model counting introduced in this thesis is defined over omega-regular languages, i.e., languages over infinite words. Concerning our bounded counting problems, we show that algorithms for solving model counting for regular languages can be adapted to solving model counting problems for bad and good prefixes.

Propositional model counting. A variety of algorithms for solving the model counting problem and its generalizations for propositional logic have been proposed [77]. In general, the algorithms can be split into exact model counting algorithm and approximate model counting algorithms. In the following, we give a brief overview of the different types of algorithms developed for solving both types of algorithms [4].

Exact counting algorithms are either based on exhaustive search techniques like DPLL, or on conversions into certain normal forms. In DPLL-based approaches, the search tree is traversed using similar optimizations to search-based SAT-solvers such as pruning unsatisfiable branches or terminating the search in a branch as soon as all clauses have at least one true literal. Examples of DPLL-based model counting tools include CDP [25], Relsat [17], Cachet [135], and sharpSAT [144]. In conversion-based approaches, the propositional formula is transformed into another normal form where the model counting problem can be solved efficiently. For example, the tool c2d [52] uses an algorithm based on a transformation to deterministic decomposable negation normal form (d-DNNF) [51].

Approximate counting algorithms are mostly based on sampling methods. Examples of these algorithms are ones implemented in the tools ApproxCount [153], SampleMiniSat [74], SampleCount [76], and ApproxMC [33, 34, 116].

Approaches for projected model counting have been proposed in the tools #Clasp [8], and dSharp [107]. For maximum model counting, an approximate approach was presented in the tool MaxCount [73].

***Remark* 1.2.** *For more information about further topics related to the model counting problem of linear-time properties, and to each of the problems presented in this thesis we refer the reader to the bibliographic remarks at the end of each chapter.*

[4] For more details on the counting algorithms, we refer the reader to the survey presented in [77]

1.5 Publications

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

- Rayna Dimitrova, Bernd Finkbeiner, and Hazem Torfah (2019). **Synthesizing Approximate Implementations for Unrealizable Specifications**. In Proceeding of the 31st International Conference on Computer Aided Verification **(CAV 2019)** [58].

- Bernd Finkbeiner, Lennart Haas, and Hazem Torfah (2019). **Canonical Representations of k-Safety Hyperproperties**. In Proceeding of the 32nd IEEE Computer Security Foundations Symposium **(CSF 2019)** [64].

- Rayna Dimitrova, Bernd Finkbeiner, and Hazem Torfah (2019). **Approximate Automata for Omega-regular Languages**. In Proceeding of the 17th International Symposium on Automated Technology for Verification and Analysis **(ATVA 2019)**.

- Bernd Finkbeiner, Christopher Hahn, and Hazem Torfah (2018). **Model Checking Quantitative Hyperproperties**. In Proceeding of the 30th International Conference on Computer Aided Verification **(CAV 2018)** [65].

- Bernd Finkbeiner and Hazem Torfah (2017). **The Density of Linear-time Properties**. In Proceeding of the 15th International Symposium on Automated Technology for Verification and Analysis **(ATVA 2017)** [69].

- Hazem Torfah and Martin Zimmermann (2014). **The Complexity of Counting Models of Linear-time Temporal Logic**. In Proceeding of the 34th International Conference on Foundations of Software Technology and Theoretical Computer Science **(FSTTCS 2014**, journal version **Acta Informatica 2018)**[145, 146].

- Bernd Finkbeiner and Hazem Torfah (2014). **Counting Models of Linear-time Temporal Logic**. In Proceeding of the 8th International Conference on Language and Automata Theory and Applications **(LATA 2014)** [68].

Part I

The Model Counting Problem

Chapter 2

Models and Properties

In this chapter we lay the foundations for modeling and specifying reactive systems. We use finite labeled transition systems for representing implementations, and linear-time properties for representing specifications for reactive systems. We are especially interested in subclasses of linear-time properties defined by omega-automata and temporal logics. In the next chapter, we will present model counting algorithms for specifications that fall within these subclasses.

2.1 Finite Labeled Transition Systems

We model implementations of reactive systems by *finite labeled transitions systems*. Labeled transitions systems represents the relation between the actions to be performed by a system, so called outputs of the system, and the readings from the surrounding environment, the inputs to the system. Formally, a labeled transition system is defined by a tuple $T = (AP, I, O, S, s_0, \tau, L)$ where:

- $AP = I \cup O$ is a set of atomic propositions that define the inputs signals I and the output signals O.
- S is a set of states a system may reach during runtime. The *size* of the system, denoted by $|T|$ is measured by the size of its set of states. The size of a transition system determines the memory needed to implement the system.

- s_0 represents the state in which the system initially starts.
- $\tau\colon S \times 2^I \to S$ is the transition relation of the system that determines the change in state after receiving new readings from the environment.
- $L\colon S \to 2^O$ is a labeling function that determines the output of the system in each state.

We consider transition systems that are finite, i.e., transition systems with a finite set of states. Furthermore, throughout our study, we assume that the transition systems are deterministic and input enabled, i.e., the transition relation is a total function that maps a state $s \in S$ and an input $i \in 2^I$ to exactly one state $s' \in S$.

In general, a transition system can be captured by its set of executions. Formally, an *execution* of a transition system T is a sequence $\rho : \mathbb{N} \to S \times 2^I$ of states and inputs that follows the transition function τ, i.e., for all $i \in \mathbb{N}$ if $\pi(i) = (s_i, e_i)$ and $\pi(i+1) = (s_{i+1}, e_{i+1})$, then $s_{i+1} = \tau(s_i, e_i)$. We call an execution *initial* if it starts with the initial state: $\pi(0) = (s_0, e)$ for some $e \in 2^I$. For an initial execution π of T, we call the sequence $\sigma_\pi\colon \mathbb{N} \to 2^{AP}.\ i \mapsto o(s_i) \cup e_i$, where $\pi(i) = (s_i, e_i)$, the *trace* of π. We denote the set of traces of a transition system T by the set $\mathit{Traces}(T)$ and define the language of T by its set of traces, i.e., $L(T) = \mathit{Traces}(T)$.

As we will see in the next section, a specification is given as a set of valid traces that are allowed to be observed in a system under scrutiny. To check a system against a property, we check the language of the system against that property.

Figure 2.1 shows an example of a labeled transition system that implements a controller that manages the access of two processes to a shared resource. The controller receives two input signals $\mathit{request}_1$ and $\mathit{request}_2$ via which the processes can request permission to enter the shared resource. Via the output signals grant_1 and grant_2, the controller grants each processes permission to use the shared resource. The transition system on the right, represents an implementation of the controller C that manages the entrance of the two processes in a round robin fashion, i.e., regardless of the input values, given by the symbol $\top$ to resemble all possible inputs, the system permanently outputs a grants to process p_1 and p_2 interchangeably, starting with grant_1.

In the rest of this thesis and for the matter of convenience we will represent the labels of the inputs in a transition system symbolically. For example, if the set of inputs is $\{r_1, r_2\}$, then an edge labeled with the symbol r_2, represents two

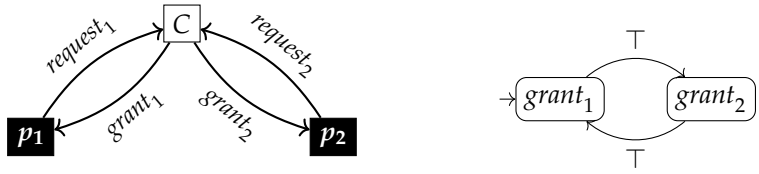


Figure 2.1: A system composed of a controller C and two processes p_1 and p_2, and a transition system T that implements the controller C. For input signals $I = \{request_1, request_2\}$ and output signals $O = \{grant_1, grant_2\}$, the transition system T defines an implementation of the controller C that grants two processes access to a shared resource in a round robin fashion.

transition, for the inputs $\{r_1, r_2\}$ and $\{r_2\}$. The symbol $\top$ then represents four transitions for each of the possible inputs from $2^{\{r_1, r_2\}}$.

2.2 Linear-time Properties

We represent specifications of reactive systems in this work by linear-time properties, which define sets of infinite traces that are allowed to be produced by the system. A linear-time property can be seen as the language of infinite words, each word describing a valid trace. An infinite word over an alphabet Σ is a sequence $\sigma : \mathbb{N} \to \Sigma$ of letters from Σ. We denote the set of all infinite words over Σ by the set Σ^ω. Given an index $i \in \mathbb{N}$ the letter of a word σ at position i is the letter equal to $\sigma(i)$. Given some alphabet Σ, a linear-time property over Σ is then a set $\varphi \subseteq \Sigma^\omega$. An example linear-time property can be given by the set $\{\sigma \in \{a, b\}^\omega \mid \forall i \in \mathbb{N}. \exists j \in \mathbb{N}.\ \sigma(j) = a\}$ that defines the set of words over the alphabet $\{a, b\}$ where the letter a appears infinitely often. We call a word $\sigma \in \Sigma^\omega$ a *model* of a linear-time property φ if $\sigma \in \varphi$. If $\sigma \notin \varphi$, then σ is called a *violation* of φ.

In the case of a reactive system that is defined over a set of atomic propositions AP, a linear-time property for this system is a subset of the set $(2^{AP})^\omega$. For a transition system T defined over AP and for a linear-time property φ over $(2^{AP})^\omega$, we define the set $Models(T, \varphi) = \{\sigma \in Traces(T) \mid \sigma \in \varphi\}$ as the set of traces that are models of φ, and the set $Violations(T, \varphi) = \overline{Models(T, \varphi)}$ as the set of traces that are not models for φ. We say that the transition system T satisfies the linear-time property φ if $Traces(T) \subseteq \varphi$, denoted by $T \models \varphi$. For example, given the set of atomic propositions $AP = \{request_1, request_2, grant_1, grant_2\}$, consider the properties $\psi_1 = \{\sigma \in (2^{AP})^\omega \mid \forall i \in \mathbb{N}.\ grants_1 \notin \sigma(i) \vee grants_2 \notin \sigma(i)\}$ and $\psi_2 = \{\sigma \in (2^{AP})^\omega \mid \forall i \in \mathbb{N}.\ request_1 \notin \sigma(i) \to grant_1 \notin \sigma(i+1)\}$. The

property $\psi_1$is a mutual exclusion property that requires that the shared resource managed by the controller C in is not entered simultaneously by the processes p_1 and p_2. The transition system in Figure 2.1 satisfies the property ψ_1, because every traces in the transition system, allows only one of the grants to be true at a time. The transition system does not however satisfy the property ψ_2, that requires that a process is granted access only if the process requested access in the step before. A violation in the transition system can be given by the any traces that starts with the prefix $\{grant_1, request_1\}\{grant_2, request_1\}\ldots$, where process p_2 is granted access without requesting to enter the shared resource.

Sometimes we are only interested in a subset of signals and define properties only for this subset. Given a trace $\sigma \in \mathit{Traces}(T)$ for some transition system T over a set of atomic propositions AP and given a set $C \subseteq AP$, we define a projection of σ to C by a trace $\sigma_C \in (2^C)^\omega$, such that, for all $c \in C$ and for all $i \in \mathbb{N}$, $c \in \sigma(i)$ if and only if $c \in \sigma_C(i)$. Given a set of traces $\Gamma \subseteq \mathit{Traces}(T)$ a projection of Γ to the propositions C is defined by the set $\Gamma_C = \{\sigma_C \mid \sigma \in \Gamma\}$.

Linear-time properties can be classified to *safety*, *co-safety*, and *liveness*, or a combination of these properties. In the following we define these classes and give some example properties that fall into each class.

2.2.1 Safety and co-safety

For an infinite word $\sigma \in \Sigma^\omega$, we say that, a finite word $w \in \Sigma^*$, where Σ^* is the set of all finite words over Σ, is a *prefix* of σ, denoted by $w < \sigma$, if there is a position $i \in \mathbb{N}$, such that, $w = \sigma(0) \cdot \ldots \sigma(i)$. For a position $i \in \mathbb{N}$, we denote the prefix of σ of length i by $\sigma[0, i]$.

Given a linear-time property φ over an alphabet Σ, we call a finite word $w \in \Sigma^*$ a *bad prefix* for φ, if for any infinite word $\sigma \in \Sigma^\omega$, the word $\sigma' = w \cdot \sigma$ is a violation of φ, i.e., $\sigma' \notin \varphi$. For a linear-time property φ we denote the set of all bad prefixes of φ by $\mathit{BadPref}(\varphi)$. For a given bound $n \in \mathbb{N}$ on the length of bad prefixes, we define the set $\mathit{BadPref}(\varphi, n) = \{w \in \mathit{BadPref}(\varphi) \mid |w| = n\}$ to contain all bad prefixes of φ of length n.

We call a linear-time property φ over an alphabet Σ a *safety property*, if for every $\sigma \notin \varphi$, there is a position $i \in \mathbb{N}$, such that, $\sigma[0, i] \in \mathit{BadPref}(\varphi)$. An example of a safety property is the property ψ_1 of mutual exclusion we presented above. A violation of this property is a trace σ, such that, in some position $i \in \mathbb{N}$, we have that $grant_1 \in \sigma(i)$ and $grant_2 \in \sigma(i)$, and thus, the prefix $\sigma[0, i]$ is a bad prefix for the property ψ_1.

The dual property for a safety property is a *co-safety property* that is defined based on the notion of *good prefixes*. A finite word $w \in \Sigma^*$ is a *good prefix* for a linear-time property φ over the alphabet Σ, if for any infinite word $\sigma \in \Sigma^\omega$, the word $\sigma' = w \cdot \sigma$ is a model of φ, i.e., $\sigma' \in \varphi$. For a linear-time property φ we denote the set of all good prefixes of φ by $\mathit{GoodPref}(\varphi)$. For a given bound $n \in \mathbb{N}$ on the length of good prefixes, we define the set $\mathit{GoodPref}(\varphi, n) = \{w \in \mathit{GoodPref}(\varphi) \mid |w| = n\}$ to contain all good prefixes of φ of length n. A linear-time property φ over an alphabet Σ a *co-safety property*, if for every $\sigma \in \varphi$, there is a position $i \in \mathbb{N}$, such that, $\sigma[0,i] \in \mathit{GoodPref}(\varphi)$. Co-safety properties include all properties where a certain guarantee needs to be reached at some bounded or unbounded point of time. For example, considering again the set of atomic propositions $AP = \{\mathit{request}_1, \mathit{request}_2, \mathit{grant}_1, \mathit{grant}_2\}$, the following co-safety property $\psi_3 = \{\sigma \in (2^{AP})^\omega \mid \exists i \in \mathbb{N}.\, \mathit{grant}_1 \notin \sigma(i) \wedge \mathit{grant}_2 \notin \sigma\}$ requires that there is a point of time where the shared resource is free. Every model σ of ψ_1 must have a point i where neither grant_1 nor grant_2 is true. The prefix $\sigma[0,i]$ is a good prefix for φ. The transition system in Figure 2.1 does not satisfy ψ_3 as no trace of the transition system has a good prefix for ψ_3.

2.2.2 Liveness

A linear-time property φ over an alphabet Σ is a *liveness property*, if every finite word $w \in \Sigma^*$ there is an infinite word $\sigma \in \Sigma^\omega$, such that, $\sigma' = w \cdot \sigma \in \varphi$. A liveness property is for example the property $\psi_4 = \{\sigma \in | \ \forall i \in \mathbb{N}.\, \mathit{request}_1 \in \sigma(i) \rightarrow \exists j \in \mathbb{N}.\, j \geq i \wedge \mathit{grant}_1 \in \sigma(j)\}$ for a set $AP = \{\mathit{request}_1, \mathit{request}_2, \mathit{grant}_1, \mathit{grant}_2\}$. The property ψ_4 requires a request from process p_1 to be eventually answered with a grant to access the shared resource. Since the time when to answer the request is not specified, any finite trace where a request from the process is observed can be extended by any infinite word where at some point the process is granted access.

The sets of liveness properties and safety are disjoint sets of properties. For some safety and liveness properties there is however a duality relation. The reason for that is that the sets of liveness properties and co-safety properties are not disjoint. Some co-safety properties are also liveness properties. For example, unbounded reachability properties are the most famous example of such properties. Consider again the property ψ_3 from above. This property is both a co-safety and a liveness property. Liveness properties that are not co-safety, are usually ones that either include *response* properties that require a certain property to be repeated infinitely often, or *persistence* properties, where

eventually an invariant must hold [1, 35]. An example of a response property is the property ψ_4. An example of a persistence property is the property $\psi_5 = \{\sigma \in (2^{AP})^\omega \mid \exists i \in \mathbb{N}. \forall j \in \mathbb{N}.\ j \geq i \rightarrow \mathit{grant}_1\}$ which requires that at some point in time only process p_1 will be granted access to the resource.

2.3 Omega-Regular Properties

In this section we define the widely used class of ω-regular properties, which generalizes the definition of regular properties to infinite words [29, 121]. We define ω-regular properties by introducing different types of ω-automata that capture the languages defined by ω-regular properties [143]. We reintroduce the classes of parity, Büchi and co-Büchi automaton in all their branching variants and show how they are related and the area of application of each type of automaton.

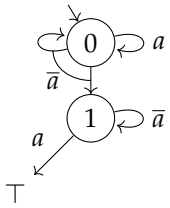
2.3.1 Parity automata

An *alternating parity automaton* is a tuple $P = (\Sigma, Q, Q_0, \delta, \mu)$ where:

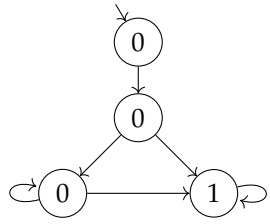
- Σ is the alphabet over which the automaton is defined.
- Q is the set of states of the automaton. The size of the automaton is defined by the size of the set of its states, i.e., $|P| = |Q|$.
- $Q_0 \subseteq Q$ is the set of initial states.
- $\delta\colon Q \times \Sigma \rightarrow \mathbb{B}^+(Q)$, where $\mathbb{B}^+(Q)$ is a positive boolean combination over the set of states Q, is the transition relation of the automaton.
- $\mu\colon Q \rightarrow C \subset \mathbb{N}$ is a coloring function that maps each state of the automaton with a natural number from the finite set C.

The ω-regular property over an alphabet Σ defined by a parity automaton is the language defined by this automaton, i.e., the set of infinite words from Σ^ω accepted by this automaton. The acceptance of an infinite word from Σ^ω by an alternating parity automaton is defined as follows.

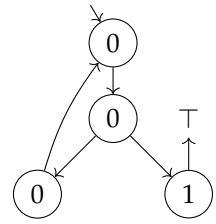
Acceptance in parity automata. A tree T over a set of directions D is a prefix-closed subset of D^*. The empty sequence ϵ is called the root. The children of a node $n \in T$ are the nodes $\{n \cdot d \in T \mid d \in D\}$. A Σ-labeled tree is a pair (T, l), where $l : T \rightarrow \Sigma$ is the labeling function. A *run* of an alternating parity



(a) An alternating parity automaton P for $\square\diamond a$



(b) A rejecting run of P for $a(\overline{a})^\omega$



(c) An accepting run of P for $(a\overline{a}a)^\omega$

Figure 2.2: An alternating automaton an its runs for the word $\sigma = a(\overline{a})^\omega$, and for the word $\sigma' = (a\overline{a}a)^\omega$.

automaton $P = (Q, Q_0, \delta, \mu)$ on an infinite word $\sigma = \alpha_0\alpha_1 \cdots \in \Sigma^\omega$ is a Q-labeled tree (T, l) that satisfies the following constraints:

1. $l(\epsilon) \in Q_0$
2. for all $n \in T$, if $l(n) = q$, then $\{l(n') \mid n' \text{ is a child of } n\}$ satisfies $\delta(q, \alpha_{|n|})$.

A run of P is *accepting* if every branch either hits a *true* transition, i.e., the last transition is of the form $\delta(q, \alpha) = \top$ for some state $q \in Q$ and letter $\alpha \in \Sigma$, or is an infinite branch $n_0n_1n_2 \cdots \in T$, and the sequence $l(n_0)l(n_1)l(n_2)\ldots$ satisfies the *parity condition*, which requires that the highest color occurring infinitely often in the sequence $\mu(l(n_0))\mu(l(n_1))\mu(l(n_2)) \cdots \in \mathbb{N}^\omega$ is even. An infinite word σ is accepted by P if there exists an accepting run of P on σ. The set of infinite words accepted by P is called its *language*, denoted $L(P)$.

Figure 2.2 shows an alternating parity automaton $P = (2^{\{a\}}, \{q_0, q_1\}, q_0, \delta, \mu)$ for the ω-regular property $\square\diamond a$. The automaton consists of two states q_0 and q_1. The initial state has color $\mu(q_0) = 0$ and the state q_1 has color $\mu(q_1) = 1$. The initial state q_0 has two outgoing transitions. A universal transition for the letter $\emptyset$, represented by the symbolic value $\overline{a}$, that reaches two states q_0 and q_1, i.e., $\delta(q_0, \emptyset) = q_0 \wedge q_1$. The other transition is a self-loop when the read letter is $\{a\}$, i.e., $\delta(q_0, \{a\}) = q_0$. The state q_1 has one transition for the letter $\emptyset$, that goes back to q_1, i.e., $\delta(q_1, \emptyset) = q_1$. For the letter $\{a\}$, the transition relation δ is accepting, i.e., $\delta(q_1, \{a\}) = \top$. Figure 2.2b shows the run for the infinite word $a(\overline{a})^\omega$. The run is represented by a graph that if unrolled is a $\{q_0, q_1\}$-labeled tree. The run is rejecting, because it has a branch where the highest infinitely often appearing color is odd. A run for the infinite word $(a\overline{a}a)^\omega$ is given in Figure 2.2c. The run

is accepting, because each branch of the run either ends with $\top$, or is an infinite branch where the highest color is even.

Parity automata can be categorized by the type of their transition relation. An alternating parity automaton $P = (\Sigma, Q, Q_0, \delta, \mu)$ is called *non-deterministic* if for all states $q \in Q$ and all letters $\alpha \in \Sigma$, the transition relation is a disjunction, i.e., $\delta(q, \alpha) = \bigvee_{q' \in Q'} q'$, for some set $Q' \subseteq Q$. The automaton P is called *universal*, if for all states $q \in Q$ and all letters $\alpha \in \Sigma$, the transition relation is a conjunction, i.e., $\delta(q, \alpha) = \bigwedge_{q' \in Q'} q'$, for some set $Q' \subseteq Q$. If P is both non-deterministic and universal, then P is called *deterministic*. All variants of parity automaton describe the same set of linear-time properties and form the so-called ω-regular fragment of linear-time properties. A fact we state by the following lemma and theorem.

Lemma 2.1 ([78, 118, 136])**.** *For every alternating parity automaton A there is*

- *a non-deterministic parity automaton N, with $L(N) = L(A)$. The size of N is exponential in the size of A.*
- *a deterministic parity automaton D, with $L(D) = L(A)$. The size of A is doubly-exponential in the size of A.*

Theorem 2.1 ([78, 114])**.** *The set of languages defined by the set of deterministic parity automata is equal to the set of ω-regular properties.*

2.3.2 Büchi automata

A Büchi automaton over an alphabet Σ is a parity automaton $B = (\Sigma, Q, Q_0, \delta, \mu)$ where the image of μ is restricted to the set $\{1, 2\}$. States with color 2 are called the accepting states of B. A run in B is accepting, if an accepting state appears infinitely often on that run. Instead of using a coloring function, we use the set F to define the set of accepting states of B, and thus the automaton can be given by the tuple $B = (\Sigma, Q, Q_0, \delta, F)$.

The set of languages defined by alternating Büchi automata over an alphabet Σ is equal to the set omega-regular properties over the same alphabet.

Theorem 2.2 (From parity to Büchi [102])**.** *For every alternating parity automaton P, there is an alternating Büchi automaton B of size polynomial in P.*

The same also hold for nondeterministic Büchi automata. In the next theorem, we show how to translate an alternating Büchi automaton, to an equivalent nondeterministic Büchi automaton.

Theorem 2.3 (Miyano-Hayashi [118]). *For every alternating Büchi automaton A, there is a nondeterministic Büchi automaton B of size $2^{O(|A|)}$ with $L(A) = L(B)$.*

Proof. Let $A = (\Sigma, Q, Q_0, \delta, F)$. We construct a nondeterministic Büchi automaton $B = (\Sigma, Q', Q'_0, \delta', F')$ as follows

- $Q' = 2^Q \times 2^Q$
- $Q'_0 = \{(\{q_0\}, \emptyset) \mid q_0 \in Q_0\}$
- $\delta = \{((\Gamma, \emptyset),\ \alpha,\ (\Gamma', \Gamma' \setminus F)) \mid \Gamma' \models \bigwedge_{q \in \Gamma} \delta(q, \alpha)\} \cup$
  $\{((\Gamma, Y),\ \alpha,\ (\Gamma', Y' \setminus F)) \mid Y \neq \emptyset,\ v' \subseteq \Gamma',$
  $\Gamma' \models \bigwedge_{q \in \Gamma} \delta(q, \alpha), Y' \models \bigwedge_{q \in Y} \delta(q, \alpha)\}$
- $F' = \{(\Gamma, \emptyset) \mid \Gamma \subseteq Q\}$

□

Deterministic Büchi automata are less expressive than nondeterministic Büchi automata, and in turn less expressive than omega-regular properties [109]. To construct an equivalent deterministic automaton for a nondeterministic Büchi automaton, we can translate the automaton to a deterministic parity automaton of exponential size.

Theorem 2.4 (Safra's Construction [134]). *For every non-deterministic Büchi automaton B there is a deterministic parity automaton P of size exponential in B, such that, $L(B) = L(P)$.*

2.3.3 Co-Büchi automata

An alternating co-Büchi automaton over an alphabet Σ is a parity automaton $C = (\Sigma, Q, Q_0, \delta, \mu)$ where the image of μ is restricted to the set $\{0, 1\}$. States with color 1 are called the rejecting states of C. A run in C is accepting, if a rejecting state appears only finitely often on that run. Instead of using a coloring function, we use the set R to define the set of rejecting states of C, and thus the automaton be can be given by the tuple $C = (\Sigma, Q, Q_0, \delta, R)$.

The set of languages defined by nondeterministic co-Büchi automata over an alphabet Σ is equal to the set omega-regular properties over the same alphabet. The set of languages defined by deterministic co-Büchi automata is however strictly weaker than the set of languages defined by nondeterministic co-Büchi

automata. Another subset of co-Büchi automata that is equivalent to the set of nondeterministic co-Büchi automata is the set of universal co-Büchi automata, since every nondeterministic Büchi automaton can be dualized to a universal co-Büchi automaton by replacing the nondeterministic transitions with universal ones and defining the set of accepting states as the set of rejecting states of the co-Büchi automaton.

Theorem 2.5. *For every non-deterministic Büchi automaton B there is a universal co-Büchi automaton U of equal size such that $L(B) = \overline{L(U)}$.*

2.4 Linear-time Temporal Logic

Linear-time temporal logic (LTL) is the most prominent temporal logic for reactive system proposed by Amir Pnueli in 1977 for defining linear-time properties [126], and has been, with the rise of model checking, the standard input language for many verification tools [22, 40, 89, 103]. LTL builds on propositional logic by adding operators that allow for the definition of truth values of the propositions over time. Although LTL does not cover the whole set of ω-regular languages, it allows for the definition of many important specifications of reactive systems.

For a set of propositions AP, an LTL formula over AP is given using the following syntax:

$$\varphi ::= a \in AP \mid \neg\varphi \mid \varphi \vee \varphi \mid \bigcirc \varphi \mid \varphi \,\mathcal{U}\, \varphi \ .$$

The *next* operator $\bigcirc$ states that a certain property should hold in the next step. For example, consider the formula $\bigcirc a$ defined over the set of atomic propositions $AP = \{a\}$. A model of $\bigcirc a$ is any infinite trace $\sigma \in (2^{AP})^\omega$ with $a \in \sigma(1)$. The *until* operator defines traces where a certain property φ_1 must hold until another property φ_2 is satisfied. For example, a model for the formula $a \,\mathcal{U}\, b$ defined of the set $AP = \{a, b\}$ is any trace $\sigma = \{a\}\{a\}\{a\}\{b\}\dots$. A trace $\sigma' = \{a\}\{a\}\{\}\{b\}$, on the other hand, violates the formula, as $a, b \notin \sigma'(2)$.

LTL also includes a list of derived operators. The *release* operator $\mathcal{R}$ is the dual operator to $\mathcal{U}$. The operator $\mathcal{R}$ states that the need to satisfying a certain property φ is released after another property φ_2 has been satisfied. A formula $\varphi_1 \,\mathcal{R}\, \varphi_2$ is equivalent to the formula $\neg\varphi_1 \,\mathcal{U}\, \neg\varphi_2$. For example, a model of the formula $a \,\mathcal{R}\, b$ is the infinite trace $\sigma = \{b\}\{b\}\{b\}\{a, b\}\dots$. The *eventually* operator $\diamondsuit$ states that a certain property must hold eventually. The trace $\sigma = \{a\}\{a\}\{a\}\{b\}\{a\}\dots$ as well as the trace $\{a\}\{a, b\}\{a\}\{a\}\dots$ satisfy the

formula $\Diamond b$. For an LTL formula φ, the formula $\Diamond \varphi$ is equivalent to the formula $\mathit{true}\,\mathcal{U}\,\varphi$. The dual operator to $\Diamond$ is the *always* operator $\Box$,which states that a property φ must hold on all positions of a trace. For an LTL formula φ, the formula $\Box \varphi$ is equivalent to the formula $\varphi\,\mathcal{R}\,\mathit{false}$.

Formally, the semantics of LTL is given as follows. For a set of atomic propositions AP, a word $\sigma \in (2^{AP})^\omega$ satisfies an LTL formula at position i based on the following rules:

$$
\begin{array}{lll}
\sigma, i \models a & \text{iff} & a \in \sigma(i) \\
\sigma, i \models \neg\varphi & \text{iff} & \sigma, i \not\models \varphi \\
\sigma, i \models \varphi_1 \vee \varphi_2 & \text{iff} & \sigma, i \models \varphi_1 \vee \sigma, i \models \varphi_2 \\
\sigma, i \models \bigcirc \varphi & \text{iff} & \sigma, i+1 \models \varphi \\
\sigma, i \models \varphi_1\,\mathcal{U}\,\varphi_2 & \text{iff} & \exists j \geq i.\, \sigma, j \models \varphi_2 \wedge \forall i \leq j' < j.\, \sigma, j' \models \varphi_1 \ .
\end{array}
$$

We say that an infinite trace $\sigma \in (2^{AP})^\omega$ satisfies an LTL formula φ, denoted by $\sigma \models \varphi$, if $\sigma, 0 \models \varphi$. Furthermore, we say that a transition system T satisfies an LTL formula φ, denoted by $T \models \varphi$, if for every $\sigma \in \mathit{Traces}(T)$ it holds that $\sigma \models \varphi$. The language of an LTL formula φ over AP, denoted by $L(\varphi)$, is defined by the set $\{\sigma \in (2^{AP})^\omega \mid \sigma \models \varphi\}$.

For every LTL formula φ we can build an alternating Büchi automaton A with $L(A) = L(\varphi)$. The size of A is linear in $|\varphi|$, where $|\varphi|$ denotes the length of φ. We show the construction in the next theorem. For the construction, we define the *closure* of an LTL formula φ by the set $\mathit{closure}(\varphi) = \{\psi, \neg\psi \mid \psi \text{ is a sub-formula of } \varphi\}$.

Theorem 2.6 (LTL to alternating automata [120, 150])**.** *For every* LTL *formula* φ *there is an alternating Büchi automaton* A *of size* $O(|\varphi|)$ *such that* $L(A) = L(\varphi)$.

Proof. Let φ be defined of a set of atomic propositions AP. We define the automaton $A = (\Sigma, Q, Q_0, \delta, F)$ as follows:

- $\Sigma = 2^{AP}$
- $Q = \mathit{closure}(\varphi)$
- $Q_0 = \{\varphi\}$
- $F = \{\neg(\varphi_1\,\mathcal{U}\,\varphi_2) \in \mathit{closure}(\varphi)\}$
- The transition relation δ is defined according the following rules

- $\delta(a, \alpha) = a \in \alpha$
- $\delta(\neg\varphi, \alpha) = \overline{\delta(\varphi, \alpha)}$
- $\delta(\varphi_1 \vee \varphi_2, \alpha) = \delta(\varphi_1) \vee \delta(\varphi_2)$
- $\delta(\bigcirc \varphi) = \varphi$
- $\delta(\varphi_1 \,\mathcal{U}\, \varphi_2, \alpha) = \delta(\varphi_2, \alpha) \vee (\delta(\varphi_1, \alpha) \wedge \varphi_1 \,\mathcal{U}\, \varphi_2)$.

□

Using the construction presented in Theorem 2.3, for an LTL formula φ, we can construct a nondeterministic Büchi automaton N of size $2^{O(|\varphi|)}$ with $L(N) = L(\varphi)$. Throughout the thesis we show that many problems can be solved more efficiently if the automaton representing an omega-regular property is unambiguous. In general, one can transform a nondeterministic Büchi automaton to an unambiguous Büchi automaton of exponential size [30]. In the next theorem, we show that, in the case of LTL, we can construct an unambiguous Büchi of size exponential in the size of the LTL formula directly without having to construct the nondeterministic Büchi automaton first. This can be done with a slight modification of the construction in Theorem 2.3 specific to LTL formulas [1].

Theorem 2.7 (LTL to unambiguous Büchi [13, 20, 50, 92, 151])**.** *For every* LTL *formula φ, there is an unambiguous Büchi automaton N of size $2^{O(|\varphi|)}$ such that $L(N) = L(\varphi)$.*

Proof. We present a translation that builds on the construction presented in Theorem 2.3, Theorem 2.6, and the translation of LTL to generalized Büchi automata [13]. Let φ be defined over a set of atomic propositions AP. We define $N = (2^{AP}, Q, Q_0, \delta, F)$ as follows

- $Q = \{(\Gamma, \Gamma') \in 2^{closure(\varphi)} \times 2^{closure(\varphi)} \mid \Gamma \text{ is elementary}\}$.
  We call a set $\Gamma \subseteq closure(\varphi)$ *elementary*, if it satisfies the following conditions
  - Γ is consistent to propositional logic. For all $\varphi', \varphi_1, \varphi_2 \in closure(\varphi)$
    - If $\varphi_1 \wedge \varphi_2 \in \Gamma$, then $\varphi_1 \in \Gamma$ and $\varphi_2 \in \Gamma$

[1] The same idea was used for the construction of Büchi automata for LTL formula via a construction of a nondeterministic generalized Büchi automaton for the formula, and translating this automaton to a nondeterministic Büchi automaton [13]. For presentation purposes, we modify the construction presented in Theorem 2.3 based on the same ideas.

  - If $\varphi' \in \Gamma$, then $\neg\varphi' \notin \Gamma$
  - If $true \in closure(\varphi)$, then $true \in \Gamma$
- Γ is locally consistent w.r.t. the until operator. For all $\varphi_1 \mathcal{U} \varphi_2 \in closure(\varphi)$ it holds that
  - If $\varphi_2 \in \Gamma$, then $\varphi_1 \mathcal{U} \varphi_2 \in \Gamma$
  - If $\varphi_1 \mathcal{U} \varphi_2 \in \Gamma$, and $\varphi_2 \notin \Gamma$, then $\varphi_1 \in \Gamma$
- Γ is maximal. For all $\varphi' \in closure(\varphi)$
  - If $\varphi' \notin \Gamma$, then $\neg\varphi' \in \Gamma$

- The definitions of Q_0, F and δ are given as in Theorem 2.3.

For a word σ, each position in the run of U on σ defines the maximal set of subformulas of φ satisfied in that position. If σ is accepted by U this run is unique for σ. □

2.5 Model Checking Omega-Regular Properties

Model checking omega-regular properties [43, 128, 13] describes the problem, where we check whether a transition system T satisfies an omega-regular property φ, i.e., $Traces(T) \subseteq \varphi$.

We recap the automata-based model checking approach, where given a nondeterministic Büchi automaton the algorithm checks whether any trace of the transition system is not the language defined by the automaton [13].

Let φ be an omega-regular property, and let $N_{\overline{\varphi}} = (2^{AP}, Q, Q_0, \delta, F)$ be the nondeterministic Büchi automaton for the complement property $\overline{\varphi}$. To check whether a transitions system $T = (AP, I, O, S, s_0, \tau, L)$ satisfies the property φ, we check the emptiness of the following nondeterministic Büchi automaton $B_\otimes$ which defines the intersection of the languages $L(N_{\overline{\varphi}}) \cap Traces(T)$. The automaton $B_\otimes = (2^{AP}, Q_\otimes, Q_{0,\otimes}, \delta_\otimes, F_\otimes)$ is defined as follows:

- $Q_\otimes = Q \times S$
- $Q_{0,\otimes} = Q_0 \times \{s_0\}$
- $F_\otimes = F \times S$
- $\delta_\otimes = \{((q,s), \alpha, (q',s')) \mid q' \in \delta(q,\alpha),\ \alpha_O = L(s),\ s' = \tau(s,\alpha_I)\}$.

If the automaton is not empty, i.e., $L(B_\otimes) \neq \emptyset$, then there is a word shared between $Traces(T)$ and $N_{\overline{\varphi}}$. Thus, there is a trace in T that violates φ.

Theorem 2.8 (LTL Model Checking [13, 151])**.** *For an omega-regular property, represented by an* LTL *formula φ, and for a transition system T, the model checking problem for T and φ can be solved in time exponential in $|\varphi|$ and polynomial in T.*

Proof. Given an LTL formula φ we can compute the nondeterministic automaton for the negated formula $\neg\varphi$ and use the model checking algorithm described above. The nondeterministic automaton is of exponential size in the length of φ, and thus the product automaton is of size exponential in the length of φ and polynomial in the size of T. The emptiness check can be done in time polynomial in the size of the product automaton using a nested depth first search algorithm [89]. □

Instead of constructing the full automaton for the negated LTL formula and computing the product automaton with the transition system, we also may guess an accepting run of the product automaton on the fly by guessing the run state by state. Guessing a state can be done by guessing a state of the transition system and an elementary subset of the closure of the LTL formula as defined in Theorem 2.7. This results in the following complexity for the LTL model checking problem.

Theorem 2.9 ([13, 151])**.** *The problem of model checking a transition system T against an* LTL *formula φ is* PSPACE*-complete.*

Proof. In the following we give a polynomial-space algorithm for model checking a transition system T against an LTL formula φ based on the construction presented in Theorem 2.7. For the proof of the lower bound we refer the reader to [13, 138].

The idea of the algorithm is to nondeterministically guess a lasso run in the product automaton of $N_{\neg\varphi}$ (the nondeterministic Büchi automaton for $\neg\varphi$), and the transition system T. This can be done by guessing the following $r = (q_0, s_0)(q_1, s_1)\dots(q_{i-1}, s_{i-1})(q_i, s_i)\dots(q_{k-1}, s_{k-1})$ where $k = |T| \cdot 2^{|closure(\varphi)|}$ defining the maximal size of the product automaton of $N_{\neg\varphi}$ and T. The algorithm then checks whether the run

$$(q_0, s_0)(q_1, s_1)\dots(q_{i-1}, s_{i-1})((q_i, s_i)\dots(q_k, s_k))^\omega$$

is an accepting run in the product automaton by checking whether there is $i \leq j \leq k-1$ such that q_j is an accepting state in $N_{\neg\varphi}$.

Using the construction in Theorem 2.7 to construct the automaton $N_{\overline{\varphi}}$, the states $q_0, \ldots, q_{k-1}$ are subsets from $\mathit{closure}(\varphi)$, and can thus be guessed by guessing such subsets. After each guess, the algorithm checks whether the new state satisfies both the transition relation of T as well as the conditions of the transition relation of $N_{\neg\varphi}$ as defined in Theorem 2.7. Furthermore, at the beginning of the process, the algorithm guesses the number i, representing the beginning of the loop of the lasso, and checks at the end of the guessed word whether it defines a loop in the automaton by checking whether there is a transition from (q_{k-1}, s_{k-1}) to (q_i, s_i). Along guessing the word, the algorithm keeps track of whether an accepting state has been guessed from the point of reaching the loop position i.

The algorithm only needs to store the number i, by storing $\lceil \log k \rceil$ bits, the current guessed state, and the loop state, which all requires space in $O(\log |T| \cdot |\varphi|)$. □

Chapter 3

Model Counting Algorithms for Omega-regular Properties

In this chapter, we introduce four model counting problems for linear-time properties, and present algorithms for solving these problems for the prominent subclass of omega-regular properties. We provide algorithms for omega-regular properties given as deterministic parity automata, nondeterministic parity automata, and LTL formulas. In the following, we briefly describe each of the four problems

- **The trace counting problem:** We start with the general problem of counting infinite traces of a transition system. Given a transition system T and a linear-time property φ, we want to compute the number of traces in T that satisfy φ. Due to the infinite length of traces, the number of traces satisfying a property may reach infinity. For omega-regular languages, we show that if the number of traces is less than infinity, then it cannot exceed a certain bound depending on the representation. In this case we can use specialized algorithms for solving the trace counting problem.

- **The bounded bad prefix counting problem:** Given a transition system T, a linear-time property φ, and a bound $n \in \mathbb{N}$, the bad prefix counting problem is the problem of computing the number of bad prefixes of φ of length n in T.

- **The bounded good prefix counting problem:** Given a transition system T, a linear-time property φ, and a bound $n \in \mathbb{N}$, the good prefix counting problem is the dual problem for the bad prefix counting problem, and asks for computing the number of good prefixes of φ of length n in T.

- **The bounded lasso counting problem:** Given a transition system T, a linear-time property φ, and a bound $n \in \mathbb{N}$, the lasso counting problem is the problem where we are interested in computing the number of lassos of size n that induce traces in T that satisfy φ.

Table 3.1 gives a summary on the run-time and space requirements of our algorithms for different representations of omega-regular properties. For the *bounded counting problems*, i.e., the counting problems for bad prefixes, good prefixes, and lassos, there is sometimes a tradeoff between the complexities in the size of the representation and in the bound.

In addition to the algorithmic complexity of each problem, we also provide a complete complexity analysis in terms of counting complexity classes, providing lower and upper bounds for each counting problem. An overview of the different counting complexity classes is given next.

3.1 Counting Complexity

Counting complexity classes are based on counting the accepting runs of nondeterministic Turing machines. The first counting complexity class, #P, was introduced by Valiant in his seminal paper on the complexity of computing the permanent [147]. Valiant defined #P as the class of all counting problems associated with decision problems in the class NP. Formally, a function $f\colon \Sigma^* \to \mathbb{N}$, for some alphabet Σ, is in #P if there is a nondeterministic polynomial time Turing machine M such that, for any word $w \in \Sigma^*$, $f(w)$ is equal to the number of accepting runs of M on w. The class #P includes many prominent counting problems including the canonical model counting problem for propositional formulas #SAT. With respect to the counting problems we introduce in this chapter, we show that the class #P does not capture the complexity of some of these problems, as some counting problems are associated with decisional problems in classes less complex than, or beyond the class NP.

Analogously to the class #P, given a decisional complexity class C, we define the class #C by the class of counting problems associated with decisional problems in the class C. For example, the class #L defines the set of functions $f\colon \Sigma^* \to \mathbb{N}$, for which there is nondeterministic logarithmic-space Turing

DPA D	Time complexity	Space complexity
Traces	$O(poly(\|D\|))$	$O(poly(\|D\|))$
Bad prefixes (n)	$O(\|D\| \cdot n)$	$O(\|D\| \cdot n)$
	$\|D\| \cdot 2^{O(n)}$	$O(\log(\|D\|) + n)$
Good prefixes (n)	$O(\|D\| \cdot n)$	$O(\|D\| \cdot n)$
	$\|D\| \cdot 2^{O(n)}$	$O(\log(\|D\|) + n)$
Lassos (n)	$2^{O(\|D\|)} \cdot n$	$2^{O(\|D\|)} \cdot n$
	$\|D\| \cdot 2^{O(n)}$	$O(\|D\| + n)$
NPA N	Time complexity	Space complexity
Traces	$2^{O(poly(\|N\|))}$	$O(poly((\|N\|))$
Bad prefixes (n)	$2^{O(poly(\|N\|))} \cdot n$	$2^{O(poly(\|N\|))} \cdot n$
	$\|N\| \cdot 2^{O(n)}$	$O(\log(\|N\|) + n)$
Good prefixes (n)	$2^{O(poly(\|N\|))} \cdot n$	$2^{O(poly(\|N\|))} \cdot n$
	$2^{O(poly(\|N\|)+n)}$	$O(poly(\|N\|) + n)$
Lassos (n)	$2^{2^{O(poly(\|N\|))}} \cdot n$	$2^{2^{O(poly(\|N\|))}} \cdot n$
	$\|N\| \cdot 2^{O(n)}$	$O(\|N\| + n)$
LTL φ	Time complexity	Space complexity
Traces	$2^{O(poly(\|\varphi\|))}$	$2^{O(poly(\|\varphi\|))}$
Bad prefixes (n)	$2^{2^{O(\|\varphi\|)}} \cdot n$	$2^{2^{O(\|\varphi\|)}} \cdot n$
	$2^{O(\|\varphi\| \cdot n)}$	$O(n + poly(\|\varphi\|))$
Good prefixes (n)	$2^{2^{O(poly(\|\varphi\|))}} \cdot n$	$2^{2^{O(poly(\|\varphi\|))}} \cdot n$
	$2^{O(\|\varphi\| \cdot n)}$	$O(poly(\|\varphi\|) + n)$
Lassos (n)	$2^{2^{O(poly(\|\varphi\|))}} \cdot n$	$2^{2^{O(poly(\|\varphi\|))}} \cdot n$
	$poly(\|\varphi\|) \cdot 2^{O(n)}$	$O(poly(\|\varphi\|) + n)$

Table 3.1: Time and space complexity of the different model counting algorithms for omega-regular properties represented by deterministic parity automaton (DPA), non-deterministic parity automata (NPA), and LTL formulas. The bound n represents the length of the bad prefix, good prefix, or the size of the lasso. For the bounded problems, the table distinguishes, using separate rows, two algorithms with different complexities in the bound. The first row shows the complexity for a propagation-based algorithm. The second row for an algorithm that guesses and checks the bounded representation.

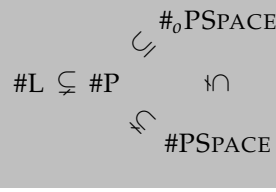
Repres.	Traces	Bad prefixes	Good prefixes	Lassos
DPA	#L-comp.	#L-comp.	#L-comp.	#L/#P
NPA	#P-comp.	#P-comp.	#P/#$_o$PSPACE	#P-comp.
LTL	#PSPACE-comp.	#P/#$_o$PSPACE	#P/#$_o$PSPACE	#P-comp.

Table 3.2: The complexities of counting problems for properties given by a deterministic or nondeterministic parity automaton, or by an LTL formula.

machine such that $f(w)$ is equal to the number of accepting runs of M on w. This stands in contrast to the interpretation proposed by Valiant, where #C denotes the class of functions counting the accepting paths of a nondeterministic polynomial-time Turing machine with an oracle in the class C. To distinguish the two definitions, we will refer to the oracle counting classes suggested by Valiant by #$_oC$. For example, the class #$_o$PSPACE will define counting problems associated with a nondeterministic polynomial-time Turing machine with access to an oracle in the class PSPACE.

In the next section, we will determine the complexities of our counting problems with respect to the counting complexity classes above. Table 3.2 gives a summary on the complexities of all problems studied in this chapter. We use parsimonious reductions to define hardness and completeness, i.e., the most restrictive notion of reduction for counting problems [124]. A function f is #P-hard, if for every $f' \in$ #P there is a polynomial-time computable function r such that $f'(x) = f(r(x))$ for all inputs x. In particular, if f' is induced by counting the accepting runs of M, then r depends on M (and possibly on its time-bound $p(n)$). Furthermore, f is #P-complete, if f is #P-hard and $f \in$ #P. Hardness and completeness for the other classes are defined analogously.

Excursion 3.1 (Relations between counting complexity classes)**.** We use this excursion to familiarize the reader with relations between the different counting classes. In the following, we give a partial inclusion hierarchy on the counting classes used in this thesis. For more details on the hierarchy of other counting complexity classes, we refer the reader to [84, 145].



Intuitively, the strict inclusions follow from the fact that the number of runs of a nondeterministic logarithmic-space Turing machine is bounded from above by a polynomial in the size of the Turing machine, and the number of runs of a nondeterministic polynomial-time Turing machine is bounded by an exponential in the size of the Turing machine. This means that the set of functions computable by Turing machines in #P is strictly larger than the set given by #L, and that the set of functions computable by Turing machines in #PSPACE is strictly larger than the set given by #P, or $\#_o$PSPACE.

3.2 Counting Infinite Traces

The number of traces in a transition system T that satisfy or violate a linear-time property may be infinitely many. Consider for example the transition system depicted in Figure 3.1. The transition system has one trace that violates the property $\Box\Diamond p$, namely the one produced by the input sequence i^ω, and infinitely many traces that satisfy the property, namely for all other inputs sequences besides i^ω.

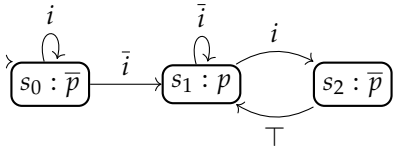


Figure 3.1: A transition system with one trace violating and infinitely many traces satisfying the property $\Box\Diamond p$.

Counting the number of traces in a transition system that satisfy a linear-time property using enumeration is thus not feasible. For an ω-regular language represented by a nondeterministic parity automaton N, we can determine the number of traces in a transition system T by analyzing the graph structure of the product automaton $N \otimes T$. This is done in two steps. First we need to determine whether the product automaton accepts infinitely many words, by checking the automaton for so-called *doubly-pumped lassos*, which we will define below. If the automaton does not contain any doubly-pumped lasso, then we can compute the number of traces in T that are in the language of N by computing the number of minimal paths in the product automaton that lead to an accepting loop.

3.2.1 Doubly-pumped lassos

A doubly-pumped lasso in a parity automaton is a subautomaton composed of two lassos, where the loop of one lasso is reachable from a state on the other lasso. Formally, we define doubly-pumped lassos as follows.

Definition 3.1 (Doubly-pumped lassos in parity automata)**.** A doubly-pumped lasso in a nondeterministic parity automaton $P = (\Sigma, Q, Q_0, \delta, \mu)$ is defined by a tuple $\rho = (u, v, u', v') \in (Q \times \Sigma)^* \times (Q \times \Sigma)^+ \times (Q \times \Sigma)^* \times (Q \times \Sigma)^+$ such that

- (u, v) and (u', v') are lassos in P, and
- there is $w \in (Q \times \Sigma)^+$ with $|u| < |w| \leq |u \cdot v|$, $w \leq u \cdot v$, and $w \leq u' \cdot v'$.

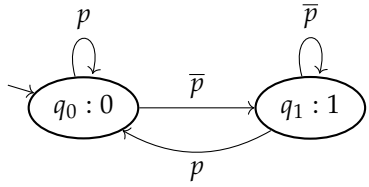
The doubly-pumped lasso ρ is called initial, if $(u \cdot v)(0) = (q_0, \alpha)$ for some $\alpha \in \Sigma$. It is called accepting, if $\max\{\mu(q_i) \mid 0 \leq i < |v'|, v'(i) = (q_i, \alpha_i)\}$ is even.

Figure 3.2 shows example doubly-pumped lassos in the product automaton of the deterministic parity automaton for $\square\diamondsuit p$, shown at the top of the figure, and the transition system depicted in Figure 3.1. The doubly-pumped lasso on the left can be given by the tuple (u_1, v_1, u_2, v_2), where $u_1 = ((s_0, q_0), \{i\})$, $v_1 = ((s_0, q_1), \{i\})$, $u_2 = ((s_0, q_0), \{i\}) \cdot ((s_0, q_1), \emptyset) \cdot ((s_1, q_1), \{p\})$, and $v_2 = ((s_1, q_0), \{p\})$. The doubly-pumped lasso on the right is defined by the tuple (u_1, v_1, u_2, v_2), where $u_1 = ((s_0, q_0), \emptyset) \cdot ((s_1, q_1), \{p\})$, $v_1 = ((s_1, q_0), \{p\})$, $u_2 = ((s_0, q_0), \emptyset) \cdot ((s_1, q_1), \{p\})$, and $v_2 = ((s_1, q_0), \{i\}) \cdot ((s_2, q_1), \{p, i\})$.

In the next lemma, we show that the number of traces in a transition system T that are models of a parity automaton P is infinite, if and only if, the product automaton $T \otimes P$ has an initial accepting doubly-pumped lasso, composed of two lassos that differ in their traces. If such a doubly-pumped lasso cannot be found, we can then compute the number of traces accepted by P using an algorithm that runs in time polynomial in $|T|$ and $|P|$, when P is unambiguous.

***Remark* 3.1.** *In all the following lemmas and theorems in this chapter, we state our results in terms of non-deterministic, or deterministic automata. All results can be carried to transition systems by building the product automaton of the transition system and the given automaton, which results in a new parity automaton. The product automaton preserves the determinism of the given automaton, because all transition systems that we consider are deterministic.*

▷ Parity automaton for $\square\diamond p$:



▷Two doubly-pumped lassos in $T \otimes N$:

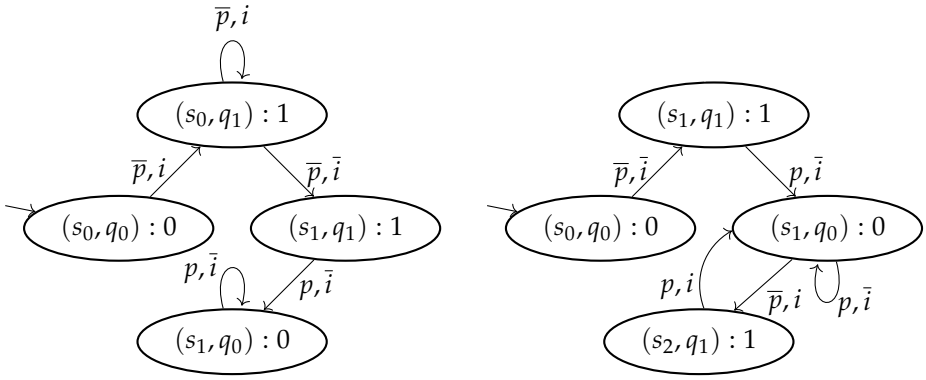


Figure 3.2: A parity automaton P for $\square\diamond p$, and two example doubly-pumped lassos in the product automaton of the transition system T given in Figure 3.1 and P.

Lemma 3.1. *Let $N = (\Sigma, Q, Q_0, \delta, \mu)$ be a nondeterministic parity automaton. The number of words in $L(N)$ is equal to infinity if and only if N has an initial accepting doubly-pumped lasso (u_1, v_1, u_2, v_2) with $trace(u_1 \cdot (v_1)^\omega) \neq trace(u_2 \cdot (v_2)^\omega)$.*

Proof. If N has an initial accepting doubly-pumped lasso (u_1, v_1, u_2, v_2), where $trace(u_1 \cdot (v_1)^\omega) \neq trace(u_2 \cdot (v_2)^\omega)$, then the language of the doubly-pumped lasso includes every infinite word with a run in N that has a prefix u_1, concatenated with an infix $(v_1)^n \cdot w$, and a suffix v_2^ω, where $n \in \mathbb{N}$, and $u_1 \cdot w \cdot v_2^\omega = u_2 \cdot v_2^\omega$ (the word w is the segment connecting the state reached by u_1 to the first states of the loop v_2). Since $trace(u_1 \cdot (v_1)^\omega) \neq trace(u_2 \cdot (v_2)^\omega)$, every $n \in \mathbb{N}$ distinguishes a unique word that is accepted by the automaton N (v_2 is accepting), and thus N accepts an infinite number of words over Σ.

To show that the existence of a doubly-pumped lasso (u_1, v_1, u_2, v_2) with

$trace(u_1 \cdot (v_1)^\omega) \neq trace(u_2 \cdot (v_2)^\omega)$ is a necessary condition for $L(N) = \infty$, we use the following argumentation based on the pigeonhole principle. If N accepts infinitely many words, then each word has a run in N that has a prefix $u \cdot v$ such that $\rho = (u, v)$ is a lasso in N. Since N has a finite number of states, and a finite number of transitions, by the pigeonhole principle, there are two lassos $\rho = (u, v)$ and $\rho' = (u', v')$ in N with $trace(u \cdot v^\omega) \neq trace(u' \cdot v'^\omega)$ such that there is a state shared between v and $u \cdot v'$. This means that (u, v, u', v') is a doubly-pumped lasso in N that is initial, accepting, and where $trace(u \cdot v^\omega) \neq trace(u' \cdot v'^\omega)$.

A more technical proof is provided for the reader in Section 3.7. □

We present an algorithm for finding a doubly-pumped lasso (u, v, u', v') in a nondeterministic parity automaton N that runs in time polynomial in the size of N. To avoid enumerating all lassos in N, the algorithm first constructs the self-composition $N_\otimes$ of N, where each run in $N_\otimes$ represents two runs in N, and thus, a lasso in $N_\otimes$ represents two lassos in N. Our algorithms checks whether $N_\otimes$ has a lasso that represents a doubly-pumped lasso in N.

The self-composition $N_\otimes$ is defined as follows. Let $N = (\Sigma, Q, Q_0, \delta, \mu)$. A self-composition of N is an automaton $N_\otimes = (\Sigma_\otimes, Q_\otimes, Q_{0,\otimes}, \delta_\otimes, \mu_\otimes)$, where

- $\Sigma_\otimes = \Sigma \times \Sigma$
- $Q_\otimes = Q \times Q$
- $Q_{0,\otimes} = Q_0 \times Q_0$
- $\delta_\otimes = \{((q_1, q_2), (\alpha_1, \alpha_2), (q_1', q_2')) \mid q_1' \in \delta(q_1, \alpha_1),\ q_2' \in \delta(q_2, \alpha_2)\}$,
- $\mu_\otimes = \{((q_1, q_2), \mu(q_2)) \mid (q_1, q_2) \in Q_\otimes\}$

Let $\rho = (u, v)$ be an initial lasso in the automaton $N_\otimes$, with $u = (q_0, q_0)(\alpha_1^1, \alpha_1^2) \cdot (q_1^1, q_1^2)(\alpha_2^1, \alpha_2^2) \dots (q_{j-1}^1, q_{j-1}^2)(\alpha_{j-1}^1, \alpha_{j-1}^2)$, and $v = (q_j^1, q_j^2)(\alpha_{j+1}^1, \alpha_{j+1}^2) \dots (q_k^1, q_k^2)(\alpha_{k+1}^1, \alpha_{k+1}^2)$, for some $j, k \in \mathbb{N}$ with $j \leq k$. The lasso ρ defines two lassos ρ_1 and ρ_2 in N. The lasso ρ_1 is defined by (u_1, v_1), where $u_1 = q_0\alpha_1^1 \cdot q_1^1\alpha_2^1 \dots q_{j-1}^1\alpha_{j-1}^1$ and $v_1 = q_j^1\alpha_{j+1}^1 \dots q_k^1\alpha_{k+1}^1$. The lasso ρ_2 is defined by (u_2, v_2), where $u_2 = q_0\alpha_1^2 \cdot q_1^2\alpha_2^2 \dots q_{j-1}^2\alpha_{j-1}^2$, and $v_2 = q_j^2\alpha_{j+1}^2 \dots q_k^2\alpha_{k+1}^2$. To check whether the lassos ρ_1 and ρ_2 form an initial accepting doubly-pumped lasso in N, the lasso ρ must satisfy the following conditions:

1. $\exists h \leq j.\ q_j^1 = q_h^2$: This condition assures that ρ_1 and ρ_2 form a doubly-pumped lasso in the graph of N. If $q_j^1 = q_h^2$, then the period of ρ_1 shares a

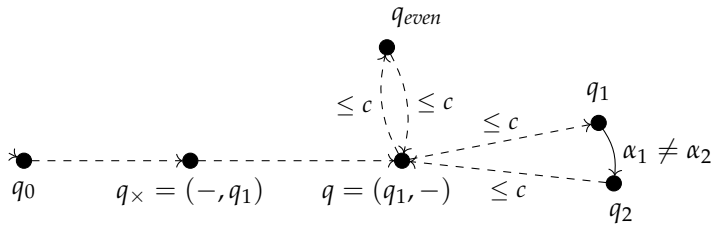


Figure 3.3: An illustration of algorithm detectDPL. The state q_0 is an initial state in $N_\otimes$. The dashed lines represent a path in $N_\otimes$ leading from one state to another. The solid line between q_1 and q_2 represents a transition (α_1, α_2) with $\alpha_1 \neq \alpha_2$. The lines annotated with $\leq c$ represent paths in $N_\otimes$, where the highest color is less or equal to c.

state with ρ_2. This implies that there is a word w with $|u_1| < |w| \leq |u_1 \cdot v_1|$ that is a prefix for ρ_1 and ρ_2.

2. ρ must be accepting, i.e., the lasso ρ_2 is an accepting lasso in N.

3. $\exists h > j.\ \alpha_h \neq \alpha'_h$: The lassos differ in a letter in a position after the shared state and thus $trace(\rho_1) \neq trace(\rho_2)$.

Checking whether $N_\otimes$ has a lasso ρ that fulfills conditions 1-3 can be done in polynomial time in the size of $N_\otimes$ using the procedure presented in Algorithm 3.1.

The idea of Algorithm 3.1 is illustrated in Figure 3.3. The algorithm checks if $N_\otimes$ has two loops that start in the same state $q = (q_1, q_2)$ (line 5). One loop that satisfies condition 2 (lines 16 -21), and another loop that satisfies condition 3 (lines 23 - 28), and where the highest color seen in the loop is not larger than the highest color seen in the first loop. For verifying condition 2, we check whether q can reach itself via a loop where the highest color is an even color c. We do that by checking whether q reaches itself via a state q_{even} with color c, and where no higher color than c is observed along the way from q to q_{even} and back. The reachability check is performed using the procedure "Reachable", which checks for two states q and q', and a color c, whether there is path in $N_\otimes$ from q to q' where the highest color seen along the path is less or equal to c. If we find a loop that fulfills condition 2, we continue with checking condition 3 by looking for a loop from q with highest color c, and with one transition (α_1, α_2), where $\alpha_1 \neq \alpha_2$.

Since both loops that satisfy condition 2 and condition 3 start in the same state, they form one big loop in $N_\otimes$, that satisfies both conditions. Assuming that all states in $N_\otimes$ are reachable from the initial state, the big loop then represents

Algorithm 3.1 Detecting Doubly-pumped Lassos (detectDPL)

Input: $N = (\Sigma, Q, Q_0, \delta, \mu)$,
output: N has an accepting doubly-pumped lasso starting in $q_0 \in Q_0$
Begin:
$b_1, b_2, b_3 := \mathit{false}$
for $q := (q_1, q_2) \in Q_\otimes$ **do**

    $c := \max\{\mu_\otimes(q) | q \in Q_\otimes\}$
    for $q_\times \in \{(q', q'') \in Q_\otimes \mid q'' = q_1\}$ **do**
        if Reachable$(q_\times, c, q)$ **then**
            $b_1 := \mathit{true}$
            break
    if not b_1 **then**
        break

    for $c := 0$; $c \leq \max\{\mu_\otimes(q) | q \in Q_\otimes\}$; $c := c + 2$ **do**
        for $q_{even} \in \{q \mid \mu_\otimes(q) = c\}$ **do**
            if Reachable$(q, c, q_{even}) \wedge$ Reachable(q_{even}, c, q) **then**
                $b_2 := \mathit{true}$
            break
        if not b_2 **then**
            break

        for $(q_1, \alpha, q_2) \in \{(q, (\alpha_1, \alpha_2), q') \mid q' \in \delta(q, (\alpha_1, \alpha_2)) \wedge \alpha_1 \neq \alpha_2\}$ **do**
            if Reachable$(q, c, q_1) \wedge$ Reachable(q_1, c, q) **then**
                $b_3 := \mathit{true}$
                break
        if not b_3 **then**
            break

    if $b_1 \wedge b_2 \wedge b_3$ **then**
        return *true*
return *false*
End.

the period of a lasso ρ in $N_\otimes$ reached by a stem from an initial state of $N_\otimes$ to q. To check that ρ represents a doubly-pumped lasso in N, we further need to check that ρ satisfies condition 1 (lines 7 - 13).

To check condition 1, we need to check whether there is a state $q_\times = (q', q'')$ with $q'' = q_1$ and such that q is reachable from $q_\times$. This is done by calling a procedure Reachable($q_\times, c, q$). Since $q_\times$ is on the stem of the lasso we are looking for, the highest color of the system is irrelevant for the acceptance of the lasso, and can thus be any color (line 7).

If the algorithm fails to satisfy one the conditions for q, it repeats the process for a new state from $Q_\otimes$. The runtime of Algorithm 3.1, as we state in the next theorem, is polynomial in the size of N.

Theorem 3.1. *Let N be a nondeterministic parity automaton. The problem of deciding whether $|L(N)| = \infty$ can be solved in time polynomial in the size of N.*

Proof. In Algorithm 3.1, each reachability problem can be solved in polynomial time using using the same procedure for checking reachability in graphs [111], and additionally tracking the color c observed along the search. The size of $N_\otimes$ is polynomial in N, and therefore, the overall procedure is also polynomial in N.

□

The algorithm above needs space logarithmic in the size of the automaton $N_\otimes$.

Corollary 3.1. *Let N be a nondeterministic parity automaton. Deciding whether $|L(N)| = \infty$ can be done in space logarithmic in the size of N.*

Proof. In Algorithm 3.1, for each iteration of the for-loop in line 7, we only need to store the value of the state q, and for checking the conditions 1, 2, and 3, we only need to keep the values of the states $q_\times$, q_{even}, q_1, q_2, the value of the letter α, and the color c. Every state can be encoded by $\log(|N_\otimes|)$ many bits. The color c can also be encoded by $\log(|N_\otimes|)$ bits, because the maximum number of different colors in $N_\otimes$ is equal to the size of $N_\otimes$. The procedure Reachable(q, c, q') only needs to maintain the highest color seen along the exploration, and thus has the same complexity of the state-reachability algorithm in graphs, which needs logarithmic space in the size of $N_\otimes$ [111]. □

When a linear-time property is given by an LTL formula φ, checking whether there are infinitely many traces in a transition system T that satisfy the LTL formula can be done by checking whether an automaton defining the product of a nondeterministic Büchi automaton B with $L(B) = \varphi$ and T has a doubly-pumped lasso satisfying the conditions 1,2, and 3 above. The doubly-pumped lasso in the product automaton can be guessed on the fly, without having to construct the Büchi automaton B, by guessing the right subsets from $2^{closure(\varphi)}$, which constitute the sets of the Büchi automaton as we have presented in Theorem 2.7. Detecting such a doubly-pumped lasso can thus be done in space polynomial in the length of φ, and logarithmic in the size of T.

Theorem 3.2. *Let φ be an* LTL *formula. Deciding whether $|L(\varphi)| = \infty$ can be done in space polynomial and time exponential in $|\varphi|$.*

Proof. We apply a similar approach as for the polynomial-space algorithm in Theorem 2.9 for model checking LTL formulas. For every LTL formula φ defined over a set of atomic propositions AP, we can build a nondeterministic Büchi automaton $N = (\Sigma, Q, Q_0, \delta, F)$ of size $|N| = 2^{O(|\varphi|)}$ with $L(N) = L(\varphi)$ as we presented in Theorem 2.7. The sets of N are defined by the set of subsets of $closure(\varphi)$. A state in N can thus be guessed by guessing a subset of $closure(\varphi)$ and checking whether the set satisfies the consistency rules defined in Theorem 2.7.

Following the approach presented in Corollary 3.1, we guess states $q = (q_1, q_2) \in Q \times Q$, $q_\times = (q', q_1)$ for some $q' \in Q$, $q_{even} = (q^1_{even}, q^2_{even})$ with $q^2_{even} \in F$, $q_1, q_2 \in Q \times Q$, and a letter $\alpha = (\alpha_1, \alpha_2) \in \Sigma \times \Sigma$ with $\alpha_1 \neq \alpha_2$, and check whether q is reachable from $q_\times$, q reaches itself via q_{even}, q_1 is reachable from q, q is reachable from q_2, and q_2 is reachable from q_1 via a transition with α. This can be done by guessing runs in the automaton N, state by state and checking whether they satisfy the transitions of N as defined in Theorem 2.7.

Storing each guessed state requires polynomial space in $|\varphi|$, and we only need to store a constant number of these sets. □

Corollary 3.2. *Checking whether a transition system T has infinitely many traces that satisfy an* LTL *formula φ can be done in space polynomial in $|\varphi|$ and logarithmic in T.*

Proof. This can be done by expanding the procedure in the last theorem to one that additionally, for each guessed state in N, guesses a state in T and checks that it satisfies the transition relation of T. □

The number of models of an omega-regular language φ is either infinity, or bounded exponentially by the size of any nondeterministic parity automaton N, with $L(N) = \varphi$. The number of models for an LTL formula is then bounded by a double exponential in the length of the formula, considering the exponential blow-up when translating the LTL formula into a nondeterministic automaton. We prove the claims in the next theorem.

Theorem 3.3. *For a parity automaton P, if $|L(P)| < \infty$, then $|L(P)| \leq 2^{|P| \cdot \log(|P|)}$.*

Proof. Let $P = (\Sigma, Q, Q_0, \delta, \mu)$. We show that if $|L(P)| > 2^{|P| \cdot \log(|P|)}$, then $|L(P)| = \infty$. Assume that $|L(P)| = k = 2^{|P| \cdot \log(|P|)} + 1$, and let $\sigma_1, \ldots, \sigma_k \in L(P)$ be all models of $L(P)$. For $1 \leq \ell \leq k$, let $\sigma_\ell = \alpha_1^\ell \cdot \alpha_2^\ell \cdots$, and let $r^\ell = (q_0^\ell, \alpha_1^\ell) \cdot (q_1^\ell, \alpha_2^\ell) \cdot (q_2^\ell, \alpha_3^\ell) \cdots$ be a run of P over σ. By the pigeonhole principle, we know that there is $0 \leq i < |P|$, and $i < j \leq |P| + 1$ such that $q_i^\ell = q_j^\ell$, and thus that there is a lasso $\rho^\ell = (u^\ell, v^\ell)$ in P with $u^\ell = (q_0^\ell, \alpha_1^\ell) \cdots (q_{i-1}^\ell, \alpha_i^\ell)$ and $v^\ell = (q_i^\ell, \alpha_{i+1}^\ell) \cdots (q_{j-1}^\ell, \alpha_j^\ell)$. We distinguish two cases:

- Assume that for some $1 \leq \ell \leq k$ it holds that $\mathit{trace}(u^\ell \cdot (v^\ell)^\omega) \neq \mathit{trace}(r^\ell)$. This implies that for a number of repetitions n, we have that $\mathit{trace}(u^\ell \cdot (v^\ell)^n) \neq \mathit{trace}(r^\ell[\ldots |u^\ell| + |v^\ell|^n])$, and that $(u^\ell \cdot (v^\ell)^n)(h-1) \neq r^\ell(h-1)$ for some $|u^\ell| + |v^\ell| < h \leq |u^\ell| + |v^\ell|^n$. Since r is accepting, then also by the pigeonhole principle there must be two positions $h \leq i < j$ such that $q_i = q_j$ and $\mu(q_j)$ is even and the highest color observed along r. It follows that P has an initial accepting doubly-pumped lasso (u^ℓ, v^ℓ, u', v') where $u' = r^\ell[\ldots i-1]$ and $v' = r^\ell[i \ldots j]$, and with $\mathit{trace}(u^\ell \cdot (v^\ell)^\omega) \neq \mathit{trace}(u' \cdot (v')^\omega)$. According to Lemma 3.1 the automaton P has infinitely many models.

- Assume for all $1 \leq \ell \leq k$ it holds that $\mathit{trace}(u^\ell \cdot (v^\ell)^\omega) = \mathit{trace}(r)$. By the pigeonhole principle, there must be $1 \leq \ell \neq \ell' \leq k$ such that v^ℓ shares a state with $u^{\ell'} \cdot v^{\ell'}$. This implies that the automaton P has an initial accepting doubly-pumped lasso $(u^\ell, v^\ell, u^{\ell'}, v^{\ell'})$. Since all $\sigma_1, \ldots, \sigma_k$ are different, it follows that $\mathit{trace}(u^\ell \cdot (v^\ell)^\omega) \neq \mathit{trace}(u^{\ell'} \cdot (v^{\ell'})^\omega)$, and thus P has infinitely many models.

□

Corollary 3.3. *For an* LTL *formula φ, $|L(\varphi)| < \infty$, then $|L(\varphi)| \leq 2^{2^{O(|\varphi|)} \cdot \log(|\varphi|)}$.*

In the next theorem we show that there are LTL formulas with doubly-exponential many words. The proof is given in Section 3.7.

Theorem 3.4. *There is a family of* LTL *formulas* φ_n *for* $n \in \mathbb{N}$ *with* $|\varphi_n| \in O(n)$ *and* $|L(\varphi_n)| \in 2^{2^{O(n)} \cdot \log(n)}$.

3.2.2 Algorithms for counting infinite traces

From Theorem 3.3, we know that if the number of models of a parity automaton P is bounded, then P has a unique accepting lasso for each model $\sigma \in L(P)$ of size at most $|P|$. To compute the number of models of P we can enumerate the accepting lassos in P up to size P, which may require time exponential in the size of $|P|$.

In the following we show that if P is unambiguous, we can compute the number of its models in time only polynomial in the size of P. The counting procedure is presented in Algorithm 3.2, which is based on the following idea. Since P is unambiguous, we know that any word accepted by P has exactly one run in P which, after at most $|P|$ steps, reaches an accepting loop in P. Since P has no doubly-pumped lassos, the number of words accepted by P is equal to the number of finite paths of length $|P|$ that reach an accepting loop in P.

The algorithm starts by marking every state that lies on an accepting loop in P with the value 1. After marking the states, the algorithm enters the following loop. In each iteration i of the loop, the algorithm marks each state q in the automaton by the number of finite paths of length i that reach an accepting loop from q. This is computed by adding the values of all the successor states of q. Since any accepting loop is reached within at most $|P|$ steps from the initial state, the algorithm iterates $|P|$ times. The sum of values mapped to the initial states defines then the number of initial paths leading to an accepting loop in P, which is equal to the number of words accepted by P.

Theorem 3.5. *Let* P *be an unambiguous parity automaton, and let* $|L(P)| < \infty$. *Computing the number of models of* $L(P)$ *can be done in time polynomial in the size of* P.

Proof. Let $P = (\Sigma, Q, Q_0, \delta, \mu)$. The correctness of Algorithm 3.2 can be shown by proving the following strengthening of the problem. Given a set $Q_\iota \subseteq Q$ and a bound $n \in \mathbb{N}$, computing the number of finite sequences r in $(Q \times \Sigma)^n$ with $proj(1, r(0)) \in Q_\iota$ that reach an accepting loop in P can be done in time polynomial in n and in the size of P. The strengthening can be proven using

Algorithm 3.2 Counting Infinite Traces (countTraces)

Input: Unambiguous parity automaton $P = (\Sigma, Q, Q_0, \delta, \mu)$, $n \in \mathbb{N}$, $Q_\iota \subseteq Q$
Output: $\sum_{q_\iota \in Q_\iota} |L(q_\iota)|$
Begin:

```
for q_ι ∈ Q_ι do
    if detectDPL(P, q_ι) then
        return ∞

C := ∅
for q ∈ Q do                                   ▷ Mark states
    if ReachesAcceptingLoop(P, q) then
        C(q) = 1
    else
        C(q) := 0

for i := 1; i ≤ n; i := i + 1 do               ▷ Count
    C' := ∅
    for q ∈ Q do
        for α ∈ Σ do
            for q' ∈ δ(q, α) do
                C'(q) := C'(q) + C(q')
    C := C'

return Σ_{q_ι ∈ Q_ι} C(q_ι)
```

End.

induction over n. We leave the proof to Section 3.7. □

The next corollaries follow from Theorem 2.7, Theorem 2.4, and Theorem 3.5.

Corollary 3.4. *Let N be a nondeterministic parity automaton, and let $|L(N)| < \infty$. Computing the number of models of N can be done in space polynomial in the size of N.*

Corollary 3.5. *Let φ be an* LTL *formula, and let $|L(\varphi)| < \infty$. Computing the number of models of φ can be done in time exponential in $|\varphi|$.*

3.2.3 Complexity bounds

In the next three theorems, we present the complexity of the trace counting problem for omega-regular properties in terms of counting complexity classes. We provide lower and upper bounds for the counting problems for omega-regular

properties represented by deterministic parity automata, nondeterministic parity automaton, and LTL formulas.

Theorem 3.6. *Let D be a deterministic parity automaton, with $|L(D)| < \infty$. The problem of computing the number models of D is* #L*-complete.*

Proof. Let $D = (\Sigma, Q, q_0, \delta, \mu)$ with $|Q| = n$. To show that the problem is in #L, we define a nondeterministic logarithmic-space Turing machine M as follows. The machine M guesses a word $\rho = (q_0, \alpha_1) \cdot (q_1, \alpha_2) \cdots (q_{n-1}, \alpha_n)$ from $(Q \times \Sigma)^n$ letter by letter. After guessing the word, the machine M continues with guessing letters from $(Q \times \Sigma)$ until it guesses a letter with a state that is equal to the state of the nth letter, or unitl it has guessed n further letters. In both cases, the machine terminates. After guessing each letter, the machine checks whether the guessed letter satisfies the transition relation δ. If not it rejects. If the last guessed letter is not equal to the nth guessed letter the machine also rejects. Finally, if the highest color of a state of a letter guessed between the nth and the last letter is not even, then the machine also rejects. In all other cases, the machine accepts.

When the machine M accepts, the guessed word represents an accepting lasso in D. The first n letter represent the stem of the lasso. The sequence of letters between the nth letter and the last one represent the loop of the lasso. Each accepting run of M represents a unique lasso in D, because each word consisting of the first n letters defines a unique finite run in D, and because the machine terminates after the first reoccurrence of the state in letter n, all the loops guessed by the machine M are unique for the guessed stem. Furthermore, all accepting runs of D are represented by some accepting run of the machine, because an accepting loop of D is reached after at most n steps, otherwise D would contain a doubly-pumped lasso.

To meet the space requirement, M only stores the currently guessed letter, the letter guessed at position n, the highest color seen after the letter n, and uses a binary counter to guess exactly $2n$ symbols. The machine M discards any other guessed letters.

The lower bound of the problem matches its upper bound, and can be proven by encoding a logarithmic-space Turing machine into the problem. The reduction is given in Section 3.7. □

If the omega-regular property is given by a nondeterministic parity automaton, we will not be able to guess the run step by step, as one word may have

more than one run in the automaton. An algorithm for counting the number of models must thus guess the complete word in one step and verify whether it reaches some accepting loop in the automaton.

Theorem 3.7. *Let N be a nondeterministic parity automaton, with $|L(N)| < \infty$. The problem of computing the number models of N is* #P*-complete.*

Proof. Let $N = (\Sigma, Q, Q_0, \delta, \mu)$. To show that the problem is in #P, we define a nondeterministic polynomial-time Turing machine M that works as follows. The machine M guesses a word σ of length $|N|$ over Σ, and checks whether a run of σ in N reaches some accepting loop in N. The check can be done in polynomial time using a depth first search (DFS) procedure. The DFS starts at one of the states in Q_0, and, in each step checks if a transition in the automaton is consistent with the next letter of the guessed word. When the DFS iterates over the word and reaches some state $q \in Q$, the DFS procedures continues by looking for an accepting loop back to q. If such loop is not found, then the guessed word is not a prefix of a model of N.

Since N has no doubly-pumped lassos, each guessed word leading to an accepting loop is a prefix of a unique model of N. This implies that the number of accepting runs of M is equal to the number of models of N.

The matching lower bound can be proven by reducing the model counting problem for propositional formulas in disjunctive normal form (#DNF) to the problem [77, 148]. We leave the proof to Section 3.7. □

For LTL we need to count the number of minimal paths in an equivalent nondeterministic Büchi automaton that lead to a loop in that automaton. We can guess such a path on the fly in space polynomial in the LTL formula using the construction presented in Theorem 2.7.

Theorem 3.8. *Let φ be an* LTL *formula, with $|L(\varphi)| < \infty$. The problem of computing the number models of φ* #PSPACE*-complete.*

Proof. For proving the upper bound we can construct the following nondeterministic polynomial-space Turing machine M. From Theorem 2.7, we know that we can construct a nondeterministic Büchi automaton N the states of which are pairs of subsets of states of the alternating automaton, i.e., its sets of states is defined by $2^{closure(\varphi)} \times 2^{closure(\varphi)}$. If the number of models of φ is bounded, then the automaton N has no doubly-pumped lasso composed of two different las-

sos, and thus to count the number of models we can count the number of runs of length N leading to an accepting loop.

The machine M guesses on the fly a word of length $2^{|closure(\varphi)|} \cdot 2^{|closure(\varphi)|}$, and checks if this word has a run that leads to an accepting loop in N. The word and the run are guessed letter by letter, and state by state. The state is guessed by guessing an elementary subset from $closure(\varphi)$ and another subset from $closure(\varphi)$ and verifying it against the conditions of the transition relation as defined in Theorem 2.7. After guessing the word, the machine checks whether the reached state can be reached again from itself, passing through an accepting state in N (similar to the machine in Theorem 3.6). This can be done in exponential time since we only need to guess at most $2 \cdot 2^{|closure(\varphi)|} \cdot 2^{|closure(\varphi)|}$ states.

Since the constructed nondeterministic automaton is unambiguous, each accepting run of the machine defines a uniques model of φ, and thus counting the number of models of φ can be done by counting the number of accepting runs of M.

To show the matching lower bound, we can encode the runs of a polynomial-space Turing machine into an LTL formula using a parsimonious reduction. The reduction is presented in Section 3.7. □

3.3 Counting Bad Prefixes

In this section, we introduce algorithms for computing the number of bad prefixes of bounded length for omega-regular properties represented by deterministic parity automaton, nondeterministic parity automaton and LTL formulas.

The bad-prefix counting problem for linear-time properties is formalized as follows

Definition 3.2 (The Bounded Bad-prefix Counting Problem)**.** Given a linear-time property φ over an alphabet Σ, and a bound $n \in \mathbb{N}$, the bad-prefix counting problem for φ and n is the problem of computing the value

$$\#_{Bad}(\varphi, n) = |\{u \in \Sigma^n \mid u \in BadPref(\varphi)\}|.$$

We start with an algorithm for deterministic parity automaton and show that computing the number of bad prefixes can be done in polynomial time in the size of the automaton and the bound on the bad prefixes. For nondeterministic parity automata and LTL formulas, we first need to construct an equiva-

lent deterministic parity automaton. This results in model counting algorithms that are exponential in the size of the nondeterministic automaton and doubly-exponential in the length of the LTL formula.

We also give a complete complexity analysis of each of the model counting problems over the different representations with respect to counting complexity classes. Here, we show that the problems of computing the number of bad prefixes of bounded length are complete for the counting complexity classes #L, #P, and $\#_o$PSPACE, when the omega-regular property is represented by a deterministic parity automaton, a nondeterministic parity automaton, and an LTL formula, respectively.

3.3.1 Algorithms for counting bad prefixes

To determine the number of bad prefixes of some length for some omega-regular property represented by a deterministic parity automaton P, we apply the algorithm presented in Algorithm 3.3, which is based on the following idea. If a word w is a bad prefix for $L(P)$, then the state reached in P by following the word w must be one that lies on no accepting run in P. The algorithm marks each state in P that does not lie on some accepting run in P (lines 4-7). A state q lies on an accepting run in P if an accepting loop is reachable from q. In the next phase, the algorithm computes a mapping that maps each state q in P with the number of words of length n that lead to a marked reachable state from q. The mapping is computed iteratively for words of increasing length up to the desired length n. The mappings computed for a length i are used to compute the mappings for length $i+1$. This is done using the following approach. Initially, each marked state q, i.e., $q \in M$, is mapped to 1 (lines 8-13). All other states are mapped to 0. In iteration $i < n$ (lines 15-21) the number of words of length $i+1$ that lead to a state in M from some state q can be computed by adding up the number of words that lead to M from all successors q' of q. This holds because, every word w of length i that leads to a state q'' in M from in q', can be extended into a word $w' = \alpha \cdot w$ of length $i+1$, where $\delta(q, \alpha) = q'$, that leads to the same state q'' in M starting in q. After n iteration we have computed a mapping that maps each state in P with the number of words of length n that reach M. The number of bad prefixes of length n is equal to the mapping $C(q_0)$ of the initial state q_0. Figure 3.4 shows an execution run of Algorithm 3.3 on a deterministic parity automaton for the language $L(a \,\mathcal{U}\, b)$. The runtime of Algorithm 3.3 is polynomial in both the bound n and the size of the given deterministic parity automaton.

Algorithm 3.3 Counting Bad Prefixes (countBadPrefixes(P, n))

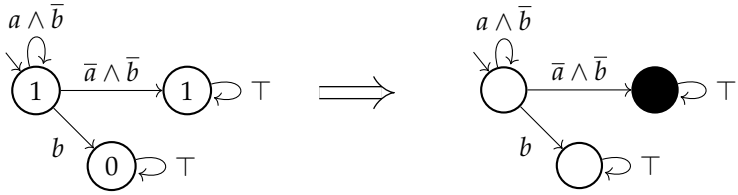
```
1:  Input: P = (Σ, Q, q_0, δ, μ), n ∈ ℕ
2:  Output: |{w ∈ Σ^n | w is a bad prefix for L(P)}|
3:  Begin:
4:  M := ∅
5:  for q ∈ Q do                                          ▷ Mark States
6:      if not ReachesAcceptingLoop(P, q) then
7:          M := M ∪ {q}
8:  C := ∅
9:  for q ∈ Q do                                          ▷ Initialize Count
10:     if q ∈ M then
11:         C(q) := 1
12:     else
13:         C(q) := 0
14: i := 0
15: while i < n do                                        ▷ Count
16:     C' := ∅
17:     for q ∈ Q do
18:         for α ∈ Σ do
19:             q' := δ(q, α)
20:             C'(q) := C'(q) + C(q')
21:     C := C'
22:     i := i + 1
23: return C(q_0)
24: End.
```

Theorem 3.9. *Computing the number of bad prefixes of length n for an omega-regular property, represented by a deterministic parity automaton D, can be done in time polynomial in n and $|D|$.*

Proof. The counting algorithm presented in lines 15 to 22 in Algorithm 3.3 resembles the algorithm for counting finite words of length n in a finite state automaton, if we consider the marked states to be the accepting states of the finite automaton [93]. This algorithm computes the number of words in polynomial time in n and the size of the finite automaton. To prove the correctness of the theorem we need to show that the markings of the states represent a finite state automaton that accepts all bad prefixes of $L(D)$, and that such markings can be computed in polynomial time in n and $|D|$.

Let $D = (\Sigma, Q, q_0, \delta, \mu)$. Assume a word w is a bad prefix for $L(D)$, and let $q_w \in Q$ be the state reached in D following the word w. If q_w is not marked, i.e., $q_w \notin M$, then q_w lies on some accepting path in D. This however means that

▷ Mark States:



▷ Count bad prefixes of length 2:

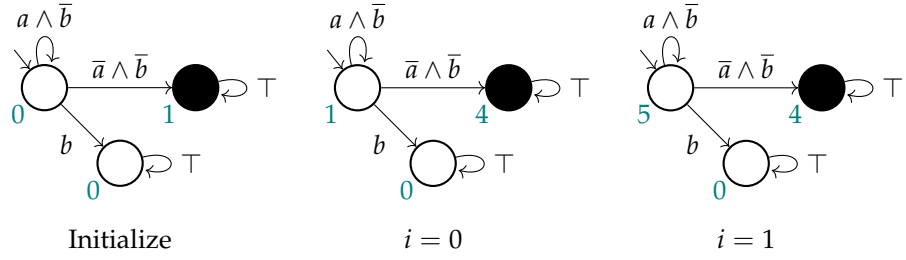


Figure 3.4: Using Algorithm 3.3 to compute the number of bad prefixes of length 2 for $a \,\mathcal{U}\, b$.

there is an infinite word σ such that $w \cdot \sigma \in L(D)$, which in turn means that w cannot be a bad prefix. If a word w is not a bad prefix for $L(D)$, then its run does not have any marked states, because if it would have such a state, then w has a prefix that is a bad prefix for $L(D)$, and thus w must also be a bad prefix for $L(D)$.

For each state $q \in Q$, we can check whether it lies on an accepting run, by checking whether there is a run $q_1 \dots q_{i-1}(q_i \dots q_m)^\omega$ in D, with $q_1 = q$, and such that $\max\{\mu(q_j) \mid i \leq j \leq m\}$ is even. Finding such a run can be done in time polynomial in $|D|$ (the reachability problem in graphs is in NL [111]). This results in a total complexity that is polynomial in n and $|D|$. □

If the omega-regular property is given by a nondeterministic parity automaton, or an LTL formula, we first need to construct an equivalent deterministic parity automaton, before applying Algorithm 3.3. This results in an exponential and doubly-exponential blow-up in the size of the nondeterministic automaton and the length of the LTL formula, respectively.

Corollary 3.6. *Computing the number of bad prefixes of length n for an omega-regular property, can be done in time polynomial in n, and*

- *exponential in $|N|$, where N is a non-deterministic parity automaton representing the omega-regular property.*
- *doubly-exponential in $|\varphi|$, where φ is an* LTL *formula representing the omega-regular property.*

3.3.2 Complexity bounds

We provide the complexity of the problem of counting bad prefixes in terms of counting complexity classes. We start by showing that the problem for omega-regular properties given as a deterministic parity automaton is #L-complete.

Theorem 3.10. *The problem of computing the number of bad prefixes of length $n \in \mathbb{N}$ for an omega-regular property given as a deterministic parity automaton is* #L*-complete.*

Proof. The completeness of the problem for the class #L follows from the completeness of the problem of counting words of a certain length for DFAs for the same counting complexity class [5]. We show how we can reduce the word counting problem for DFAs to the bad-prefix counting problem for DPAs, and vice versa.

Given a DPA $D = (\Sigma, Q, q_0, \delta, \mu)$, we construct a DFA $A = (\Sigma, Q, q_0, \delta, F)$ where F is defined by the set of states in $Q \setminus W$ where W is the set of states from which an accepting loop in D can be reached. Given a bound n, the number of bad prefixes of length n of the omega-regular property represented by D is equal to the number of words of length n accepted by A.

To prove the lower bound, let $A = (\Sigma, Q, q_0, \delta, F)$, and let $n \in \mathbb{N}$. We construct a DPA $D = (\Sigma, Q \times \{0, \ldots, n+1\}, q_0', \delta', \mu)$, where

- $q_0' = (q_0, 0)$
- $\delta((q,c),\alpha) = \begin{cases} (q', c+1) & c < n-1,\ \delta(q,\alpha) = q' \\ (q', n) & c = n-1,\ \delta(q,\alpha) = q',\ q' \in F \\ (q', n+1) & c = n-1,\ \delta(q,\alpha) = q',\ q' \notin F \\ (q', n) & c = n,\ \delta(q,\alpha) = q' \\ (q', n+1) & c = n+1,\ \delta(q,\alpha) = q' \end{cases}$
- $\mu(q,c) = \begin{cases} 1 & c \leq n \\ 0 & c = n+1 \end{cases}$

The automaton D follows the transition relation of A and additionally tracks the length of a word up to a length n. If, after reading n letters, the word is accepted by A, then the word is a bad prefix for the property represented by D, since any run starting in the state reached by that word will have an infinite appearance of the color 1, and thus is a rejecting run in D. If the word ends in a nonaccepting state of A, every run starting in that state will have an infinite appearance of the color 0, und is thus accepting. In conclusion, every word of length n accepted by A represents a unique bad prefix of length n for D. □

The completeness of the problem of counting bad prefixes for an omega-regular property given by a nondeterministic parity automaton for the class #P can be shown using similar constructions.

Theorem 3.11. *The problem of computing the number of bad prefixes of length $n \in \mathbb{N}$ for an omega-regular property, represented by a nondeterministic parity automaton is* #P*-complete.*

Proof. The completeness of the problem follows from the completeness of the problem of counting words of a certain length for a NFA which is complete for the class #P [93].

The reductions are similar to the reductions presented in the last theorem. Given an NPA, we can construct an NFA in the same way we did for DPAs. For the other direction, we construct an NPA in the same way as for DPAs, this time respecting the nondeterministic transition relation of a given NFA. □

To prove the upper bound for LTL, we make use of its polynomial-space model checking algorithm.

Theorem 3.12. *The problem of computing the number of bad prefixes of length $n \in \mathbb{N}$ for an omega-regular property, represented by an* LTL *formula is in* $\#_0$PSPACE.

Proof. Let φ be an LTL formula defined over the set of atomic propositions AP. We construct the following nondeterministic polynomial-time oracle Turing machine M with a polynomial-space oracle. The machine M guesses a word of length n over 2^{AP} and checks using the oracle whether the guessed word is a bad prefix for φ. This can be done in polynomial space with the same procedure used for checking the satisfiability of LTL formulas [138], with the additional constraint that a model of the LTL formula must start with the guessed word.

If the oracle accepts the guessed word then the machine M rejects, otherwise it accepts. □

Theorem 3.13. *The problem of computing the number of bad prefixes of length $n \in \mathbb{N}$ for an omega-regular property, represented by an* LTL *formula is* #P*-hard.*

Proof. We can encode the runs of a nondeterministic polynomial-time Turing machine using the same encoding presented in the proof of Theorem 3.8 (See Section 3.7). This time, we can encode the Id's using only logarithmic many bits in the size of the Turing machine. We choose the bound n to be equal to the maximum length of a run of the Turing machine. The number of bad prefixes of length n of the negation of the LTL formula is equal to the number of accepting runs of the Turing machine. □

3.3.3 Counting good prefixes

Good-prefixes are related to co-safety properties, which require a "good thing to eventually happen" (See Section 2.2.1). Co-safety properties are dual to safety properties, and thus a good prefix for a co-safety property φ is a bad prefix for the dual safety property $\overline{\varphi}$. Computing the number of good prefixes of length n for a property can be done by computing the number of bad prefixes of length n for the dual property.

The good-prefix counting problem for linear-time properties is formalized as follows

Definition 3.3 (The Bounded Good-prefix Counting Problem)**.** Given a linear-time property φ over an alphabet Σ, and a bound $n \in \mathbb{N}$,the good-prefix counting problem for φ and n is the problem of computing the value

$$\#_{Good}(\varphi, n) = |\{u \in \Sigma^n \mid u \in GoodPref(\varphi)\}|.$$

We summarize the complexities for computing the number of good prefixes for an omega-regular properties in the following theorem.

Theorem 3.14. *Computing the number of good prefixes of length n for an omega-regular property can be done in*

1. *time and space polynomial in $|D|$ and n, for a deterministic parity automaton D.*

2. *time exponential in n, and space logarithmic in $|D|$ and polynomial in n, for a deterministic parity automaton D.*

3. *time and space exponential in $|N|$ and polynomial in n, for a nondeterministic parity automaton N.*

4. *time exponential in n, and space polynomial in $|N|$ and n, for a nondeterministic parity automaton N.*

5. *time exponential in n, and space logarithmic in $|U|$ and polynomial in n, for a universal co-Büchi automaton U.*

6. *time and space doubly-exponential in $|\varphi|$ and polynomial in n, for an* LTL *formula φ.*

7. *time exponential in n, and space polynomial in $|\varphi|$ and n, for an* LTL *formula φ.*

Proof. The results in 1 and 2 follow from the results for computing the number of bad prefixes for deterministic parity automata. When the omega-regular property is given by a nondeterministic parity automaton we need to transform this automaton to a deterministic automaton, introducing an exponential blow-up in the size of the automaton (3). If we enumerate words of length n, we can check whether this words are bad prefixes for the dual property by guessing the run of the deterministic automaton on the fly. Statement 5 follows from dualizing the automaton to a nondeterministic automaton and counting the bad prefixes. As for LTL, the complexity in 6 and 7 result from negating the LTL formula and applying the procedures for counting bad prefixes. □

With respect to counting complexity classes we summarize the results in the following theorem.

Theorem 3.15. *The problem of computing the number of good prefixes of length $n \in \mathbb{N}$ for an omega-regular property φ is*

1. #L*-complete when φ is represented by a deterministic parity automaton.*

2. #P*-hard and in* $\#_o$PSPACE *when φ is represented by a nondeterministic parity automaton.*

3. #P*-complete when φ is represented by a universal co-Büchi automaton.*

4. #P*-hard and in* $\#_o$PSPACE *when φ is represented by an* LTL *formula.*

Proof. Statement 1 follows from the #L-completeness of the bad-prefix counting problem, since complementing the deterministic automaton can be done in polynomial time.

To prove the lower bound in statement 2, we can again encode a DNF formula as in the proof of Theorem 3.7. An algorithm for computing the number of good prefixes is one that enumerates all words of length n, and checks whether a word is a bad prefix for the complement language.

Statement 3 follows from the #P-completeness of the bad-prefix counting problem, since any universal co-Büchi automaton can be dualized to a nondeterministic Büchi automaton in polynomial time.

Statement 4 follows from the results for counting bad prefixes. An algorithm for computing the number of good prefixes is one that counts the bad prefixes for the negated formula. The lower bound is shown by reducing a nondeterministic polynomial-time Turing machine to the problem. □

3.4 Counting Lassos

We proceed with the lasso counting problem for linear-time properties, and introduce algorithms for solving the problem for omega-regular properties given by deterministic parity automata, nondeterministic parity automata and LTL formulas. The lasso counting problem is formalized as follows

Definition 3.4 (The Bounded Lasso Counting Problem)**.** Given a linear-time property φ over an alphabet Σ, and a bound $n \in \mathbb{N}$, let $\mathit{Lasso}(\varphi, n) = \{(u, v) \in \Sigma^* \times \Sigma^+ \mid u \cdot v| = n, u \cdot v^\omega \models \varphi\}$. The lasso counting problem for φ and n is the problem of computing the value

$$\#_{\mathit{Lasso}}(\varphi, n) = |\mathit{Lasso}(\varphi, n)|.$$

We start with a symbolic algorithm based on a translation to propositional logic. For deterministic parity automata, nondeterministic parity automata, and LTL formulas, the translations are all polynomial in both the size of the representation and the bound. This in turn means that the run time of our algorithm is exponential in the size of the representation and the bound due to the exponential complexity of solving propositional formulas (SAT is NP-complete [124]). We also present an automata-based construction that improves the complexity

exponentially in the bound. This however comes at the cost of an exponential blow-up in the complexity with respect to the size of the representation.

3.4.1 Algorithms for counting lassos

We start with the symbolic approaches by encoding the lasso counting problems for deterministic parity automata, nondeterministic parity automata, and LTL formulas into propositional constraints. For an omega-regular property φ, and a bound n, a satisfying assignment of the propositional constraint is one that uniquely represents a lasso of length n that satisfies φ. Computing the number of lassos of length n is thus done by counting the satisfying assignments of the propositional constraint. Encoding the satisfaction relation of a lasso and φ depends on the representation in which φ is given. We present the constructions and the different encodings of the satisfaction relation in the next three theorems.

Theorem 3.16. *Given a deterministic parity automaton D, and a bound $n \in \mathbb{N}$, we can construct a propositional formula ϕ of size polynomial in $|D|$ and n such that the number of models of ϕ is equal to $\#_{Lasso}(L(D), n)$.*

Proof. Let $D = (\Sigma, Q, q_0, \delta, \mu)$ with $|D| = k$, and w.l.o.g. assume that $\mu : Q \to \{0, \ldots, |D| - 1\}$. Our encoding is based on the following fact. Let a lasso $\rho = (u, v) \in \Sigma^* \times \Sigma^+$ with $|u \cdot v| = n$. Since both the lasso ρ and the automaton D are finite, the run r of the word $u \cdot v^\omega$ can be represented by a lasso $\rho_r = (u_r, v_r) \in (Q \times \Sigma)^* \times (Q \times \Sigma)^+$ of length at most $n \cdot k$. In our encoding below, a satisfying assignment represents a lasso ρ of length n over Σ, and another lasso ρ_r over $Q \times \Sigma$ of length $n \cdot k$ representing the accepting run of D on the word $u \cdot v^\omega$. Although, D is deterministic, which means that each accepted word by D has a unique run in D, a run can still be represented by two different lasso runs (See Figure 3.5). To avoid counting multiple lasso runs for the same infinite run, we make sure that ϕ only accepts the valuation of one unique lasso for the run.

The propositional formula ϕ is defined as the following conjunction of propositional formulas

$$\phi_{init} \wedge \phi_{loop} \wedge \phi_{run} \wedge \phi_{accepting} \wedge \phi_{unique}.$$

The formula ϕ_{init} is defined over variables $V_\delta = \{\delta_{(q,\alpha,q')} \mid q, q' \in Q, \alpha \in \Sigma\}$ and variables $V_\mu = \{\mu_q^c \mid q \in Q, 0 \leq c \leq |D| - 1\}$ that encode the automaton D. A variable $\delta_{(q,\alpha,q')}$ is used to state whether the transition $\delta(q, \alpha) = q'$ is a valid transition in D. A variable μ_q^c is used to state whether the coloring of q in D

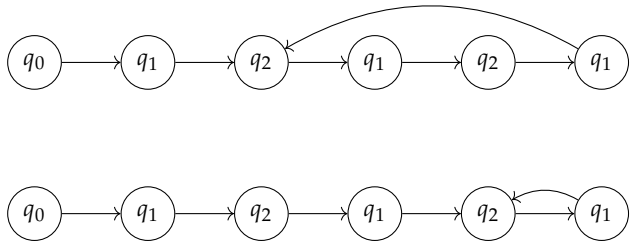


Figure 3.5: Two lasso runs inducing the same run.

is equal to c. Since D is deterministic, if a variable $\delta_{(q,\alpha,q')}$ is used in a positive literal, then all variables $\delta_{(q,\alpha,q'')}$ for $q' \neq q'' \in Q$ must be used in a negative literal, i.e.,

$$\bigwedge_{q\in Q} \bigwedge_{\alpha\in\Sigma} \bigwedge_{q'\in Q} \delta_{(q,\alpha,q')} \rightarrow \bigwedge_{q''\neq q'} \overline{\delta_{(q,\alpha,q'')}} \ .$$

Furthermore, every state of D is assigned to only one color. This means for a state $q \in Q$ the variables μ_q^c can be positive for only one $0 \leq c \leq |D| - 1$, i.e.,

$$\bigwedge_{q\in Q} \bigwedge_{0\leq c\leq |D|-1} \mu_q^c \rightarrow \bigwedge_{0\leq c'\neq c\leq |D|-1} \overline{\mu_q^{c'}} \ .$$

The formula ϕ_{loop} defines the loop position of the lasso ρ, and is defined over the variables $V_{loop} = \{\ell_i \mid 0 \leq i \leq n-1\}$. A variable ℓ_i is used to state whether the loop of the lasso is at position i. Each lasso has a unique loop position. This means that only one variable ℓ_i for $0 \leq i \leq n-1$ can be true at a time, i.e,

$$\bigwedge_{0\leq i\leq n-1} \ell_i \rightarrow \bigwedge_{0\leq j\neq i\leq n-1} \overline{\ell_j} \ .$$

The formula ϕ_{run} defines the lasso ρ_r representing the run r in D on the lasso ρ, and is defined over the variables $V_{states} = \{q_h \mid q \in Q,\ 0 \leq h \leq n \cdot k\}$ and $V_{letters} = \{\alpha_i \mid \alpha \in \Sigma,\ 0 \leq i \leq n\}$. A variable q_h is used to state whether the state at position h in the lasso ρ_r of the automaton D is equal to q. A variable α_i is used to state whether the ith letter of a lasso ρ is equal to α. Furthermore, the formula determines the loop position in the lasso ρ_r using the variables $V_{runLoop} = \{r_h \mid 0 \leq i \leq n \cdot k - 1\}$. As for the loops of the lasso ρ, a variable r_h is used to state whether the loop of the run of the lasso is at position h, and only one variable for some position h can be true at a time. To encode the lasso ρ_r, the formula ϕ_{run} defines the relation between the states and letters along the run with respect to the letter of the lasso ρ and the transition relation δ of D. The constraint ϕ_{run} is

given as follows

$$q_{00} \tag{3.1}$$

$$\wedge \bigwedge_{0 \le i \le n \cdot k - 1} \bigwedge_{q,q' \in Q} \bigwedge_{\alpha \in \Sigma} (q_i \wedge \bigvee_{0 \le j < n} (\ell_j \wedge \alpha_{\Delta(i,n,j)}) \rightarrow (\delta_{q,\alpha,q'} \leftrightarrow q'_{i+1})) \tag{3.2}$$

$$\wedge \bigwedge_{q \in Q} \bigwedge_{\alpha \in \Sigma} ((q_{n \cdot k} \leftrightarrow (\bigvee_{0 \le j < n \cdot k} r_j \wedge q_j))) \tag{3.3}$$

The lasso representing the run must start with the initial state of D (3.1). If a state q is at position i in the lasso run ρ_r, and α is the letter in the lasso ρ corresponding to the position i in ρ_r, then the next state q' at position $j+1$ must satisfy the transition relation δ (3.2). The correspondence between a position in ρ_r and ρ is computed using the function Δ, which is defined by

$$\Delta(i,n,j) = \begin{cases} i & \text{if } i < n, \\ ((i-n) \mod (n-j)) + j & \text{otherwise.} \end{cases}$$

where $n - j$ is the size of the loop of the lasso ρ. The last formula (3.3) makes sure that the states form a lasso. This is encoded by checking that the last state is equal to the state at the loop position defined by the variables $V_{runLoop}$, i.e., if the state at position $n \cdot k$ is equal to some state $q \in Q$, then the state at position j, where j is the loop position of the lasso ρ_r must be equal to q.

The formula $\phi_{accepting}$ ensures that the lasso ρ_r is an accepting lasso, i.e., the highest color observed in the loop is even. Let $o_1, \ldots, o_h \in \{c \mid 0 \le c \le |D| - 1, c \text{ is odd } \}$, and let $e_1, \ldots, e_{h'} \in \{c \mid 0 \le c \le |D| - 1, c \text{ is even } \}$ for some h, h'. The formula $\phi_{accepting}$ is defined by

$$\bigvee_{1 \le i \le h'} highestColor_{e_i} \text{, where}$$

$$highestColor_x = \bigvee_{0 \le j < n \cdot k} (r_j \wedge existsColor_{j,x} \wedge \bigwedge_{x < x' < |D|-1} \overline{existsColor_{j,x'}})$$

$$existsColor_{j,x} = (\bigvee_{q \in Q} (q_j \wedge \mu_q^x)) \vee existsColor_{j+1,x}$$

$$existsColor_{n \cdot k,x} = \text{false}$$

Lastly, the formula ϕ_{unique} makes sure that we only choose one representative ρ_r for the run r. To understand the idea of ϕ_{unique}, consider again the two lassos depicted in Figure 3.5. Both lasso runs induce the same infinite run in D. When solving the constraint system using only the formulas above, we might get two satisfying assignments representing two different lassos for the same run in D.

To avoid counting a run twice, with the formula ϕ_{unique}, we add a constraint that only accepts the lower lasso (the one with the shortest loop). The formula ϕ_{unique} is defined as follows

$$\bigvee_{0 \le i < n \cdot k} \bigvee_{q \in Q} \bigvee_{0 \le j < n} (r_i \wedge q_i \wedge \ell_j \wedge \overline{existsState(i+1, q, j, \Delta(i, n, j))}) \ ,$$

where

$$\begin{aligned} existsState(i, q, j, d) = & \bigwedge_{q' \in Q} (q'_i \rightarrow q = q') \wedge (d = \Delta(i, n, j)) \\ & \vee existsState(i+1, q, j, d) \\ existsState(n \cdot k, q, j, d) = & \mathit{false} \ . \end{aligned}$$

The number of variables needed to encode the lasso counting problem is polynomial in n and $|D|$. The size of each of the formulas presented above is also polynomial both in the bound n and in the size of the automaton D. □

In the next theorem we provide an encoding of the lasso counting problem for nondeterministic parity automata. The encoding is based on the satisfaction relation of the universal co-Büchi automaton for the complement language. Counting the number of lassos of the complement language allows to determine the number of lasso of the language itself by computing the difference to the total number of lassos.

Theorem 3.17. *Given a universal co-Büchi automaton U, and a bound $n \in \mathbb{N}$, we can construct a propositional formula ϕ of size polynomial in $|U|$ and n such that the number of models of ϕ is equal to $\#_{Lasso}(L(U), n)$.*

Proof. Let $U = (\Sigma, Q, q_0, \delta, R)$, and w.l.o.g. assume q_0 has no ingoing transitions. Our encoding is based on the following fact. If an infinite word over Σ represented by a lasso $\rho = (u, v)$ is accepted by U, then its run does not have a branch with infinite appearances of rejecting states of U (remember, the acceptance condition of a universal co-Büchi automaton requires no rejecting state to appear infinitely often on any branch of the run). Since both the automaton and the lasso are of finite size, the run of U on ρ can be given by a finite automaton $U' = (\Sigma, Q', q'_0, \delta', R')$ where

- $Q' = Q \times \{0, \ldots, n-1\}$, where each state represents a state in U and a position in ρ.

- $q'_0 = (q_0, 0)$
- $R' = R \times \{0, \ldots, n-1\}$
- $\delta'((q,c),\alpha) = \begin{cases} (\delta(q,\alpha), ((c+1) \mod (n - |u|)) + |u|) & \text{if } \alpha = (u \cdot v)(c) \\ \bot & \text{otherwise} \end{cases}$

The automaton U' for ρ is unique, since U is universal. In our encoding below, a satisfying assignment represents a lasso ρ of length n and its accepting run represented by U'. The number of satisfying assignments is thus equal to the number of lassos of length n representing models of U.

The formula ϕ is defied by the following conjunction

$$\phi_{init} \wedge \phi_{loop} \wedge \phi_{run} \wedge \phi_{accepting} \wedge \phi_{unique} \ .$$

The formula ϕ_{init} encodes the transition relation of U and is defined over variables $V_\delta = \{\delta_{(q,\alpha,q')} \mid q, q' \in Q,\ \alpha \in \Sigma\}$ that define the transitions of U.

The formula ϕ_{loop} defines a unique loop position of the lasso ρ, and is defined over variables $V_{loop} = \{\ell_i \mid 0 \leq i \leq n-1\}$.

The formula ϕ_{run} defines the automaton U' and is defined over variables $V_{states} = \{v_{(q,c)} \mid q \in Q, c \in \{0, \ldots, n-1\}\}$, and $V_{letters} = \{\alpha_i \mid \alpha \in \Sigma,\ 0 \leq i \leq n\}$. The formula ϕ_{run} determines which states in U' are reachable from the initial state $(q_0, 0)$ and is defined as follows

$$\begin{aligned} & v_{(q_0,0)} \\ & \wedge \bigwedge_{q \in Q} \bigwedge_{c \in \{0,\ldots,n-1\}} \Big(\ v_{(q,c)} \leftrightarrow \\ & \qquad \bigvee_{q' \in Q} \bigvee_{c' \in \{0,\ldots,n-1\}} \bigvee_{\alpha \in \Sigma} \ v_{(q',c')} \wedge \\ & \qquad\qquad \bigwedge_{0 \leq i < n} (\ell_i \rightarrow c' = \Delta(c,n,i)) \wedge \\ & \qquad\qquad \delta_{(q',\alpha,q)} \wedge \\ & \qquad\qquad \alpha_{c'} \\ & \qquad \Big) \ . \end{aligned}$$

The formula describes that if a state (q,c) is reachable from $(q_0,0)$, then there must be a predecessor (q',c') that reaches (q,c) via a letter α, and such that c' is a position preceding c in the lasso ρ, either directly or via the loop, and α must be equal to the letter in ρ at position c'.

The formula $\phi_{accepting}$ makes sure that U' is accepting. As mentioned earlier, the automaton U' is accepting, if it contains no loop with a rejecting state. This can be defined using the following idea that we adopt from the bounded synthesis encoding [67]. If U' has no loop with a rejecting state, then the number of rejecting state observed so far up to a state in U' is finite. Thus each state in U' can be annotated by the number of rejecting state seen up to this state. The formula $\phi_{accepting}$ is defined over the variables $V_{annotate} = \{\lambda^d_{(q,c)} \mid q \in Q, c \in \{0,\ldots,n-1\}, d \in \{0,\ldots,|Q|\cdot n\} \cup \{-\}\}$. We define $\phi_{accepting}$ as follows

$$
\begin{aligned}
&\lambda^0_{(q_0,0)} \wedge \\
&\bigwedge_{q\in Q} \bigwedge_{c\in\{0,\ldots,n-1\}} \bigwedge_{d\in\{0,\ldots,|Q|\cdot n\}\cup\{-\}} \lambda^d_{(q,c)} \rightarrow \bigwedge_{d\neq d'\in\{0,\ldots,|Q|\cdot n\}\cup\{-\}} \overline{\lambda^{d'}_{(q,c)}} \wedge \\
&\bigwedge_{q\in Q\setminus R, q'\in Q} \bigwedge_{c,c'\in\{0,\ldots,n-1\}} Succ((q,c),(q',c')) \wedge \overline{\lambda^-_{(q,c)}} \rightarrow Ann(q',c') \geq Ann(q,c) \\
&\bigwedge_{q\in R, q'\in Q} \bigwedge_{c,c'\in\{0,\ldots,n-1\}} Succ((q,c),(q',c')) \wedge \overline{\lambda^-_{(q,c)}} \rightarrow Ann(q',c') > Ann(q,c) \ ,
\end{aligned}
$$

where $Succ((q,c),(q',c'))$ is true when (q',c') is a successor of (q,c), and where $Ann(q',c') \geq Ann(q,c)$ is true when the annotation of (q',c') is larger than or equal to the annotation of (q,c). If U' is not accepting, then there is no annotation that satisfies the formula $\phi_{accepting}$.

Lastly, the formula ϕ_{unique} makes sure that the annotation is unique. This is done by requiring that each state is annotated with the exact maximum number of rejecting states seen along a path leading to this state, and by requiring that each state not reachable from the initial state is annotated with "-". The formula is define as follows

$$
\begin{aligned}
&\bigwedge_{q\in Q} \bigwedge_{c\in\{0,\ldots,n-1\}} v_{(q,c)} \leftrightarrow \overline{\lambda^-_{(q,c)}} \\
&\forall q \in Q.\, \forall c \in \{0,\ldots,n-1\}.\, \forall d \in \{0,\ldots,|U'|\}. \\
&\lambda^d_{(q,c)} \rightarrow \exists q' \in Q \setminus R.\, \exists c' \in \{0,\ldots,n-1\}.\, Succ((q',c'),(q,c)) \wedge \lambda^d_{(q',c')} \vee \\
&\qquad \exists q' \in R.\, \exists c' \in \{0,\ldots,n-1\}.\, Succ((q',c'),(q,c)) \wedge \lambda^{d-1}_{(q',c')} \ .
\end{aligned}
$$

□

The encoding presented in the last theorem can be used to compute the number of lassos for some bound n for a nondeterministic parity automaton, by first

translating the automaton to a universal co-Büchi automaton for the complement language. The number of lassos can then be computed by subtracting the number of lassos accepted by the universal co-Büchi automaton from the total number of lassos of size n over Σ.

Corollary 3.7. *Given a nondeterministic parity automaton N over an alphabet Σ, and a bound $n \in \mathbb{N}$, we can construct a propositional formula ϕ of size polynomial in $|N|$ and n such that the number of models of ϕ is equal to $n \cdot (2^{|\Sigma|})^n - \#_{Lasso}(L(N), n)$.*

Theorem 3.18. *Given an* LTL *formula φ, and a bound $n \in \mathbb{N}$, we can construct a propositional formula ϕ of size polynomial in $|\varphi|$ and n such that the number of models of ϕ is equal to $\#_{Lasso}(L(\varphi), n)$.*

Proof. We use the same idea as in the encoding of the bounded model checking problem [22]. The formula ϕ is defined as the conjunction

$$\phi_{loop} \wedge \phi_{\varphi} \ .$$

Let $V_{AP} = \{a_i \mid a \in AP, 0 \leq i < n\}$ such that a_i states whether the atomic proposition a is true at position i in the lasso. The formula ϕ_{loop} ensures that there is exactly one loop, as we have seen in the last two theorems.

The formula ϕ_{φ} encodes the satisfaction relation of φ by stating which valuation of atomic propositions are valid at a every position in the lasso. The formula defines an unrolling of the alternating automaton of φ over a lasso of size n and is given inductively as follows

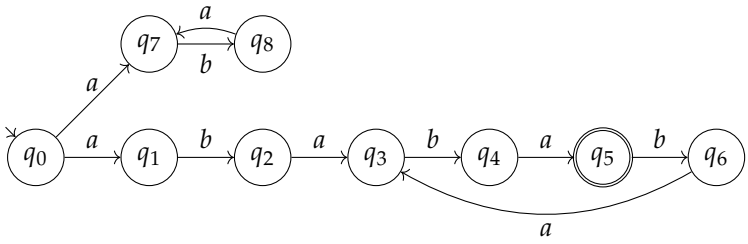
	$i < n$	$i = n$
$[\![a]\!]_n^i$	a_i	$\bigvee_{j=0}^{n-1}(l_j \wedge a_j)$
$[\![\neg a]\!]_n^i$	$\neg a_i$	$\bigvee_{j=0}^{n-1}(l_j \wedge \neg a_j)$
$[\![\bigcirc \varphi_1]\!]_n^i$	$[\![\varphi_1]\!]_n^{i+1}$	$\bigvee_{j=0}^{n-1}(l_j \wedge [\![\varphi_1]\!]_n^{j+1})$
$[\![\varphi_1 \,\mathcal{U}\, \varphi_2]\!]_n^i$	$[\![\varphi_2]\!]_n^i \vee ([\![\varphi_1]\!]_n^i \wedge [\![\varphi_1 \,\mathcal{U}\, \varphi]\!]_n^{i+1})$	$\bigvee_{j=0}^{n-1}(l_j \wedge \langle \varphi_1 \,\mathcal{U}\, \varphi_2 \rangle_n^j)$
$\langle \varphi_1 \,\mathcal{U}\, \varphi_2 \rangle_n^i$	$[\![\varphi_2]\!]_n^i \vee ([\![\varphi_1]\!]_n^i \wedge \langle \varphi_1 \,\mathcal{U}\, \varphi \rangle_n^{i+1})$	false
$[\![\varphi_1 \,\mathcal{R}\, \varphi_2]\!]_n^i$	$[\![\varphi_2]\!]_n^i \wedge ([\![\varphi_1]\!]_n^i \vee [\![\varphi_1 \,\mathcal{R}\, \varphi]\!]_n^{i+1})$	$\bigvee_{j=0}^{n-1}(l_j \wedge \langle \varphi_1 \,\mathcal{R}\, \varphi_2 \rangle_n^j)$
$\langle \varphi_1 \,\mathcal{R}\, \varphi_2 \rangle_n^i$	$[\![\varphi_2]\!]_n^i \wedge ([\![\varphi_1]\!]_n^i \vee \langle \varphi_1 \,\mathcal{R}\, \varphi \rangle_n^{i+1})$	true

□

All encodings are polynomial in the bound and in the size of the representation, solving the model counting problem thus requires time exponential in the bound and the size of the representation.

We introduce an alternative automaton-based approach that improves the time complexity exponentially in the bound, but at the cost of an exponential blow-up in the size of the representation if the property is given as an unambiguous parity automaton, or a double-exponential blow-up when the property is given by a nondeterministic parity automaton or an LTL formula.

The automata-based approach builds on the following fact. Consider the automaton depicted in Figure 3.6, and the lasso $\rho = (u, v)$ with $u = a \cdot b$ and $v = a \cdot b$. Since both the lasso and the automaton are bounded in size, we can give a finite representation for the run of the automaton on the word $u \cdot v^\omega$ as depicted in Figure 3.6. For each position in the lasso, we track the states of the automaton visited at that position and order them with respect to the order in which they were visited. Consider for example the representation of the run on the left. In the first iteration, we visit the states q_0, q_1, q_2, q_3, and in this order. In the second iteration, we have already entered the loop of the lasso and thus only positions within the loop are marked again by the states of the automaton visited at each position. Traversing the positions of the loop, we visit the states q_4, q_6, and in the iteration that follows, we visit the states q_6, q_3. At this point we have reached a loop in the automaton, because we have seen the state q_3 twice at the end of the loop. From here on, the sequences of states q_4, q_5 and q_6, q_3 will repeat interchangeably forever. In general, such a loop is reached in at most n iterations, where n is the size of the automaton. In our example, the left run can thus be represented by the tuple $(q_0, q_1, (q_2, q_4, q_6), (q_3, q_5, q_3))$. Such a representation is unique for each run on the lasso.



a	b	$($	a	b	$)^\omega$
q_0	q_1	q_2	q_3		q_4
		q_4	q_5		q_6
		q_6	q_3		q_4
		q_4	q_5		

a	b	$($	a	b	$)^\omega$
q_0	q_7	q_8	q_7		
		q_8	q_7		

Figure 3.6: Runs of a Büchi automaton over a lasso.

Given a lasso $\rho = (u, v)$, we can check whether $u \cdot v^\omega$ is accepted by some nondeterministic automaton N, by guessing the representation of a run of N on ρ, and checking whether it represents an accepting run. The acceptance condition is checked by verifying that along the sequence of states between the last state in the representation (in our example this was q_3) and its repetition in the representation, we have observed an accepting state of the automaton (in our example, we have observed the state q_5 in the second iteration of the loop, which lies between the two appearances of q_3).

Guessing the representation can be done nondeterministically by first guessing the tuple of states that would have appeared after the end of the loop (in our example this is the tuple (q_4, q_6, q_4)), and guessing the next tuple while traversing the word $u \cdot v$ backwards. The guessing is done nondeterministically by choosing a predecessor for each state in the tuple (in our example we guess the tuples, (q_3, q_5, q_3) ,(q_2, q_4, q_6), in this order). When a tuple t' is found that is equal to the initially guessed tuple t shifted by one position, i.e., for all $1 \leq i < n$, $t(i) = t'(i+1)$, it remains to check that the state at the first position of the tuple t' (in our case state q_2) is reachable from the initial state (q_0) of the automaton with the remainder of the letters in $u \cdot v$ (in our example, moving backwards we read the letters b then a and reach the state q_0).

In the next theorem, we build a nondeterministic finite automaton A that is based on the procedure we just described. For a nondeterministic Büchi automaton N, and a finite word w, the automaton A checks whether w^{-1} can be split into words $w = u \cdot v$ such that $u \cdot v^\omega$ is in $L(N)$. As explained above, the automaton guesses a tuple of $|N|$ states in N, and checks whether the tuple shifted by one position to the right can be reached traversing w, and whether an accepting state from N is observed along the way between the two iterations describing the repetition of the last state of the representation. If such a tuple is found for the initially guessed tuple, then A checks if the initial state of N can be reached from the first state in this tuple. The automaton is exponential in the size of N.

Theorem 3.19. *For a nondeterministic Büchi automaton N over Σ, we can construct a nondeterministic finite automaton A over Σ such that a word $w \in L(A)$ if and only if there are words $u \in \Sigma^*$ and $v \in \Sigma^+$ with $w^{-1} = u \cdot v$ and $u \cdot v^\omega \in L(N)$.*

Proof. Let $N = (\Sigma, Q, Q_0, \delta, F)$ be an unambiguous Büchi automaton. We define a nondeterministic finite automaton $A = (\Sigma, Q', Q'_0, \delta', F')$ as follows

- $Q' = (Q^{|N|} \times \{0, \ldots, |N| - 1\}) \times Q^{|N|} \times \{\bot, \top\}^{|N|} \times \{\bot, \top\}$.
  A state (q, ℓ, q', t, b) is composed of a tuple $q = (q_1, \ldots, q_{|N|})$ of $|N|$ states

from Q representing the guessed states at the end of each iteration in the representation of a run of a lasso. The position ℓ defines the iteration between which an accepting state of N has to be observed. The tuple $q' = (q_1, \ldots, q_{|N|})$ defines the tuple of states reached from the tuple q. The tuple t states whether an accepting state has been observed along an iteration. Finally, the flag b, determines whether a shifted tuple of q has been seen up to reaching this state.

- $Q'_0 = \{((q, \ell), q, \bot^{|N|}, \bot) \mid q \in Q^{|N|}, 0 \leq \ell < h < |N|, q(\ell) = q(h)\}$
  A guessed initial tuple must have two positions i and j with equal states, otherwise it does not represent a loop in the run of a lasso.

- $F' = \{((q, \ell), q', v, b) \mid q'(0) \in Q_0,\ b = \top\}$
  A word is accepted, if the shifted initially guessed tuple and an accepting loop has been seen along the way.

- $\delta' : Q' \times \Sigma \to 2^{Q'}$.
  $\delta'((q, \ell, q', t, b), \alpha) = \{(q, \ell, q'', t', b') \mid \forall 1 \leq i \leq |N|.\ q'(i) \in \delta(q'(i), \alpha) \wedge t'(i) = t(i) \vee q''(i) \in F,$
  $b' = b \vee$
  $\forall \ell \leq j < |N| - 1.\ q(j) = q''(j+1)$
  $\exists \ell \leq j < |N|.\ t'(j)$
  $\}$

□

If we apply the last construction to an unambiguous automaton, then each representation of a lasso-run will have a unique run in A. Based on this fact, in the following, we present an algorithm for solving $\#_{Lasso}(L(N), n)$ for an unambiguous Büchi automaton N. The procedure is given in Algorithm 3.4.

The algorithm starts by expanding the set of states of A to ones that additionally count the number of repetitions of the initially guessed tuple of states, given by the set $Q' = Q \times \{0, \ldots, n\}$. If a word reaches a state $(q, i) \in Q'$ then the word allows i lassos to be defined over that word. In each iteration i of the for-loop in line 8, the algorithm computes for each state in Q' the number of words of length i that reach this state starting from an initial state with the same guessed tuples of states. These values are stored in the mapping Ω and are computed by summing up the values of the predecessor states. When the process is finished, the total number of lassos can be computed by summing up the values

of the initial states, each multiplied by the number c of repetitions of the initially guessed tuple.

Algorithm 3.4 An Automata-based Algorithm for Counting Lassos

Input: Unambiguous Büchi automaton N, $n \in \mathbb{N}$
Output: $\#_{Lasso}(L(N), n)$
Begin:
Construct $A = (\Sigma, Q, Q_0, \delta, F)$ for N as in Theorem 3.19
$Q' := Q \times \{0, \ldots, n\}$
$\Omega = \{((q, 0), 1) \mid q \in Q_0\}$
$\Omega' = \emptyset$
for $i := 1, i \leq n, i++$ **do**
  for $q \in Q'$ **do**
    for $\alpha \in \Sigma$ **do**
      for $q' \in \delta(q, \alpha)$ **do**
        for $c \in \{1, \ldots, n\}$ **do**
          let $(\overrightarrow{q_1}, \ell, \overrightarrow{q_2}, \overrightarrow{t}, b) = q'$
          if $\forall \ell \leq j < |N|.\ \overrightarrow{q_1}(j) = \overrightarrow{q_2}(j+1) \wedge b$ **then**
            $\Omega'(q', c) := \Omega'(q', c) + \Omega(q, c-1)$
          else
            $\Omega'(q', c) := \Omega'(q', c) + \Omega(q, c)$
  $\Omega = \Omega'$
  $\Omega' = \emptyset$
return $\sum_{(q_0, c) \in Q_0 \times \{0, \ldots, n\}} \Omega(q_0, c) \cdot c$
End.

The complexity of the algorithm is summarized in the next theorem.

Theorem 3.20. *For an unambiguous Büchi automaton N, and a bound $n \in \mathbb{N}$, the problem of computing the value $\#_{Lasso}(L(N), n)$ can be solved in time linear in n and exponential in $|N|$.*

The complexity for nondeterministic Büchi automata or LTL formulas follows from translating the nondeterministic automaton and the LTL formula to equivalent unambiguous Büchi automata (See Theorem 2.4 and Theorem 2.7).

Corollary 3.8. *For a nondeterministic Büchi automaton N, and a bound $n \in \mathbb{N}$, the problem of computing the value $\#_{Lasso}(L(N), n)$ can be solved in time linear in n and double-exponential in $|N|$.*

Corollary 3.9. *For an* LTL *formula φ, and a bound $n \in \mathbb{N}$, the problem of computing the value $\#_{Lasso}(L(\varphi), n)$ can be solved in time linear in n and double-exponential in $|\varphi|$.*

3.4.2 Complexity bounds

Theorem 3.21. *For a deterministic parity automaton D, and a bound $n \in \mathbb{N}$, the problem of computing $\#_{Lasso}(L(D), n)$ is* #L*-hard and in* #P*.*

Proof. The lower bound can be proven using the same proof for the lower bound of Theorem 3.6. To prove the upper bound we can build a nondeterministic polynomial-time Turing machine that works as follows. The machine guesses a lasso of length n by guessing a stem and period of appropriate size. It then checks whether the lasso is accepted by the automaton, which can be done in polynomial time in the size of the lasso and the automaton. □

Theorem 3.22. *For a nondeterministic parity automaton N, and a bound $n \in \mathbb{N}$, the problem of computing $\#_{Lasso}(L(N), n)$ is* #P*-complete.*

Proof. As for deterministic parity automata, we can construct a nondeterministic polynomial-time Turing machine that guesses a lasso and checks whether it is accepted by N. To prove a lower bound we can encode a DNF formula into a nondeterministic parity automaton as we did in the proof for Theorem 3.7. We choose the bound to be equal to the size of the automaton, which is polynomial in the size of the DNF formula. □

Theorem 3.23. *For an* LTL *formula φ, and a bound $n \in \mathbb{N}$, the problem of computing $\#_{Lasso}(L(\varphi), n)$ is* #P*-complete.*

Proof. The upper bound follows from the fact that we can encode the problem into a propositional formula as we have presented above. The lower bound can be shown by encoding a nondeterministic polynomial-time Turing machine in an LTL formula as we did in the proof for Theorem 3.8, this time with id's up to a polynomial in the size of the Turing machine. We choose the bound to be equal to the maximum id value. □

3.5 Projected Model Counting

We define the projected model counting problem over linear-time properties defined over an alphabet 2^{AP} for a set AP of atomic propositions as follows

Definition 3.5 (Projected Model Counting)**.** Given a linear-time property φ over an alphabet $\Sigma = 2^{AP}$ for a set of atomic propositions AP, and given set $X, Z \subseteq AP$ such that $X \cup Z = AP$, the projected model counting problem $\#^{proj}(\varphi, X)$ is to compute the value

$$|\{\sigma_X \in (2^X)^\omega \mid \exists \sigma_Z \in (2^Z)^\omega.\, \sigma_X \cup \sigma_Z \in L(\varphi)\}|,$$

where $\sigma_1 \cup \sigma_2 = \{\sigma \mid \forall i \in \mathbb{N}.\, \sigma(i) = \sigma_1(i) \cup \sigma_2(i)\}$ is the point-wise union of σ_1 and σ_2.

Every projected model counting problem for an omega-regular property φ defined over an alphabet 2^{AP} for some atomic proposition set AP, and for a projection set $X \subseteq AP$, can be reduced to a model counting problem by projecting an automaton representing the property to another automaton over the alphabet 2^X. If the automaton of the property is nondeterministic, then the projection automaton is also nondeterministic, and thus the complexities of the projected model counting problems for traces, bad prefixes, good prefixes, and lasso are equal to the complexities of the model counting problems for nondeterministic parity automata. In case the property is represented by a deterministic parity automaton, the the projection automaton becomes nondeterministic, und thus the complexities for deterministic parity automata for the different problems is equal to those of nondeterministic automata. For the case of LTL formula, we only face a blow in the complexity for the counting algorithm. This blow-up results from the fact that projecting the unambiguous automaton necessary for computing the number of traces is a nondeterministic automaton that first needs to be determinized again.

We state the complexities of the projected model counting problem for omega-regular properties presented by deterministic parity automata and LTL formulas in Table 3.3. The difference to the complexities of model counting for the same representations are marked in red.

3.6 Maximum Model Counting

We define the maximum model counting problem over linear-time properties defined over an alphabet 2^{AP} for a set AP of atomic propositions as follows

Definition 3.6 (Maximum Model Counting)**.** Given a linear-time property

DPA D	Time complexity	Space complexity
Traces	$2^{O(poly(\|D\|))}$	$O(poly((\|D\|))$
Bad prefixes (n)	$2^{O(poly(\|D\|))} \cdot n$	$2^{O(poly(\|D\|))} \cdot n$
	$\|D\| \cdot 2^{O(n)}$	$O(\log(\|D\|) + n)$
Good prefixes (n)	$2^{O(poly(\|D\|))} \cdot n$	$2^{O(poly(\|D\|))} \cdot n$
	$2^{O(poly(\|D\|)+n)}$	$O(poly(\|D\|) + n)$
Lassos (n)	$2^{2^{O(poly(\|D\|))}} \cdot n$	$2^{2^{O(poly(\|D\|))}} \cdot n$
	$\|D\| \cdot 2^{O(n)}$	$O(\|D\| + n)$

LTL φ	Time complexity	Space complexity
Traces	$2^{2^{O(poly(\|\varphi\|))}}$	$2^{O(poly(\|\varphi\|))}$
Bad prefixes (n)	$2^{2^{O(\|\varphi\|)}} \cdot n$	$2^{2^{O(\|\varphi\|)}} \cdot n$
	$2^{O(\|\varphi\| \cdot n)}$	$O(n + poly(\|\varphi\|))$
Good prefixes (n)	$2^{2^{O(poly(\|\varphi\|))}} \cdot n$	$2^{2^{O(poly(\|\varphi\|))}} \cdot n$
	$2^{O(\|\varphi\| \cdot n)}$	$O(poly(\|\varphi\|) + n)$
Lassos (n)	$2^{2^{O(poly(\|\varphi\|))}} \cdot n$	$2^{2^{O(poly(\|\varphi\|))}} \cdot n$
	$\|\varphi\| \cdot 2^{O(n)}$	$O(poly(\|\varphi\|) + n)$

Table 3.3: Complexity of projected model counting.

over an alphabet $\Sigma = 2^{AP}$ for a set of atomic propositions AP, and given sets $X, Y, Z \subseteq AP$, such that, $AP = X \cup Y \cup Z$, the maximum model counting problem $\#^{\max}(\varphi, X, Y, Z)$ is to compute the value

$$\max_{\sigma_Y \in (2^Y)^\omega} |\{\sigma_X \in (2^X)^\omega \mid \exists \sigma_Z \in (2^Z)^\omega.\, \sigma_X \cup \sigma_Y \cup \sigma_Z \in L(\varphi)\}|,$$

where $\sigma_1 \cup \sigma_2 = \{\sigma \mid \forall i \in \mathbb{N}.\, \sigma(i) = \sigma_1(i) \cup \sigma_2(i)\}$ is the point-wise union of σ_1 and σ_2.

For omega-regular properties, we solve the maximum model counting in two steps. For a maximum model counting problem $(L(P), X, Y, Z)$ for some parity automaton P, we first check if there is a sequence $\sigma_X \in (2^X)^\omega$ for which there exists an infinite number of sequences $\sigma_Y \in (2^Y)^\omega$, for which in turn there exist a $\sigma_Z \in (2^Z)^\omega$ such that $\sigma_X \cup \sigma_Y \cup \sigma_Z \in L(P)$. This can be done with a doubly-pumped lasso analysis as for model counting with the difference that

this time we need to find a doubly-pumped lasso (u, v, u', v') in P such that $trace(u \cdot v^\omega)_{|X} = trace(u' \cdot v'^\omega)_{|X}$, and $trace(u \cdot v^\omega)_{|Y} \neq trace(u' \cdot v'^\omega)_{|Y}$. This is necessary, because we are maximizing the counts over sequence of valuations over X. Finding such a lasso can be done in polynomial time in the size of the automaton P, by modifying Algorithm 3.1 to look for a transition with labels (α_1, α_2) such that $(\alpha_1)_{|Y} \neq (\alpha_2)_{|Y}$, and loop in the automaton that includes this transition, and that additionally does not differ at any transition in the loop on the valuations of X. As for model counting, if the number of sequences from $\sigma_Y \in (2^Y)^\omega$ exceeds the exponential threshold, then the number of such sequences is equal to infinity.

Lemma 3.2. *Let a parity automaton P over a set of atomic propositions AP and sets $X, Y \subseteq AP$. For each sequence $\sigma_X \in (2^X)^\omega$ the number of sequences $\sigma_{X\cup Y} \in (2^{X\cup Y})^\omega$ that are projections of a model $\sigma \in L(P)$ with $(\sigma_{X\cup Y})_{|X} = \sigma_{|Y}$ is less or equal than $2^{|P|\cdot\log(|P|)}$, otherwise it is ∞.*

If the number of sequence σ_Y is not equal to infinity for any sequence σ_X, then we can solve the maximum model counting problem in time exponential in the automaton P. The procedure is given in Algorithm 3.5

Algorithm 3.5 Maximum Model Counting

Input: $B = (Q, q_0, 2^{AP}, \delta, F)$, disjoint $X, Y, Z \subseteq AP$, $n \in \mathbb{N}$
Output: $\#_{X,Y,Z}(B) > n$
$SCC = acceptingSCC(B)$
$i = 1$
$W = \bigcup_{S\in SCC} S$
while $i \leq |Q|$ **do**
  $i = i + 1$
  for $q \in W$ **do**
    for (q', α, q) **do**
      $W' = W' \cup \{q'\}$
      for $\sigma \in \Pi(q)$ **do**
        $\Pi'(q') = \Pi'(q') \cup \{\alpha_{Y\cup X} \cdot \sigma\}$
  $W = W'$
  $W' = \emptyset$
  for $q \in W$ **do**
    $\Pi'(q) = \emptyset$
return $\max_{X,Y,Z} \Pi(q_0) \geq n$

From Lemma 3.2, we know that if no sequence $\sigma_Y \in (2^Y)^\omega$ matches to infinitely many $X \cup Y$-projected models then the number of such models is bound

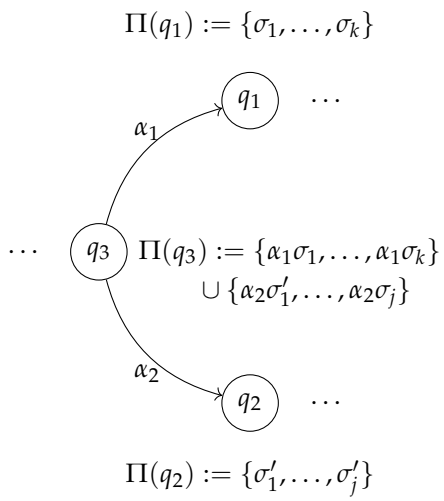


Figure 3.7: A Sketch of a step in this algorithm: Current elements of our working set are $q_1, q_2 \in W$ and $q_3 \in W'$. If $i = 0$, i.e., we are in the first step of the algorithm, then q_1 and q_2 are states of accepting SCCs.

by $2^{|Q|}$. Each of these models has a run in B which ends in an accepting loop that are not in a doubly-pumped lasso with the conditions mentioned above, in at most $|Q|$ steps. For each finite sequence w_Y of length $|w_Y| = |Q|$ that reaches an accepting loop, we count the number $X \cup Y$-projected words w of length $|Q|$ with $w =_Y w_Y$ and that end in an accepting loop. This number is equal to the maximum model counting number.

The algorithm works in a backwards fashion starting with states of accepting loops. In each iteration i, the algorithm maps each state of the automaton with $X \cup Y$-projected words of length i that reach an accepting strongly connected component. After $|Q|$ iterations, the algorithm determines from the mapping of initial state q_0 a Y-projected word of length $|Q|$ with the maximum number of matching $X \cup Y$-projected words. Figure 3.7 shows how the mapping is computed for a state at each iteration.

Theorem 3.24. *For an omega-regular property φ defined over 2^{AP} for a set of atomic propositions AP, and for sets $X, Y, Z \subseteq AP$ with $X \cup Y \cup Z = AP$, if $\#^{\max}(\varphi, X, y, Z) < \infty$, then solving $\#^{\max}(\varphi, X, Y, Z)$ can be done in time exponential in the size of a non-deterministic parity automaton representing φ.*

3.7 Proofs

Proof of Lemma 3.1

Let $N = (\Sigma, Q, Q_0, \delta, \mu)$ be a non-deterministic parity automaton. The number of words in $L(N)$ is equal to infinity if and only if N has an initial accepting doubly-pumped lasso (u_1, v_1, u_2, v_2) with $trace(u_1 \cdot (v_1)^\omega) \neq trace(u_2 \cdot (v_2)^\omega)$.

Proof.

- $\Rightarrow$: We prove this directions in two steps. In the first step we show that the existence of an initial doubly-pumped lasso is a necessary condition for $L(N) = \infty$. In the second step we show that one doubly-pumped lasso (u, v, u', v') must fulfill the condition $trace(u \cdot (v)^\omega) \neq trace(u' \cdot (v')^\omega)$, in order to represent an infinite number of words.

  Step 1: Let $\sigma = \alpha_1 \cdot \alpha_2 \cdots \in L(N)$, and let $\rho = (q_0, \alpha_1) \cdot (q_1, \alpha_2) \ldots$ be a run for σ in N. Because N has finitely many states, we know that there are indices $0 \leq i < j \in \mathbb{N}$ such that $q_i = q_j$, $\mu(q_j) = \max\{\mu(q_h) \mid h \in \mathbb{N}\}$, and $\mu(q_j)$ is even. This in turn means that N has an accepting lasso (u, v), where $u = (q_0, \alpha_1) \cdot (q_1, \alpha_2) \cdots (q_{i-1}, \alpha_i)$, and $v = (q_i, \alpha_{i+1}) \cdot \cdots \cdot (q_{j-1}, \alpha_j)$. Without the loss of generality, we assume that there is no position $\ell \in \{0, \ldots, i-1, i+1, \ldots, j-1\}$ with $q_\ell = q_j$. We distinguish two cases. In the first case, we assume that $j - i > |Q| + 1$. This means that there are two further positions $i < i' < j' < j$ such that $q_{i'} = q_{j'}$. It follows that N has an initial accepting doubly-pumped lasso (u', v', u, v), where $u' = (q_0, \alpha_1) \cdot (q_1, \alpha_2) \cdots (q_{i'-1}, \alpha_{i'})$, and $v' = (q_{i'}, \alpha_{i'+1}) \cdot \cdots \cdot (q_{j'-1}, \alpha_{j'})$. In the second case, where $j - i < |Q| + 1$, we further distinguish the following two cases. If $|u \cdot v| > |Q|$, then there must two positions $i' < j' < j$ such that $q_{i'} = q_{j'}$, and again N has an initial accepting doubly-pumped lasso. If $|u \cdot v| \leq |Q|$, and N has no initial accepting doubly-pumped lasso, then N accepts only a finite number of traces, as the number of lassos of length $|Q|$ is finite. This however contradicts the assumption that $|L(N)| = \infty$.

  Step 2: Assume that for all doubly-pumped lassos $\rho = (u, v, u', v')$ in N we have $trace(u \cdot (v)^\omega) = trace(u' \cdot (v')^\omega)$. We show that under this assumption the languages represented by the doubly-pumped lasso would be a singleton. Assume two different infinite words are in the language of ρ, then there two points reachable in ρ by words of length n that differ in the transition label. This however contradicts the assumption that all words of length n reachable in ρ are equal.

- $\Leftarrow$: Let (u, v, u', v') be a doubly-pumped lasso in N with $\mathit{trace}(u \cdot v^\omega) \neq \mathit{trace}(u' \cdot v'^\omega)$. The number of different accepting runs in N is at least as large as $\Gamma = \{\rho \mid \rho \in u \cdot v^* \cdot w \cdot v'^\omega\}$, where w is the sequence of states in N such that the last state of $u \cdot v \cdot w$ is a predecessor of the first state of v'. Such a sequence must exist, because any state in v' is reachable from any state in v'. Because $u \cdot v^\omega \neq u' \cdot v'^\omega$, the size of Γ is infinitely large.

□

Proof of Theorem 3.4

There is a family of LTL *formulas* φ_n *for* $n \in \mathbb{N}$ *with* $|\varphi_n| \in O(n)$ *and* $|L(\varphi_n)| \in 2^{2^n \cdot \log(n)}$.

Proof. Consider the following omega-regular property over the alphabet 2^{AP} where $AP = \{b_0, \ldots, b_{n-1}, e\}$

$$\begin{aligned} L_n = \{\sigma \in (2^{AP})^\omega \mid\ & \forall 0 \leq i < 2^n.\ \mathit{binary}(b_0, \ldots, b_{n-1}, i) \subset \sigma(2 \cdot i), \\ & \forall i \in \mathbb{N}.\ (e \in \sigma(i)) \leftrightarrow (i \text{ is even}), \\ & \forall i \geq 2^n.\ \sigma(i) \cap \{b_0, \ldots, b_{n-1}\} = \emptyset\} \end{aligned}$$

where $\mathit{binary}(b_0, \ldots, b_{n-1}, i)$ defines a set $X \subset \{b_0, \ldots, b_n\}$ such that X resembles the binary encoding of i with b_0 and b_n being the first and the last bit, respectively. A language L_n defines all words the represent a counter that count to 2^n and which values are defined in the odd positions of the word. between two values of the counter, any evaluation of the propositions $\{b_0, \ldots, b_n\}$ is allowed. After the counter reaches its maximum values the counter is reset to 0 and remains 0 forever. The size of a language L_n is equal to $n^{2^n} = 2^{2^n \cdot \log(n)}$.

Each property L_n can be defined by an LTL formula

$$\varphi_n = \varphi_{\mathit{init}} \wedge \varphi_{\mathit{even}} \wedge \varphi_{\mathit{count}}$$

where

- $\varphi_{\mathit{init}} = \neg b_0 \wedge \ldots, \wedge \neg b_n \wedge \neg e$. At the initial position of each word of φ_n the counter is set to 0.
- $\varphi_{\mathit{even}} = \overline{e} \wedge \Box(e \leftarrow \bigcirc \overline{e})$ is an auxiliary formula to identify the even and odd positions of a word. The counter is defined over the odd positions of a word, i.e., if at some odd position i the counter values defines the number c, then the values of the bits $b_0, \ldots, b_n$ at position $i + 2$ define the number $c + 1$.

- $\varphi_{count} = \square(\bar{e} \rightarrow Inc(b_0, \ldots, b_{n-1}, 2))$ where

$$\begin{aligned} Inc(b_1, \ldots, b_\ell, d) = (\bigvee_{0<i\leq\ell} \neg b_i) \rightarrow \bigwedge_{0<i\leq\ell} \big[((\neg b_i \wedge \bigwedge_{i<j\leq\ell} b_j) \rightarrow \bigcirc^d b_i) \\ \wedge ((\neg b_i \wedge \neg \bigwedge_{i<j\leq\ell} b_j) \rightarrow \bigcirc^d \neg b_i) \\ \wedge ((b_i \wedge \bigwedge_{i<j\leq\ell} b_j) \rightarrow \bigcirc^d \neg b_i) \\ \wedge ((b_i \wedge \neg \bigwedge_{i<j\leq\ell} b_j) \rightarrow \bigcirc^d b_i) \big] \end{aligned}$$

  which defines the operation of incrementing a binary sequence by one.

The size of the formula φ_n is polynomial in n. □

Proof of Theorem 3.5

Let P be an unambiguous parity automaton, and let $|L(P)| < \infty$. Computing the number of models of $L(P)$ can be done in time polynomial in the size of P.

Proof. We proof the following strengthening of the problem. Given a set $Q_\iota \subseteq Q$ and a bound $n \in \mathbb{N}$, computing the number of finite sequences r in $(Q \times \Sigma)^n$ with $proj(1, r(0)) \in Q_\iota$ that reach an accepting loop in P can be done in time polynomial in n and in the size of P.

The proof of the strengthening follows by proving the correctness of Algorithm 3.2, which we will do by induction over n.

- $n = 0$: In this case, the set $(Q \times \Sigma)^0 = \{\epsilon\}$, and thus number of sequences r in $(Q \times \Sigma)^n$ with $proj(1, r(0)) \in Q_\iota$ that reach an accepting loop in P is equal to the number of states q_ι that are in an accepting loop in the automaton. Since $n = 0$, the automaton will only mark the states according to lines 9-13 and return the number of states in Q_ι marked with 1.

- $n \rightarrow n+1$: Assume that each state in Q is marked with the correct number of sequences r in $(Q \times \Sigma)^n$ that reach an accepting loop in P. The number of sequences r in $(Q \times \Sigma)^{n+1}$ that reach an accepting loop in P, can be computed as follows. Let $q \in Q$, and $q' \in Q$ such that $(q, \alpha, q') \in delta$ for some $\alpha \in \Sigma$. If r is sequence r in $(Q \times \Sigma)^n$ with $proj(1, r(0)) = q'$ that reaches an accepting loop in P, then $(q, \alpha) \cdot r$ is a sequence r in $(Q \times \Sigma)^{n+1}$ with $proj(1, r(0)) = q$ that reaches an accepting loop in P. Thus, the

number of sequences $(q, \alpha) \cdot r$ in $(Q \times \Sigma)^{n+1}$ that reach an accepting loop in P is equal to the number of sequences r in $(Q \times \Sigma)^n$ with $proj(1, r(0)) \in succ(q)$ that reach an accepting loop in P, where $succ(q)$ are all successor states of q.

If $Q_\iota = Q_0$ then all sequences r in $(Q \times \Sigma)^n$ with $proj(1, r(0)) \in Q_\iota$ that reach an accepting loop in P is equal to the number of accepting runs of P. Since P is unambiguous, this number is equal to the number of models of P.

The run-time of Algorithm 3.2 is polynomial in $|P|$ and n. When $n = |P|$, the algorithm is polynomial in the size of P. □

Proof of Theorem 3.6 (Lower Bound)

Let D be a deterministic parity automaton, with $L(D) < \infty$. The problem of computing the number models of D is #L*-complete.*

Proof. We complete the proof of Theorem 3.6 by showing that the problem is #L-hard. Let $M = (Q, q_0, Q_F, \Sigma, \delta)$ be a two-tape nondeterministic logarithmic space Turing machine (a tape for the input, and a tape for the output), where Q is the set of states, q_0 is the initial state, Q_F is the set of accepting states, Σ is the alphabet, and $\delta\colon (Q \setminus Q_F) \times \Sigma \to 2^{Q \times \Sigma \times \{-1,0,1\}}$ is the transition function, where -1 and 1 encode the directions of the head. Note that the accepting states are terminal and that $\mathcal{M}$ rejects by terminating in a nonaccepting state. Let M be $\log(p(n))$-space bounded for some polynomial p, and let $w = w_0 \cdots w_{n-1}$ be an input to M. Let $p'(n)$ be a polynomial (which only depends on M) such that M terminates in at most $p'(n)$ steps on inputs of length n.

We construct a deterministic parity automaton D, which is polynomial in $|w|$ and $|M|$, respectively, such that the number of accepting runs of M on w is equal to the number of models of D. Let $l_c = \lceil \log(p(n)) \rceil$ be the maximal size of a configuration of M on w. The automaton $D = (\Sigma_D, Q_D, q_{0,D}, \delta_D, \mu_D)$ such that

- $\Sigma_D = Q \times \Sigma \times \{-1, 0, 1\} \times \{-1, 0, 1\} \cup \{\$\}$
- $Q_D = Q \times \{0, \ldots, l_c - 1\} \times \{0, \ldots, n\} \times (\Sigma \cup \{\#\})^{l_c} \times \{0, \ldots, p'(n)\}$
  $\cup \{q_\bot, q_\top\}$
- $q_{0,D} = (q_0, 0, 0, \#^{l_c})$
- $\mu : Q_D \to \{0, 1\}.\ q \mapsto \begin{cases} 0 & \text{if } q = q_\top \\ 1 & \text{otherwise} \end{cases}$

- $\delta_D : Q_D \times \Sigma_D \to Q_D$.
  $\delta_D((q'', o, i, s_1, \ldots, s_{l_c}, c), \gamma) =$
  $$\begin{cases} (q', o + \ell_o, i + \ell_i, s_1, \ldots, \alpha, s_{o+1} \ldots, s_{l_c}, c + 1) & \text{if } c < p'(n), q'' \notin Q_F, \\ & \text{and } \gamma = (q', \alpha, \ell_o, \ell_i) \\ q_\top & \text{if } q'' \in Q_F, \text{and } \gamma = \$ \\ q_\top & \text{if } q = q_\top, \text{and } \gamma = \$ \\ q_\bot & \text{otherwise} \end{cases}$$

Each state of the automaton D represents a configuration of M A run in the automaton represents a computation of M. The states $q_\top$ and $q_\bot$ represent the acceptance and rejection of M, respectively. The symbol \$ represents the termination of M. A run of D represents a computation of M. Each accepting computation of M can be matched to a unique run in D, namely the run of the form:

$$(c_0, \gamma_1) \cdot (c_1, \gamma_2) \cdots (c_{p'(n)}, \$) \ (q_\top, \$)^\omega$$

where c_0 is the initial configuration of M, for all $0 < i \leq p'(n)$, c_i respects the transition relation δ of M, and $c_{p'(n)}$ is an accepting configuration of M. The number of words accepted by D are thus equal to the number of accepting runs of M. □

Proof of Theorem 3.7 (Lower Bound)

Let N be a nondeterministic parity automaton, with $L(N) < \infty$. The problem of computing the number models of N is #P-complete.

Proof. Let ϕ be propositional formula in disjunctive normal form with n disjuncts $D_1, \ldots, D_n$ and m variables $x_1, \ldots, x_m$. For $1 \leq i \leq n$, let $\mathrm{Lit}(D_i)$ define the set of all literals in D_i. We define a nondeterministic Büchi automaton $N = (\Sigma, Q, Q_0, \delta, F)$ such that

- $\Sigma = \{x_1, \ldots, x_m, \overline{x_1}, \ldots, \overline{x_m}\}$
- $Q = \bigcup_{1 \leq i \leq n} \{\mathrm{Lit}(D_i), \mathrm{Lit}(D_i) \setminus \{x_1, \overline{x_1}\}, \mathrm{Lit}(D_i) \setminus \{x_1, \overline{x_1}, x_2, \overline{x_2}\}, \ldots, \mathrm{Lit}(D_i) \setminus \{x_1, \ldots, x_m \overline{x_1}, \ldots, \overline{x_m}\}\} \times \{0, \ldots, m\} \cup \{q_\bot\}$
- $Q_0 = \{(\mathrm{Lit}(D_i), m) \mid 1 \leq i \leq n\}$
- $F = \{(\emptyset, 0)\}$

- $\delta : Q \times \Sigma \to 2^Q$.
  - $\forall 1 \leq i \leq m, \delta((X,c), x_i) = \begin{cases} (X \setminus \{x_i\}, c-1) & \text{if } X \neq \emptyset, i = c, \text{and} \\ & x_i \in X \to \overline{x_i} \notin X \\ q_\perp & \text{otherwise} \end{cases}$
  - $\forall 1 \leq i \leq m, \delta((X,c), \overline{x_i}) = \begin{cases} (X \setminus \{\overline{x_i}\}, c-1) & \text{if } X \neq \emptyset, i = c, \text{and} \\ & \overline{x_i} \in X \to x_i \notin X \\ q_\perp & \text{otherwise} \end{cases}$
  - $\delta((\emptyset,c),\alpha) = \begin{cases} (\emptyset, c-1) & \text{if } c > 0, \alpha = x_c \\ (\emptyset, 0) & c = 0, \alpha = x_1 \\ q_\perp & \text{otherwise} \end{cases}$
  - $\delta(q_\perp, \alpha) = q_\perp$

The automaton resembles the algorithm for solving DNF formulas using the order $x_1, \ldots, x_m$. Each initial state represents a disjunction of the DNF formula. The sequence of letters reaching the state $(\emptyset, 0)$ from one of the initial states, represents a solution for the disjunct defined by that initial state, and thus represents also a solution for the DNF formula. Each word accepted by the automaton represents a unique satisfying assignment for the DNF formula, and vice versa. Figure 3.8 shows the construction for the formula □

Proof of Theorem 3.8 (Lower Bound)

Let φ be an LTL *formula, with $L(\varphi) < \infty$. The problem of computing the number models of φ* #PSPACE*-complete.*

Proof. We complete the proof of Theorem 3.8 by showing that the problem is #PSPACE-hard. Let $\mathcal{M} = (\Sigma, Q, q_\iota, Q_F, \delta)$ be a one-tape nondeterministic polynomial-space Turing machine, where Q is the set of states, q_ι is the initial state, Q_F is the set of accepting states, Σ is the alphabet, and $\delta\colon (Q \setminus Q_F) \times \Sigma \to 2^{Q \times \Sigma \times \{-1,0,1\}}$ is the transition function, where -1,0 and 1 encode the directions of the head, left, stay, and right, respectively. Note that the accepting states are terminal and that $\mathcal{M}$ rejects by terminating in a nonaccepting state. Let M be $p(n)$-space bounded for some polynomial p, and let $w = w_0 \cdots w_{n-1}$ be an input to M. Let $p'(n)$ be a polynomial (which only depends on M) such that M

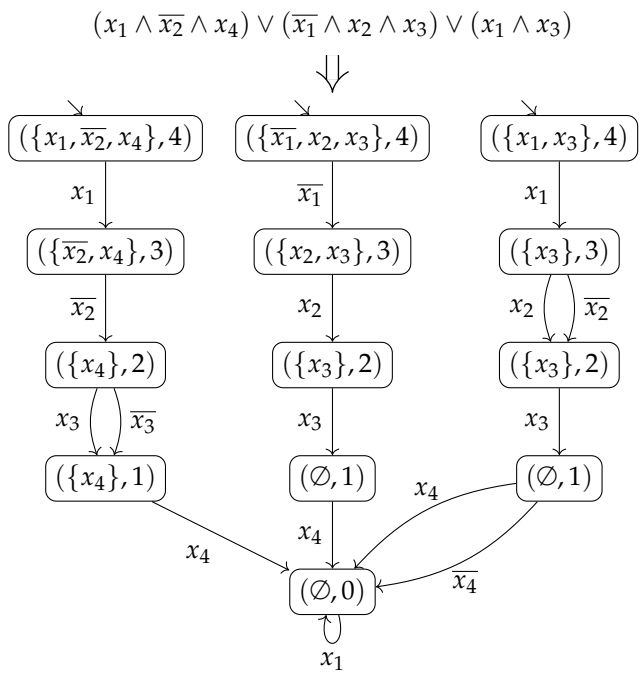


Figure 3.8: A reduction from a propositional formula in disjunctive normal form to a nondeterministic Büchi automaton.

terminates in at most $2^{p'(n)}$ steps on inputs of length n. We construct an LTL formula φ_M^w, which is polynomial in $|w|$ and $|M|$ such that the number of accepting runs of M on w is equal to the number of models of φ_M^w.

A run of M on w is encoded by a finite alternating sequence of id's id_i and configurations c_i that is followed by an infinite repetition of a dummy symbol:

$$\$\ \mathrm{id}_0\ \#\ c_0\ \$\ \mathrm{id}_1\ \#\ c_1\ \$\ \mathrm{id}_2\ \#\ c_2\ \$\ \cdots\ \$\ \mathrm{id}_{2^{l_c}}\ \#\ c_{2^{l_c}}\ (\bot)^\omega \tag{3.4}$$

for $l_c = p'(n)$. The runs of M on w are encoded in the prefixes of models up to the symbol $\bot$, and such that the only possible suffix of models is $\bot^\omega$, ensuring that exactly 2^{l_c} configurations are encoded (by repeating the final configuration if necessary). This ensures that an accepting run is encoded by exactly one model.

Let $l_r = p(n)$ be the maximal size of a configuration of M on w. For the id's we use an encoding of a binary counter with $l_c = p'(n)$ many bits. Let $AP = (Q \cup \Sigma) \uplus \{b_1, \ldots, b_{l_c}, \$, \#, \bot\}$ be the set of atomic propositions. The atomic propositions in $Q \cup \Sigma$ are used to encode the configurations of M by encoding the tape contents, the state of the machine, and the head position. The atomic

propositions $b_1, \ldots, b_{l_c}$ represent the bit values of an id. The symbols \$ and # are used as separators between id's and configurations, and $\perp$ is a dummy symbol for the model's period. The distance between two \$ symbols and also between two # symbols in the encoding is given by $d = l_r + 3$. Then, φ_M^w is the conjunction of the following formulas:

- *Id* encodes the id's of the configurations. It uses a formula $Inc(b_1, \ldots, b_{l_c}, d)$ that asserts that the number encoded by the bits b_j after d steps is obtained by incrementing the number encoded at the current position.
- *Init* asserts that the run of M starts with the initial configuration.
- *Accept* asserts that the run must reach an accepting configuration.
- *Config* declares the consistency of two successive configurations with the transition relation of M. Here, we use d many next operators to relate the encoding of the two configurations.
- *Repeat* asserts that the encoding of an accepting configuration is repeated until the maximum id is reached
- *Loop* defines the period of the word-model, which may only contain $\perp$.

We show that all these properties can be expressed with polynomially-sized formulas. Furthermore, we need a formula to specify technical details: atomic propositions encoding the id's are not allowed to appear in the configurations and vice versa, symbols such as \$ and # only appear as separators, each separator appears $2^{p'(n)}$ times every d positions, configuration encodings are represented by singleton sets of letters in Σ with the exception of one set that contains a symbol from Q to determine the head position and the state of M, etc.

We start with the formula *Id*, which uses the formula *Inc* that enforces an increment of a binary counter. The formula *Inc* is parameterized by the propositions $b_1, \ldots, b_\ell$ encoding the bits (b_1 being the most significant one) and the distance d between the two positions to be compared. Intuitively, the different subformulas distinguish whether the increment ripples through to the current bit b_i or not. Note that the increment property only has to hold if there is no overflow of the counter.

$$\begin{aligned} Inc(b_1,\ldots,b_\ell,d) = (\bigvee_{0<i\le\ell} \neg b_i) \rightarrow \bigwedge_{0<i\le\ell} \big[\, ((\neg b_i \wedge \bigwedge_{i<j\le\ell} b_j) \rightarrow \bigcirc^d\ b_i) \\ \wedge((\neg b_i \wedge \neg \bigwedge_{i<j\le\ell} b_j) \rightarrow \bigcirc^d \neg b_i) \\ \wedge((\ b_i \wedge \bigwedge_{i<j\le\ell} b_j) \rightarrow \bigcirc^d \neg b_i) \\ \wedge((\ b_i \wedge \neg \bigwedge_{i<j\le\ell} b_j) \rightarrow \bigcirc^d\ b_i) \,\big] \end{aligned}$$

Now, the formula *Id* is defined by initializing the counter to zero and always requiring an increment after a \$-separator:

$$Id = \$ \ \wedge\ \bigcirc(\bigwedge_{0<j\le l_c} \neg b_j) \ \wedge\ \Box(\$ \rightarrow \bigcirc Inc(b_1,\ldots,b_{l_c},d))$$

We continue with the formula *Init*. In the initial configuration the tape of M contains the input word w, the head is on the first cell, and M is in its initial state:

$$Init = \bigcirc^2(\# \ \wedge\ \bigcirc q_\iota \ \wedge\ (\bigwedge_{0\le j<n} \bigcirc^j w_j) \ \wedge\ (\bigwedge_{n\le j\le l_r} \bigcirc^j \text{␣}))$$

The symbol ␣ refers to the blank cells of the tape.

The formula *Accept* considers the maximal id and checks whether it is followed by an accepting configuration:

$$Accept = \Box(\$ \wedge \bigcirc(\bigwedge_{0<j\le l_c} b_j) \ \rightarrow \bigvee_{q\in Q_F} \bigvee_{0<j\le l_r} \bigcirc^{j+2} q)$$

For atomic propositions $q \in Q \setminus Q_F$ and $\alpha \in \Sigma$ a formula $config_{q,\alpha}$ asserts the relation between the states, the head positions, and the content of the cell where the head is pointing to in two successive configurations:

$$config_{q,\alpha} = \Box(\ q \wedge \alpha \rightarrow \bigvee_{(q',\beta,\mathrm{dir})\in\delta(q,\alpha)} \bigcirc^d \beta \wedge \bigcirc^{d+\mathrm{dir}} q'\ )$$

Another formula $config_\alpha$ asserts the relation of the other tape cells of successive configurations; the content of these cells is copied, unless the id is maximal:[1]

$$config_\alpha = \Box(\$ \wedge \bigcirc(\bigvee_{0<j\le l_c} \neg b_j) \rightarrow \bigwedge_{0<j\le l_r} \bigcirc^{j+2}((\bigwedge_{q\in Q\setminus Q_F} \neg q) \wedge \alpha \rightarrow \bigcirc^d \alpha))$$

[1] Note that this is not necessary for $config_{q,\alpha}$ since the machine terminates in at most $p(n)$ steps

Config is the conjunction of all formulas $config_\alpha$ and $config_{q,\alpha}$.

The formula *Repeat* requires an accepting configuration to be repeated if the id is not yet maximal. The repetition of the letters is taken care of by the formulas $config_\alpha$. Hence, *Repeat* only requires to copy the state and the head position.

$$Repeat = \square(\$ \wedge (\bigcirc \bigvee_{0<j\leq l_c} \neg b_j) \rightarrow \bigwedge_{q_f \in Q_F} \bigwedge_{0<j\leq l_r} \bigcirc^{j+2}(q_f \rightarrow \bigcirc^d q_f))$$

Finally, the period of the model is fixed by the formula *Loop* which requires the symbol $\perp$ to be repeated after reaching the configuration with the maximal id:

$$Loop = \square(\$ \wedge \bigcirc(\bigwedge_{0<j\leq l_c} b_j) \rightarrow \bigcirc^{l_r+3} \square\perp)$$

Each accepting run of $\mathcal{M}$ on w corresponds to exactly one model of $\varphi^w_{\mathcal{M}}$ that encodes the run in its prefix of length $2^{l_c} \cdot (l_r + 3)$. Thus, the number of models is equal to the number of accepting runs of $\mathcal{M}$ on w. The formula $\varphi^w_{\mathcal{M}}$ can be obtained in polynomial time in $|w| + |\mathcal{M}|$. □

Chapter 4

The Relation of Model Counting to Probabilistic Model Checking

Implementing a reactive system that satisfies all the guarantees on all inputs of the environment is not always possible. However, in many cases, certain input scenarios of the environment are less likely to happen. Given a probabilistic model on the occurrence of inputs of the environment, we are able to better assess how reliable a system is and determine the likelihood of the system failing to fulfill the requirements posed by the specification.

In this chapter, we investigate the problem of computing the probability of satisfying a linear-time property by a transition system. Transition systems can be viewed as probabilistic systems, with a uniform probability distribution over the inputs, i.e., each transition occurs with equal probability. We show that using model counting, we can provide lower and upper bounds on the probability of satisfying the property, which allows us to check whether the satisfaction of the property is within some certain predefined thresholds.

We start the chapter with a brief recap on the model checking problem for probabilistic systems modeled as *discrete-time Markov chains* [95] and show that model counting coincides with the probability measures defined for Markov chains, provided a uniform probability distribution on the inputs.

4.1 Probabilistic Model Checking

To model the probabilistic behavior of environments and systems, transition systems are enhanced with probabilities. The enhancement can be done in different ways [13]. In Markov chains [95], each choice in the system, i.e., each transition, is assigned the probability of its input. Other models, such as *Markov decision processes* [18], additionally allow nondeterministic choices that model the randomized interleaving behavior of systems.

In this chapter, we will focus on probabilistic models based on Markov chains, since transition systems for reactive systems can be defined as Markov chains with a uniform probability function.

Definition 4.1 (Discrete-time Markov Chains [13, 95])**.** For a set of atomic propositions AP, a discrete-time Markov chain over AP is defined by the tuple

$$M = (AP, S, \iota, P, L)$$

where

- S is a countable set of states
- $\iota : S \to [0,1]$ is the initial distribution such that $\sum_{s \in S} \iota(s) = 1$
- $P : S \times S \to [0,1]$ is the transition probability function such that

$$\forall s \in S.\ \sum_{s' \in S} P(s, s') = 1$$

- $L : S \to 2^{AP}$ is a labeling function that maps each state in S to a valuation of AP

A *path* in a Markov chain M is an infinite sequence of states $s_0 s_1 \cdots \in S^\omega$ such that $\iota(s_0) > 0$ and for all $i \geq 0$ we have $P(s_i, s_{i+1}) > 0$. The set $Paths(M)$ defines the set of all paths in M. The set defined as $Paths_{fin}(M) = Prefix(Paths(M))$ is the set of finite paths in M.

To be able to compute the probability of a set of paths Γ in a Markov chain M, the set Γ needs to be measurable, i.e., it needs to be associated with a probability space of M, which defines the set of measurable sets over M.

Definition 4.2 (σ-Algebra and Probability Space [13])**.** A *σ-algebra* is a pair (Γ, Ω) for a nonempty set Γ and a set $\Omega \subseteq 2^\Gamma$ such that

- $\Gamma \in \Omega$,
- if $\Gamma' \in \Omega$, then $\Gamma \setminus \Gamma' \in \Omega$, and
- if $\Gamma_0, \Gamma_1, \dots \in \Omega$, then $\bigcup_{i \in \mathbb{N}} \Gamma_i \in \Omega$

Note that since $\Gamma \in \Omega$, also $\emptyset \in \Omega$.

A *probability measure* on a σ-algebra (Γ, Ω) is a probability function $P : \Omega \to [0, 1]$ such that

- $P(\Gamma) = 1$, and
- if $\Gamma_0, \Gamma_1, \dots \in \Omega$, and for all $i, j \in \mathbb{N}$ with $i \neq j$ it holds that $\Gamma_i \cap \Gamma_j = \emptyset$, then $P(\bigcup_{i \in \mathbb{N}} \Gamma_i) = \sum_{i \in \mathbb{N}} P(\Gamma_i)$

A *probability space* is a tuple $((\Gamma, \Omega), P)$, where (Γ, Ω) is a σ-algebra, and P is a probability measure on (Γ, Ω).

A probability space given by $((\mathit{Paths}(M), \Omega), P')$ for a discrete-time Markov chain $M = (AP, S, \iota, P, L)$ can be defined in terms of *cylinder sets* over $\mathit{Paths}_{\mathit{fin}}(M)$. For $s_0 s_1 \dots s_m \in \mathit{Paths}_{\mathit{fin}}(M)$, a cylinder set $\mathfrak{C}(s_0 s_1 \dots s_m)$ is defined by

$$\mathfrak{C}(s_0 s_1 \dots s_m) = \{\pi \in \mathit{Paths}(M) \mid s_0 s_1 \dots s_m \in \mathit{Prefix}(\pi)\}.$$

The probability space $((\mathit{Paths}(M), \Omega), P')$ for M is then defined such that for $w \in \mathit{Paths}_{\mathit{fin}}(M)$, it holds that $\mathfrak{C}(w) \in \Omega$ [13]. To prove that a set of paths $\Gamma \subseteq \mathit{Paths}(M)$ in a Markov chain M is measurable, we need to show that $\Gamma \in \Omega$. This can be done by showing that the set Γ can be defined as a countable union of disjoint cylinder sets in Ω, i.e., there are finite paths $w_i \in \mathit{Paths}_{\mathit{fin}}(M)$, for $i \in X \subseteq \mathbb{N}$ such that

$$\Gamma' = \bigcup_{i \in X} \mathfrak{C}(w_i).$$

The probability $P(\Gamma')$ can then be computed by computing

$$\sum_{i \in X} P(\mathfrak{C}(w_i))$$

where $P(\mathfrak{C}(\pi_i))$ for $w_i = s_0 s_1 \dots s_m$ and for some $m \in \mathbb{N}$ is defined by

$$P(\mathfrak{C}(s_0 s_1 \dots s_m)) = \prod_{0 \leq i < m} P(s_i, s_{i+1}).$$

4.2 Probability of Linear-time Properties

For a linear-time property φ defined over an alphabet 2^{AP} for a set of atomic propositions AP, and a Markov chain $M = (AP, S, \iota, P, L)$, the probability of M with respect to φ is defined by the probability

$$P(M, \varphi) = P(\{\pi \in \mathit{Paths}(M) \mid L(\pi) \in \varphi\})$$

where $L(\pi)$ is the generalization of the labeling function L to paths. The probability $P(M, \varphi)$ is only computable if the set $\{\pi \in \mathit{Paths}(M) \mid L(\pi) \in \varphi\}$ is measurable.

To determine the probability of a linear-time property φ for a transition system $T = (AP, I, O, S, s_0, \tau, L)$, we can compute the probability of φ over the following underlying Markov chain $M = (AP, S, \iota, P, L)$ where

- $\iota(s) = \begin{cases} 1 & \text{if } s = s_0 \\ 0 & \text{otherwise} \end{cases}$
- $P(s, s') = \frac{1}{2^{|I|}}$ for all $s, s' \in S$

Consider, for example, the transition system T depicted in Figure 4.1 and the underlying Markov chain shown in the same figure. The transition system satisfies the property given by the formula $\varphi = \Box p$ with probability 0, since, out of the infinitely many traces in T, there is only one trace that satisfies φ, namely the infinite trace with the input r^ω. This probability can be computed over the Markov chain, by computing the probability of directly reaching s_1 from the initial state, and staying in this state, which is equal to $\frac{1}{2} \cdot \prod_{i \in \mathbb{N}} \frac{1}{2} = 0$.

We can also compute the probability of φ by computing the probability of the set of traces violating φ. Since φ is a safety property, each violation of φ in the transition system must have a bad prefix for φ. For a bound $n = 2$ on the bad prefixes, we can distinguish two bad prefixes for φ, namely $\{p, \overline{r}\}\{\overline{p}, r\}$ and $\{p, \overline{r}\}\{\overline{p}, \overline{r}\}$. This means that at least half of the traces of T violate φ. For $n = 3$, we can compute a new lower bound $\frac{3}{4}$ on the probability of violating φ, since we can distinguish six different bad prefixes out of eight possible prefixes of length 3. For a bound $n \in \mathbb{N}$, we can give the probability for n by the formula $\sum_{i=1}^{n} (\frac{1}{2})^i$. If we would compute the limit of this formula when n grows towards ∞, i.e., the value $\lim_{n \to \infty} \sum_{i=1}^{n} (\frac{1}{2})^i$, we would get the value 1, which in turn means that the probability of satisfying φ in T is equal to 0. To compute an upper bound on the probability we can compute the rate of the number of bad prefixes

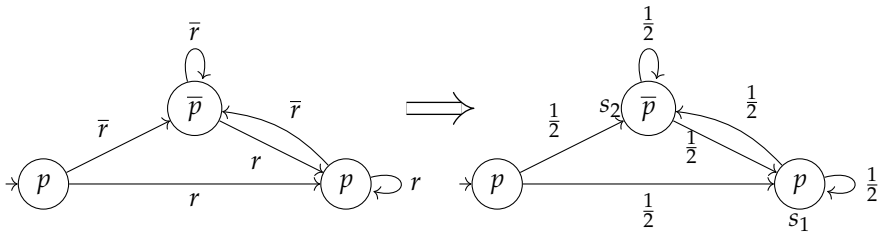


Figure 4.1: A transition system and its underlying Markov chain. The Markov chain satisfies $\Box p$ with probability 0

of a certain length. In case the property is a safety property, the limits of this rate is equal to the exact probability.

We discuss how counting the number of bad prefixes, good prefixes, and lassos for a linear-time property φ can be used to compute upper and lower bounds on the probability of a transition system satisfying φ, and when the rate of theses models in the limit is equal to the exact probability of φ. We show that for any linear-time property the sets of paths in a Markov chain defined in terms of bad prefixes or good prefixes of bounded length are always measurable, and how to use these measures for computing the probability of φ. For lassos, we show that the rate of lassos in a transition system that satisfy a property φ is not always defined.

4.3 Probabilities based on Bad Prefixes

If the set $\mathit{Models}(T, \varphi) = \{\sigma \in \mathit{Traces}(T) \mid \sigma \in \varphi\}$ for transition system T, and a linear-time property φ, is measurable, then so is the set $\mathit{Violations}(T, \varphi)$, and we can compute the probability $P(T, \varphi)$ by computing the probability $1 - P(T, \overline{\varphi})$.

In the following, we show that the probability $P(T, \overline{\varphi})$ can be bounded from below by the probability $P(\{\sigma \in \mathit{Traces}(T) \mid \exists w \in \mathit{BadPref}(\varphi).\ w < \sigma\})$. The set $\{\sigma \in \mathit{Traces}(T) \mid \exists w \in \mathit{BadPref}(\varphi).\ w < \sigma\}$ is measurable, since it can be defined as the union of cylinder sets

$$\bigcup_{m \in \mathbb{N},\ w_0 w_1 \ldots w_m \in \mathit{BadPref}(\varphi)} \mathfrak{C}(w_0 w_1 \ldots w_m)$$

and because

$$\{\sigma \in \mathit{Traces}(T) \mid \exists w \in \mathit{BadPref}(\varphi).\ w < \sigma\} \subseteq \mathit{Violations}(T, \varphi)$$

it follows that

$$P(\{\sigma \in \mathit{Traces}(T) \mid \exists w \in \mathit{BadPref}(\varphi).\ w < \sigma\}) \leq P(T, \overline{\varphi}).$$

Furthermore, since for every $n \in \mathbb{N}$ the set $\{\sigma \in \mathit{Traces}(T) \mid \exists w \in \mathit{BadPref}(\varphi, n).\ w < \sigma\}$ is also measurable, the probability of the set for any $n \in \mathbb{N}$ is also a lower bound for the probability $P(T, \overline{\varphi})$.

Lemma 4.1. *For every safety property φ, and a transition system T, it holds that*

$$P(\{\sigma \in \mathit{Traces}(T) \mid \exists w \in \mathit{BadPref}(\varphi).\ w < \sigma\}) = P(T, \overline{\varphi})$$

Proof. Since φ is a safety property, it holds that

$$\{\sigma \in \mathit{Traces}(T) \mid \exists w \in \mathit{BadPref}(\varphi).\ w < \sigma\} = \mathit{Violations}(T, \varphi).$$

□

We show how we can compute the probability $P(\{\sigma \in \mathit{Traces}(T) \mid \exists w \in \mathit{BadPref}(\varphi).\ w < \sigma\})$ using model counting over bad prefixes. Our proof is based on the following definition.

Definition 4.3 (Bad-prefix Rate)**.** For a transition system T and a linear-time property φ, the *bad-prefix rate function* $\mathcal{E}^{T,\varphi}_{Bad} : \mathbb{N} \to [0, 1]$ is defined by

$$\mathcal{E}^{T,\varphi}_{Bad}(n) = \frac{|\mathit{BadPref}(\varphi, n) \cap \mathit{Traces}_{fin}(T)|}{|\mathit{Traces}_{fin}(T, n)|}.$$

The *bad-prefix rate* is defined by $\mathcal{E}^{T,\varphi} = \lim_{n\to\infty} \mathcal{E}^{T,\varphi}_{Bad}(n)$.

For a transition system T and linear-time property φ, we define $\mathcal{E}^{T,\varphi} = \lim_{n\to\infty} \mathcal{E}^{T,\varphi}_{Bad}(n)$, and show that $\mathcal{E}^{T,\varphi}$ is equal to the probability $P(\{\sigma \in \mathit{Traces}(T) \mid \exists w \in \mathit{BadPref}(\varphi).\ w < \sigma\})$, and thus defines a lower bound for $P(T, \overline{\varphi})$. This can be done by first establishing the relation between the function $\mathcal{E}^{T,\varphi}_{Bad}(n)$ and the probability of the sets

$$B_n = \{\sigma \in \mathit{Traces}(T) \mid \exists w \in \mathit{BadPref}(\varphi, n).\ w < \sigma\}$$

for $n \in \mathbb{N}$.

Lemma 4.2. *For a transition system T, a linear-time property φ, and a bound $n \in \mathbb{N}$, it holds that*

$$\mathcal{E}^{T,\varphi}_{Bad}(n) = P(B_n).$$

Proof. Let $T = (AP, I, O, S, s_0, \delta, L)$. The set B_n can be defined as the union

$$\bigcup_{w_1 w_2 \ldots w_n \in BadPref(\varphi,n) \cap Traces_{fin}(T)} \mathfrak{C}(w_1 w_2 \ldots w_n).$$

The probability $P(B_n)$ is thus equal to the value

$$\sum_{w_1 w_2 \ldots w_n \in BadPref(\varphi,n) \cap Traces_{fin}(T)} P(\mathfrak{C}(w_1 w_2 \ldots w_n))$$

which in turn is equal to

$$\sum_{w_1 w_2 \ldots w_n \in BadPref(\varphi,n) \cap Traces_{fin}(T)} \frac{1}{(2^{|I|})^n}.$$

Because T is deterministic, this in turn is equal to

$$\frac{|BadPref(\varphi, n) \cap Traces_{fin}(T)|}{(2^{|I|})^n} = \frac{|BadPref(\varphi, n) \cap Traces_{fin}(T)|}{|Traces_{fin}(T, n)|}.$$

□

Using the last lemma, we show that we can use $\mathcal{E}^{T,\varphi}$ to compute an upper bound on $P(T, \varphi)$.

Theorem 4.1. *For a transition system T, a linear-time property φ, it holds that*

$$P(T, \varphi) \leq 1 - \mathcal{E}^{T,\varphi}$$

Proof. We show that $\mathcal{E}^{T,\varphi} = P(\{\sigma \in Traces(T) \mid \exists w \in BadPref(\varphi).\ w < \sigma\})$. This can be done by showing that (1) the sets B_n form a countable union that is equal to the set $\{\sigma \in Traces(T) \mid \exists w \in BadPref(\varphi).\ w < \sigma\}$. Then we show that (2) for each $n \in \mathbb{N}$ it holds that $B_n \subseteq B_{n+1}$. This implies that the probability of the set $P(\{\sigma \in Traces(T) \mid \exists w \in BadPref(\varphi).\ w < \sigma\})$ is equal to $\lim_{n\to\infty} P(B_n)$. Finally, we show that (3) $\lim_{n\to\infty} P(B_n)$ is equal to $\mathcal{E}^{T,\varphi}$.

(1) Let $\sigma \in \{\sigma \in Traces(T) \mid \exists w \in BadPref(\varphi).\ w < \sigma\}$. It follows that there is w with $|w| = n$ for some $n \in \mathbb{N}$ such that $w \in BadPref(\varphi, n)$. This in turn means that $\sigma \in B_n$. With a similar argumentation we can also prove the other direction, i.e., that each $\sigma \in B_n$ for some $n \in \mathbb{N}$ is also an element of the set $\{\sigma \in Traces(T) \mid \exists w \in BadPref(\varphi).\ w < \sigma\}$.

(2) Let $\sigma \in B_n$. It follows that there is w with $|w| = n$ such that $w \in \mathit{BadPref}(\varphi, n)$. Let $\alpha \cdot \sigma' \in \Sigma^\omega$, such that $w \cdot \alpha \cdot \sigma' = \sigma$. Since w is a bad prefix, so is $w \cdot \alpha$, and thus $\sigma \in B_{n+1}$.

(3) From the last lemma we know that for each $n \in \mathbb{N}$, it holds that $\mathcal{E}_{Bad}^{T,\varphi}(n) = P(B_n)$. This implies that $\lim_{n\to\infty} P(B_n) = \mathcal{E}^{T,\varphi}$.

□

In summary, for a value $n \in \mathbb{N}$, the value of the rate function $\mathcal{E}_{Bad}^{T,\varphi}(n)$ defines a lower bound for the probability $P(T, \overline{\varphi})$, and thus we can compute an upper bound on the probability $P(T, \varphi)$ by computing the value $1 - \mathcal{E}_{Bad}^{T,\varphi}(n)$. In case φ is a safety property, by computing the value $\mathcal{E}^{T,\varphi}$ we can compute the exact probability $P(T, \varphi)$ and $P(T, \overline{\varphi})$.

4.4 Probabilities based on Good Prefixes

For every linear-time property φ, we know that every good prefix for φ is a bad prefix for $\overline{\varphi}$. Computing the probability for the set

$$\{\sigma \in \mathit{Traces}(T) \mid \exists w \in \mathit{GoodPref}(\varphi).\ w < \sigma\}$$

can thus be done by computing the probability for the set

$$\{\sigma \in \mathit{Traces}(T) \mid \exists w \in \mathit{BadPref}(\overline{\varphi}).\ w < \sigma\}.$$

Since

$$\{\sigma \in \mathit{Traces}(T) \mid \exists w \in \mathit{BadPref}(\overline{\varphi}).\ w < \sigma\} \subseteq \mathit{Violations}(T, \overline{\varphi})$$

it follows that

$$\{\sigma \in \mathit{Traces}(T) \mid \exists w \in \mathit{GoodPref}(\varphi).\ w < \sigma\} \subseteq \mathit{Models}(T, \varphi)$$

and thus

$$P(\{\sigma \in \mathit{Traces}(T) \mid \exists w \in \mathit{GoodPref}(\varphi).\ w < \sigma\}) \leq P(T, \varphi)$$

and the probability of every set $\{\sigma \in \mathit{Traces}(T) \mid \exists w \in \mathit{GoodPref}(\varphi, n).\ w < \sigma\}$ is also a lower bound for the probability $P(T, \varphi)$. In the case that φ is a co-safety property, the probability of the set of good prefixes is equal to the probability $P(T, \varphi)$.

Lemma 4.3. *For every co-safety property φ, and a transition system T, it holds that*

$$P(\{\sigma \in \mathit{Traces}(T) \mid \exists w \in \mathit{GoodPref}(\varphi).\ w < \sigma\}) = P(T, \varphi)$$

Proof. Since φ is a co-safety property, it holds that

$$\{\sigma \in \mathit{Traces}(T) \mid \exists w \in \mathit{GoodPref}(\varphi).\ w < \sigma\} = \mathit{Models}(T, \varphi).$$

□

We show now that computing the probability $P(\{\sigma \in \mathit{Traces}(T) \mid \exists w \in \mathit{GoodPref}(\varphi).\ w < \sigma\})$, can be done by computing the limit of the rate of good prefixes.

Definition 4.4 (Good-prefix Rate)**.** For a transition system T and a linear-time property φ, the *good-prefix rate function* $\mathcal{R}^{T,\varphi}_{\mathit{Good}} : \mathbb{N} \to [0,1]$ is defined by

$$\mathcal{R}^{T,\varphi}_{\mathit{Good}}(n) = \frac{|\mathit{GoodPref}(\varphi, n) \cap \mathit{Traces}_{\mathit{fin}}(T)|}{|\mathit{Traces}_{\mathit{fin}}(T, n)|}$$

The *good-prefix rate* is defined by $\mathcal{R}^{T,\varphi} = \lim_{n\to\infty} \mathcal{R}^{T,\varphi}_{\mathit{Good}}(n)$.

For a transition system T and linear-time property φ, we define $\mathcal{R}^{T,\varphi} = \lim_{n\to\infty} \mathcal{R}^{T,\varphi}_{\mathit{Good}}(n)$, which, as we will show, is equal to the probability of the set of good prefixes of φ.

Theorem 4.2. *For a transition system T, a linear-time property φ, it holds that*

$$\mathcal{R}^{T,\varphi} \leq P(T, \varphi)$$

Proof. We show that $\mathcal{R}^{T,\varphi} = P(\{\sigma \in \mathit{Traces}(T) \mid \exists w \in \mathit{GoodPref}(\varphi).\ w < \sigma\})$. Since $P(\{\sigma \in \mathit{Traces}(T) \mid \exists w \in \mathit{GoodPref}(\varphi).\ w < \sigma\}$ is equal to $P(\{\sigma \in \mathit{Traces}(T) \mid \exists w \in \mathit{BadPref}(\overline{\varphi}).\ w < \sigma\}$, it holds that $P(\{\sigma \in \mathit{Traces}(T) \mid \exists w \in \mathit{GoodPref}(\varphi).\ w < \sigma\} = \mathcal{E}^{T,\overline{\varphi}}$. The claim follows from the fact that $\mathcal{E}^{T,\overline{\varphi}} = \mathcal{R}^{T,\varphi}$, which is a consequence of $\mathit{GoodPref}(\varphi, n) = \mathit{BadPref}(\overline{\varphi}, n)$, for every $n \in \mathbb{N}$. □

4.5 Probabilities based on Lassos

In the following, we show the relation between the rate of lassos of a certain length that satisfy a linear-time property φ and the probability of traces in a transition system T that satisfy the property φ. The rate of lassos is defined as follows.

Definition 4.5 (Lasso Rate)**.** For a transition system T and a linear-time property φ we define the *lasso rate function* by the function

$$\mathcal{R}^{T,\varphi}_{Lasso}(n) = \frac{|Lasso(\varphi, n) \cap Lasso(Traces(T), n)|}{|Lasso(Traces(T), n)|}$$

In contrast to the bad-prefix rate, that allows us to compute the exact probability for safety properties, we will show that the rate function $\mathcal{R}^{T,\varphi}_{Lasso}(n)$ for some transition system T and for a linear-time property φ does not necessarily converge to the value $P(T, \varphi)$ when n grows to infinity. The reason for this is that lassos cannot capture violations of the system that are not ultimately periodic. For any n, the sets $Lasso(\varphi, n) \cap Lasso(Traces(T), n)$ and $Lasso(Traces(T), n)$, only define subsets of the sets $Models(T, \varphi)$ and $Traces(T)$, respectively, and thus the rates will not always coincide with the probabilities.

Theorem 4.3. *There is a transition system T and a linear-time property φ, such that, $\mathcal{R}^{T,\varphi}_{Lasso}(n)$ is nonconvergent.*

Proof. We first define the property φ and then give a transition system T with a non-convergent lasso rate function for φ.

- *The property φ*: We define a linear-time property φ over the atomic propositions $AP = \{r, p\}$. Let $c_0, c_1, \cdots \in \mathbb{N}$ and $d_0, d_1, \cdots \in \mathbb{N}$ such that, for all $i \in \mathbb{N}$, we have $c_i < d_i$ and $d_i \leq c_{i+1}$. The property φ is defined as the union $\varphi = \bigcup_{j \in \mathbb{N}} \Gamma_j$ where:
  - $\Gamma_0 = \emptyset$
  - If $j > 0$ we distinguish two cases
    - $\exists i \in \mathbb{N}.\, j \in [c_i, d_i)$ then
      $\Gamma_j = \{\sigma \in (2^{AP})^\omega \mid \exists (u, v) \in Lasso((2^{\{p\}})^\omega, j).\, \sigma_{|\{p\}} = u \cdot v^\omega\} \setminus \bigcup_{h=0}^{j-1} \Gamma_h$
    - $\exists i \in \mathbb{N}.\, j \in [d_i, c_{i+1})$ then
      $\Gamma_j = \Gamma_{j-1}$

  The property φ defines the set of all traces over AP that are ultimately periodic for the set of propositions $\{p\}$ and for which the smallest lasso they can be represented by is of length $j \in [c_i, d_i)$ for some $i \in \mathbb{N}$. Figure 4.2 gives an illustration on which traces over AP are models of φ.

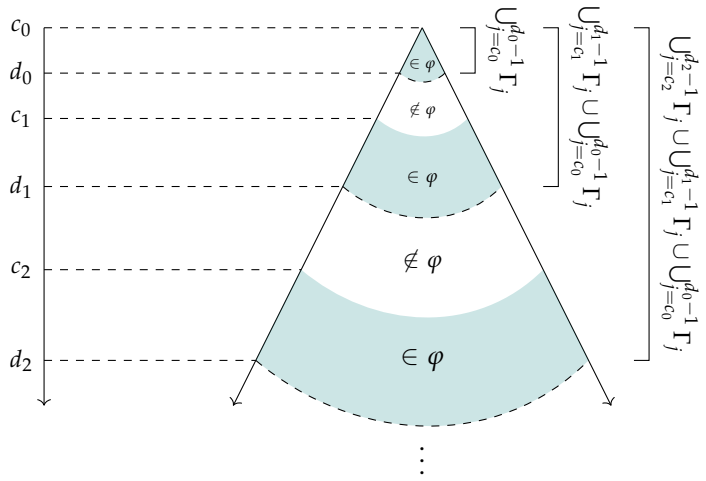
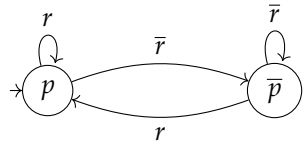


Figure 4.2: A nonconvergent linear-time property φ. The colored areas represent ultimately-periodic words that are in φ. The white areas represent ultimately-periodic words that are not in φ.

- *The system T*: We define T as the following transition system



  Every ultimately periodic trace in T over $\{p\}$ is also ultimately periodic over AP.

- *The lasso rate*: We show that there exist intervals $[c_i, d_i)$ and $[d_i, c_{i+1})$ for all $i \in \mathbb{N}$ such that the function $\mathcal{R}^{T,\varphi}_{Lasso}(n)$ is not convergent. The definition of $\mathcal{R}^{T,\varphi}_{Lasso}(n)$ distinguishes two cases:

  – $\exists i \in \mathbb{N}.\ n \in [d_i, c_{i+1}]$: For the bound n the set of models in T that can be represented by lassos of length n is defined as follows:

$$Lasso(Traces(T) \cap \varphi, n) = Lasso(Traces(T) \cap \varphi, n-1)$$

  because every model that can be represented by a lasso of length $n-1$ can be represented by one of length n, and all traces that are representable by lassos of length n, but not by smaller ones, are violations

of φ. This implies that

$$\mathcal{R}^{T,\varphi}_{Lasso}(n) = \frac{Lasso(Traces(T) \cap \varphi, n-1)}{Lasso(Traces(T), n)}$$

and in turn that

$$\mathcal{R}^{T,\varphi}_{Lasso}(n) = \mathcal{R}^{T,\varphi}_{Lasso}(c_i - 1) \cdot \left(\frac{Lasso((2^{|I|})^\omega, n-1)}{Lasso((2^{|I|})^\omega, n)} \right)^{n-c_i+1}.$$

The rate $\left(\frac{Lasso((2^{|I|})^\omega, n-1)}{Lasso((2^{|I|})^\omega, n)} \right)^{n-c_i+1} < 1$ and monotonically decreasing towards 0 when n grows towards ∞. This means that the function $\mathcal{R}^{T,\varphi}_{Lasso}$ is decreasing as long as $d_i \leq n \leq c_{i+1}$, and that there is a c_{i+1} such that, $\mathcal{R}^{T,\varphi}_{Lasso}(c_{i+1}) < \mathcal{R}^{T,\varphi}_{Lasso}(c_i)$, because it is decreasing towards 0.

– $\exists i \in \mathbb{N}.n \in [c_i, d_i)$: For the bound n, the set of lasso violations in T is defined as follows:

$$\begin{aligned} Lasso(Traces(T) \cap \varphi, n) = \; & Lasso(Traces(T) \cap \varphi, n-1) \\ & \cup (Lasso((2^{|I|})^\omega, n) - Lasso((2^{|I|})^\omega, n-1)) \end{aligned}$$

because every model that can be represented by a lasso of length $n-1$ can be represented by one of length n, and all traces that are representable by lassos of length n, but not by smaller ones, are models of φ. This implies that

$$\begin{aligned} & \mathcal{R}^{T,\varphi}_{Lasso}(n) \\ &= \frac{|Lasso(Traces(T) \cap \varphi, n-1) \cup (Lasso((2^{|I|})^\omega, n) - Lasso((2^{|I|})^\omega, n-1))|}{|Lasso((2^{|I|})^\omega, n)|} \\ &= \frac{|Lasso(Traces(T) \cap \varphi, n-1)| + |Lasso((2^{|I|})^\omega, n)| - |Lasso((2^{|I|})^\omega, n-1)|}{|Lasso((2^{|I|})^\omega, n)|} \\ &= \frac{|Lasso(Traces(T) \cap \varphi, n-1)| + |Lasso((2^{|I|})^\omega, n)| - |Lasso((2^{|I|})^\omega, n-1)|}{|Lasso((2^{|I|})^\omega, n-1)| + |Lasso((2^{|I|})^\omega, n)| - |Lasso((2^{|I|})^\omega, n-1)|} \end{aligned}$$

which in turn means that

$$\mathcal{R}^{T,\varphi}_{Lasso}(n) > \mathcal{R}^{T,\varphi}_{Lasso}(n-1)$$

because $|Lasso((2^{|I|})^\omega, n)| - |Lasso((2^{|I|})^\omega, n-1)| > 0$, and

$$\frac{|Lasso(Traces(T) \cap \varphi, n-1)| + \delta}{|Lasso((2^{|I|})^\omega, n-1)| + \delta} > \frac{|Lasso(Traces(T) \cap \varphi, n-1)|}{|Lasso((2^{|I|})^\omega, n-1)|}$$

for any $\delta \in \mathbb{N}$. This further means that the function $\mathcal{R}^{T,\varphi}_{Lasso}$ is increasing towards 1 when n grows towards ∞. This means that $\mathcal{R}^{T,\varphi}_{Lasso}(n)$ is increasing as long as $c_i \leq n < d_i$ and that there is a d_i such that $\mathcal{R}^{T,\varphi}_{Lasso}(d_i - 1) > \mathcal{R}^{T,\varphi}_{Lasso}(d_{i-1} - 1)$, because it is increasing towards 1.

The function $\mathcal{R}^{T,\varphi}_{Lasso}(n)$ is increasing towards 1 when $n \in [c_i, d_i)$ and decreasing towards 0 when $n \in [d_i, c_{i+1})$, which means it will never converge. □

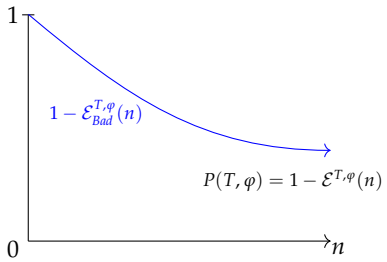
We summarize our result from the last three sections and show how the number of each type of bounded model can be used to determine lower and upper bounds on the probability of satisfying a linear-time property. Figure 4.3 shows the shapes of the rate functions for the different types of models.

The rate of bad prefixes gives an upper bound on the satisfiability probability, and is equal to the actual probability, if φ is a safety property Figure 4.3c. The rate of good prefixes gives a lower bound on the satisfiability probability, and is equal to the actual probability, if φ is a co-safety property Figure 4.3b. When a linear-time property is both safety and co-safety, then the rate of bad prefixes and good prefixes are equal. If we are not able to compute the limits of the rates of bad prefixes and good prefixes, the the value of the rate functions define a lower bound, and an upper bound, respectively, for any bound on the bad and good prefixes.

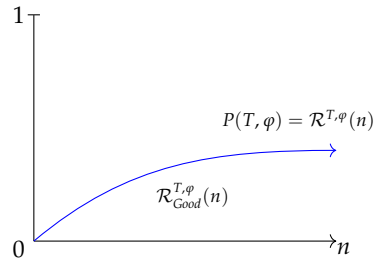
The limit of the lasso rate is not always defined for linear-time properties in general.

4.6 Bibliographic Remarks

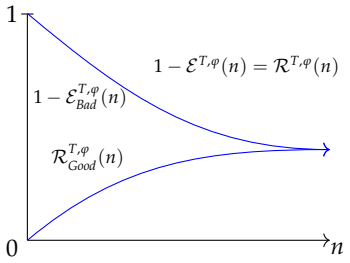
The rate functions defined in this chapter are closely related to the notion of density of languages. The density of a language defines a probabilistic measure on the language by relating the number of words in the language to the total number of words definable over the alphabet of the language [27, 69]. The study of density has a long history [27, 38, 70, 83, 140]. For each language $\varphi \subseteq \Sigma^*$ of finite words over some alphabet Σ, the *density function* is defined as the quotient $d_\varphi(n) = |\varphi \cap \Sigma^n| / |\Sigma^n|$, i.e., the number of words of length n in φ over the number of all words of length n. The rate functions for bad prefixes and good prefixes, define the densities of the languages of bad prefixes and good prefixes of bounded length. In 1958, Chomsky and Miller [38] showed that for each regular language φ, there exists an initial length n_0 such that for all $n \geq n_0$, $|\varphi \cap \Sigma^n|$ can be described by a linear recurrence. For example, for the language ψ of the



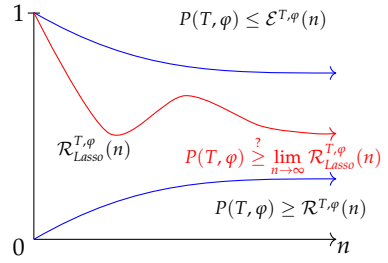
(a) φ is a safety property.



(b) φ is a co-safety property.



(c) φ is a safety and a co-safety property.



(d) φ is a combination of different properties

Figure 4.3: Bounding the probability of a linear-time property φ in a transition system T using model counting over bad prefixes, good prefixes, and lassos

regular expression $(ab + baa)^*$, we have that $|\psi \cap \Sigma^n| = |\psi \cap \Sigma^{n-2}| + |\varphi \cap \Sigma^{n-3}|$. The recursive description of $|\varphi \cap \Sigma^n|$ allows for a detailed analysis of the shapes of $|\varphi \cap \Sigma^n|$ and $d_\varphi(n)$ (cf. [83]). The result was later extended to the nonambiguous context-free languages [83]. Much attention has focussed on *sparse* languages, i.e., languages where $|\varphi \cap \Sigma^n|$ can be bounded from above by a polynomial [54, 70, 140]. Sparse languages can be used to restrict NP-complete problems so that they can be solved polynomially [54]. An interesting application of the density is to determine how well a non-regular language is approximated by a finite automaton [59]; this is important in streaming algorithms, where the incoming string must be classified quickly, and it suffices if the classification is correct most of the time.

Notions related to the density have also been used to quantitatively define the safety level of a property by computing the probability that a prefix of a

word not in φ is a bad prefix of φ [62]. The notion used in this work uses a generalization of density where the sizes of two different languages are related to each other.

Anther notion related to the density is language entropy[38]. Language entropy was used for the investigation of the asymptotic behavior of temporal logic [7], to show the relation between formulas in parametric linear-time temporal logic and formulas in standard LTL as some bounds tend to infinity.

Part II

Applications of Model Counting in Formal Verification

Chapter 5

Model Checking of Counting Hyperproperties

Specifications of security-critical systems require us in many cases to reason about multiple executions of the system. Such specifications are therefore not expressible as linear-time properties. Hyperproperties, on the other hand, define sets of sets of traces, and allow us to relate traces of a system with each other [46]. In this chapter, we study a special type of hyperproperties called counting hyperproperties, which allow for the specification of quantitative security policies. We study counting hyperproperties in the setting of HyperLTL [44], a temporal logic for hyperproperties, and show that, while counting hyperproperties can be verified using the traditional model checking algorithm for HyperLTL, we can improve the complexity of model checking counting hyperproperties using a new algorithm based on a maximum model counting algorithm for LTL.

5.1 Information-Flow Policies

The main goal in information-flow control is to regulate the flow of information in a system, preventing any information with a higher security level from leaking to users with a lower security level. One of the first formalizations of information-flow control was introduced with the notion of *noninterference* [75]. Noninterference states that a change in the secret values of a system should not

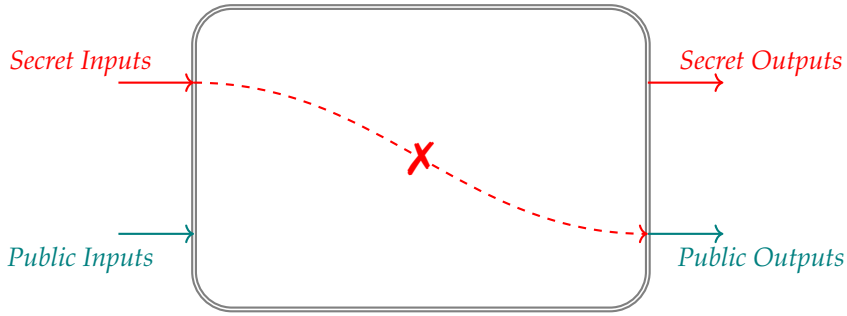


Figure 5.1: A security-critical systems with distinguished public and secret inputs and outputs. Information about the secret inputs should not flow to the public outputs.

be visible by an adversary that is capable of observing the inputs and outputs to and of that system. For reactive systems, noninterference can be formalized based on the model depicted in Figure 5.1. In a security-critical reactive system, we can distinguish between two types of inputs and output, *secret* and *public*. *Secret* inputs and outputs represent confidential information that is not directly visible to any external party observing the system. *Public* inputs and outputs, on the other hand, are visible to any user interacting with the system [1]. For each new secret and public input to the system, the system computes a new secret and public output. Noninterference for reactive systems can then be formally defined as follows. For a transition system T defined over public inputs PI, and public outputs PO, noninterference is defined by the formula[2]

$$\forall \pi_1, \pi_2 \in \mathit{Traces}(T).(\pi_1 =_{PI} \pi_2) \rightarrow (\pi_1 =_{PO} \pi_2)$$

where $=_X$, for some set of atomic propositions X, denotes the relation that only includes pairs of infinite words that agree on the valuation of propositions in X. Noninterference states that if an adversary observes two traces of the system that share the same valuation of the public inputs, then the adversary should see no difference in the valuations of the public outputs of these two traces.

Security policies like noninterference cannot be defined by linear-time properties since we need to reason about pairs of traces of the system rather than

[1] This is a very abstract view on security-critical systems where we distinguish only between two levels of secrecy. In practice, we may distinguish between several secrecy levels ordered within a lattice, the so-called security lattice [55]. Noninterference can then be formulated as the policy that requires change in the input of a certain level not to be visible in outputs with a lower secrecy level.

[2] This is one of many definitions of noninterference in the setting of reactive systems. Other possible definition may require that the public outputs are equal only as long as the public inputs are. We chose one of these definitions for the purpose of demonstration.

individual traces. In fact, noninterference defines a set of sets of traces, those that include only traces where two traces that are equal on the public input are also equal on the public output. Properties of this type are called *hyperproperties* [46], and can be formalized as follows.

Definition 5.1 (Hyperproperties [46])**.** For an alphabet Σ, let Σ^ω denote the set of infinite traces over Σ. A hyperproperty over Σ is a set $H \subseteq 2^{\Sigma^\omega}$.
A system T satisfies a hyperproperty H, denoted by $T \models H$ if and only if $\mathit{Traces}(T) \in H$.

Intuitively, the sets defined by hyperproperties define sets of traces that are allowed to co-exist in a system. Hyperproperties have been widely used for the specification of information-flow policies, but are not only restricted to this domain. Prominent examples of hyperproperties include robustness properties for reactive systems, where we want to make sure that the outputs of systems remain stable under the influence of noise [123]. Hyperproperties can also be used to define symmetries between process that access a shared resource, or to define the functional correctness of encoders and decoders of error resistant codes, where we require all pairs of code words to have a certain minimum Hamming distance [66].

In this chapter, we will focus on hyperproperties that describe information-flow policies and quantitative extensions of such policies. Our model checking approach is nevertheless adaptable to all other mentioned domains, and beyond.

5.2 HYPERLTL: A Temporal Logic for Hyperproperties

We investigate hyperproperties that can be expressed in the temporal hyperlogic HYPERLTL [44]. HYPERLTL extends linear-time temporal logic (LTL) with quantifiers over trace variables. Let $\mathcal{V}$ be an infinite supply of *trace variables*, and let AP be a set of atomic propositions. The syntax of HYPERLTL is given by the following grammar:

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

where $a \in AP$ is an atomic proposition, and $\pi \in \mathcal{V}$ is a trace variable. Note that atomic propositions are indexed by trace variables. The derived operators $\Diamond$, $\Box$ and $\mathcal{W}$ are defined as for LTL. We call a trace variable π *free* in a HyperLTL

formula if there is no quantification over π, and we call a HyperLTL formula φ *closed* if there exists no free trace variable in φ.

Formally, the semantics of HYPERLTL formulas over an alphabet Σ is given with respect to a *trace assignment* Π from $\mathcal{V}$ to Σ^ω, i.e., a partial function mapping trace variables to actual traces. $\Pi[\pi \mapsto t]$ denotes that π is mapped to t, with everything else mapped according to Π. $\Pi[i, \infty]$ denotes the trace assignment such that for all π, $\Pi[i,\infty](\pi) = \Pi(\pi)[i,\infty]$. Given a set of traces T, the semantics of HYPERLTL is defined as follows

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

We say a set of traces T *satisfies* a HYPERLTL formula φ if $\emptyset \models_T \varphi$, where $\emptyset$ is the empty trace assignment.

Example 5.1 (Noninterference)**.** Let a set of atomic propositions $AP = SI \cup SO \cup PI \cup PO$ be composed of the sets if secret inputs, secret outputs, public inputs, and public outputs. The policy of noninterference for a system T defined over AP can be given by the HYPERLTL formula

$$\forall \pi_1.\forall \pi_2.\square(\bigwedge_{i \in PI} i_{\pi_1} = i_{\pi_2}) \rightarrow \square(\bigwedge_{o \in PO} o_{\pi_1} = o_{\pi_2}).$$

The formula states that if two traces π_1 and π_2 in $\mathit{Traces}(T)$ share the same valuation of propositions in PI at every position, then they should share the same valuation of propositions in PO at every position.

Example 5.2 (Deniability)**.** Another prominent information-flow policy is *deniability* [23, 32]. Deniability defines the policy that states that by observing the public output alone, we should not be able to infer any of the values of the secret inputs. Deniability is an important policy with respect to electronic voting, where we should not infer any information about the choices of voters, after knowing the outcome of the elections. Formally, we can define deniability by the formula

$$\forall \pi_1.\exists \pi_2.(\pi_1 =_{PO} \pi_2) \wedge (\pi_1 \neq_{SI} \pi_2).$$

The specification states that every public output sequence of the system should be the result of at least two different sequences of secret inputs. In HYPERLTL, deniability can be defined as follows. Let the set of atomic propositions $AP = SI \uplus SO \uplus PI \uplus PO$. Deniability for a system T defined over AP can be given by the HYPERLTL formula

$$\forall \pi_1 . \exists \pi_2 . \square (\bigwedge_{o \in PO} o_{\pi_1} = o_{\pi_2}) \wedge \diamondsuit (\bigvee_{i \in SI} i_{\pi_1} \neq i_{\pi_2}).$$

For each trace π_1 in $Traces(T)$, we must find another trace π_2 that shares the same valuation of public output propositions, and differs from π_1 in the valuation of secret input propositions in at least one position along the two traces.

5.2.1 A model checking algorithm for HYPERLTL

Model checking of HYPERLTL formulas builds on the algorithm for model checking properties given in LTL. We start by presenting the model checking algorithm for existential HYPERLTL formulas, i.e., formulas with only existential quantifiers. An algorithm for the full fragment of HYPERLTL is then given based on the constructions used for the existential fragment.

The model checking problem for existential HYPERLTL can be reduced to LTL model checking. To model check a system T against an LTL formula φ (See Section 2.5), we construct a Büchi automaton accepting the language defined by $\overline{\varphi}$, comput the product of this automaton and the transition system T, and check the product automaton for emptiness. The language of the product automaton defines the mutual traces in the system and the Büchi automaton for $\overline{\varphi}$, and thus a trace accepted by the product automaton represents a trace in T violating the property φ.

Model checking a transitions system T against an LTL formula φ can be seen as model checking T against the HYPERLTL formula $\exists \pi . \overline{\varphi}$, where we look for a trace π in T that violates φ. To check T against an existential HYPERLTL formula $\exists \pi_1 \ldots . \exists \pi_k . \psi$, we need to find a set of k traces in T that satisfy ψ, and thus compute the product of the Büchi automaton for ψ with k copies of the transition system T, one copy for finding each of the k traces. The ith copy of the system is associated with atomic propositions in ψ indexed with trace variable π_i. Such an automaton construction was presented in [66]. We introduce a slightly modified automata construction that is compatible with finite labeled transition systems. Instead of labeling over states of a Kripke structure as in [66], we label over sets of atomic propositions. The constructions are equivalent as any nondeterminism

in the Kripke structure is inherently resolved, because formulas in HYPERLTL quantify over traces instead of paths.

Automata for HYPERLTL. Let φ be a closed existential HYPERLTL formula defined over a set of atomic propositions AP. Let $\mathcal{V}$ be the set of all trace variables introduced by the existential quantifiers in φ. Let $AP_{[\mathcal{V}]}$ be defined as the set

$$AP_{[\mathcal{V}]} = \{a_\pi \mid a \in AP, \pi \in \mathcal{V}\}.$$

A Büchi automaton B_φ for φ is defined over the alphabet $\Sigma = (2^{AP_{[\mathcal{V}]}})$ and is constructed inductively as as follows.

For atomic propositions, Boolean connectives, and temporal operators in the quantifier free part of φ, our construction follows the standard translation from LTL to alternating Büchi automata [150]. Let $B_{\psi_1} = (\Sigma, Q_1, q_{0,1}, \delta_1, F_1)$, and $B_{\psi_2} = (\Sigma, Q_2, q_{0,2}, \delta_2, F_2)$ be two alternating Büchi automata for two quantifier-free HYPERLTL formulas ψ_1 and ψ_2. An alternating Büchi automaton for a quantifier free HYPERLTL formula ψ is constructed as follows:

- $\psi = a_\pi$ for some $a_\pi \in AP_{[\mathcal{V}]}$:

  $$B_\psi = (\Sigma, \{q_0\}, q_0, \delta, \emptyset) \text{ where } \forall \alpha \in \Sigma.\ \delta(q_0, \alpha) = (a_\pi \in \alpha).$$

- $\psi = \neg a_\pi$ for some $a_\pi \in AP_{[\mathcal{V}]}$:

  $$B_\psi = (\Sigma, \{q_0\}, q_0, \delta, \emptyset) \text{ where } \forall \alpha \in \Sigma.\ \delta(q_0, \alpha) = (a_\pi \notin \alpha).$$

- $\psi = \psi_1 \wedge \psi_2$:

  $$B_\psi = (\Sigma, Q_1 \cup Q_2 \cup \{q_0\}, q_0, \delta, F_1 \cup F_2)$$

  where q_0 is a fresh state,
  $\forall \alpha \in \Sigma.\ \delta(q_0, \alpha) = \delta_1(q_{0,1}, \alpha) \wedge \delta_2(q_{0,2}, \alpha)$, and
  $\forall i \in \{1, 2\}. \forall q \in Q_i. \forall \alpha \in \Sigma.\ \delta(q, \alpha) = \delta_i(q, \alpha)$

- $\psi = \psi_1 \vee \psi_2$:

  $$B_\psi = (\Sigma, Q_1 \cup Q_2 \cup \{q_0\}, q_0, \delta, F_1 \cup F_2)$$

  where q_0 is a fresh state,
  $\forall \alpha \in \Sigma.\ \delta(q_0, \alpha) = \delta_1(q_{0,1}, \alpha) \vee \delta_2(q_{0,2}, \alpha)$, and
  $\forall i \in \{1, 2\}. \forall q \in Q_i. \forall \alpha \in \Sigma.\ \delta(q, \alpha) = \delta_i(q, \alpha)$

- $\psi = \bigcirc \psi_1$:

  $$B_\psi = (\Sigma, Q_1 \cup \{q_0\}, q_0, \delta, F_1)$$

where q_0 is a fresh state,
$\forall \alpha \in \Sigma.\ \delta(q_0, \alpha) = q_{0,1}$, and
$\forall q \in Q_1. \forall \alpha \in \Sigma.\ \delta(q, \alpha) = \delta_1(q, \alpha)$.

- $\psi = \psi_1 \,\mathcal{U}\, \psi_2$:

$$B_\psi = (\Sigma, Q_1 \uplus Q_2 \uplus \{q_0\}, q_0, \delta, F_1 \uplus F_2)$$

where q_0 is a fresh state,
$\delta(q_0, \alpha) = \delta_2(q_{0,2}, \alpha) \vee (\delta_1(q_{0,1}, \alpha) \wedge q_0)$, and
$\forall i \in \{1, 2\}. \forall q \in Q_i. \forall \alpha \in \Sigma.\ \delta(q, \alpha) = \delta_i(q, \alpha)$.

- $\psi = \psi_1 \,\mathcal{R}\, \psi_2$:

$$B_\psi = (\Sigma, Q_1 \uplus Q_2 \uplus \{q_0\}, q_0, \delta, F_1 \uplus F_2 \uplus \{q_0\})$$

where q_0 is a fresh state,
$\delta(q_0, \alpha) = \delta_2(q_{0,2}, \alpha) \wedge (\delta_1(q_{0,1}, \alpha) \vee q_0)$, and
$\forall i \in \{1, 2\}. \forall q \in Q_i. \forall \alpha \in \Sigma.\ \delta(q, \alpha) = \delta_i(q, \alpha)$.

Intuitively, the construction above defines an automaton that represents the language of sets of k traces $\{\pi_1, \ldots, \pi_k\}$ that satisfy the formula ψ. Each trace π_i, is distinguished by its own set of atomic propositions $\{a_{\pi_i} \mid a \in AP\}$.

To handle the existential quantifiers of φ we first construct the alternating automaton as presented above for the quantifier-free part of φ, transform the automaton into a nondeterministic Büchi automaton that defines the same language (Lemma 2.1), and then compute the product of this automaton with a k self-composition of the transition system under scrutiny, each copy in the composition for tracking one of the k traces. The automaton for a formula $\varphi = \exists \pi. \psi$ is constructed inductively as follows.

Let $B_\psi = (\Sigma, Q, q_0, \delta, F)$ be a nondeterministic Büchi automaton constructed for an existential HYPERLTL formula ψ, and let $T = (AP, I, O, S, s_0, \tau, L)$ be a transition system. Let further $X_{[x \mapsto y]}$ denote a set where every $x \in X$ is substituted with a y, and $X_\pi = \{x_\pi \mid x \in X\}$. The product automaton for φ is built using the following rule:

- $\varphi = \exists \pi. \psi$

$$B_\varphi = (2^{AP_{[\mathcal{V}]} \setminus \{a_\pi \mid a \in AP\}}, (Q \times S) \uplus \{q_0'\}, \delta', F \times S)$$

where q_0' is a fresh state, and

$$\forall \alpha \in \Sigma.\ \delta'(q_0', \alpha) = \{(q', s') \mid \exists \alpha' \in 2^{\{a_\pi \mid a \in AP\}}.\ q' \in \delta(q_0, \alpha \cup \alpha') \wedge L(s_0)_\pi = \alpha'_{O_\pi} \wedge s' \in \tau(s_0, \{\alpha'_{I_\pi}\}_{[I_{[\pi]} \mapsto I]})\}$$

and

$$\forall \alpha \in \Sigma. \forall q \in Q. \forall s \in S.$$
$$\delta'((q,s),\alpha) = \{(q',s') \mid \exists \alpha' \in 2^{\{a_\pi \mid a \in AP\}}. \quad q' \in \delta(q, \alpha \cup \alpha') \wedge L(s)_\pi = \alpha'_{O_\pi} \wedge s' \in \tau(s, \{\alpha'_{I_\pi}\}_{[I_{[\pi]} \mapsto I]})\}$$

After constructing the product automaton we preform an emptiness check. If the automaton is not empty, then there is a set of k traces in the system that satisfy the property φ.

***Remark* 5.1.** *For more details on automata and representations for hyperproperties we refer the reader to [64].*

Model checking universally quantified formulas. To model check a transition system T against a universal HYPERLTL formula $\varphi = \forall \pi_1 \dots \forall \pi_k. \psi$ we search for k traces in $Traces(T)$ that violate the formula ψ. If no such traces are found, then T satisfies φ. Thus, the model checking problem for φ can be reduced to model checking problem for existential formulas.

Theorem 5.1 ([66])**.** *The problem of model checking a transition system T against an alternation-free* HYPERLTL *formula φ is* PSPACE*-complete in the length of φ and* NL*-complete in the size of T.*

Proof. The size of the alternating automaton B_φ for a HYPERLTL formula $\varphi = \exists \pi_1 \dots \exists \pi_k. \psi$ is linear in the length of φ. The size of the product automaton is of size polynomial in the size of transition system T. To check that the product automaton is empty, we need to find an accepting lasso run in the automaton. Guessing such lasso can be done in space polynomial in the length of φ and space logarithmic in the size of T [152]. The lower bound follows from the lower bound for LTL model checking, since HYPERLTL subsumes LTL [138]. □

Handling quantifier alternation. The construction of automata for existential HYPERLTL formulas can be extended to full HYPERLTL by handling negated existential formulas, i.e., formulas of the form $\neg \exists \pi. \psi$ for some HYPERLTL formula ψ. An automaton for such a formula can be constructed by computing the complement automaton of the automaton for the formula $\exists \pi. \psi$ [78]. Let *NB* be the nondeterministic Büchi automaton constructed for the formula $\exists \pi. \psi$ using

our construction above. We extend the rules for automata construction by the rule

- $\varphi = \neg\exists\pi.\psi$:

$$B_\varphi = \overline{NB}$$

where $\overline{NB}$ is the complement automaton of NB.

Constructing the complement automaton adds exponential cost to our construction. The size of the automaton constructed for model checking a system against a HYPERLTL formula thus depends on the number of quantifier alternations in the formula.

Definition 5.2 (Alternation Depth)**.** Given a HYPERLTL formula φ, the alternation depth of φ is number of alternations from existential to universal quantifiers and from universal to existential quantifiers in the quantifier prefix.

For example, for a quantifier free formula ψ, the formulas $\exists\pi\exists\pi'.\psi$ and $\forall\pi\forall\pi'.\psi$ have alternation depth 0, the formula $\exists\pi.\forall\pi'.\psi$ has alternation depth 1, and the formula $\forall\pi.\exists\pi'.\forall\pi''\exists\pi'''.\psi$ has depth 3.

Let $exp : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ be the function defined as follows:

$$depth(k, c) = \begin{cases} c & k = 0 \\ 2^{depth(k-1,c)} & k \geq 1 \end{cases}$$

In the next theorem, we use the function *depth* to give the overall complexity of the model checking problem of HYPERLTL.

Theorem 5.2 ([66])**.** *Model checking a transition system T against a* HYPERLTL *formula φ with alternation depth n can be done in space $O(depth(n, |\varphi|))$ and in space $O(depth((n-1), |T|))$.*

5.3 Counting Hyperproperties

Information leakage is unavoidable in practice, and sometimes necessary to keep a certain level of functionality in the system. For example, a password checker leaks information about a secret password by replying whether an entered phrase is right or wrong. Sometimes, password checkers inform users that the phrase must contain some special character or number, giving even more information about the secret password. Leaking such information is nevertheless still acceptable as it does not endanger the secrecy of the private password.

In quantitative information flow [45, 79, 98] we determine the security of a system by measuring the amount of information leaked by the system. As long as the amount is below a certain threshold appropriate for the system under scrutiny, the system is considered to be secure against adversaries observing the public values of the system. Quantitative approaches to information-flow control are mostly based on notions of entropy [139]. Two prominent entropy notions that have been argued to be most adequate for measuring information leakage are *min-entropy* [139] and *minimal guessing entropy* [96].

Min-entropy measures the expected likelihood of guessing the secret inputs with one try after observing the public values of the system, and is defined by the logarithm of the number of different public outputs. The *bounding problem* [154] for min-entropy is to determine whether that amount is bounded from above by a constant 2^n, corresponding to n bits of information. Specifying that the min-entropy is bounded by n bits can be done in terms of the following hyperproperty H that bounds the number of public outputs of the system [129]

$$\Gamma \in H \text{ iff } |\Gamma_{PO}| \leq 2^n$$

where Γ_{PO} defines the set of traces in Γ projected to only to the values of the public outputs. If we also want to include the public inputs, we can define the boundedness of min-entropy by the hyperproperty H' that bounds the number of public outputs for each public input of the system [129]

$$\Gamma \in H \text{ iff } \forall \Gamma' \subseteq \Gamma.\ (\forall \pi, \pi' \in \Gamma'.\ \pi =_{PI} \pi') \rightarrow |\Gamma_{PO}| \leq 2^n$$

The hyperproperty H defines a quantitative version of noninterference, where a public input may result in up to 2^n different public outputs.

Minimal guessing entropy is closely related to deniability. It measures the worst-case likelihood needed to guess the secret inputs after observing the public values of the system, and is defined as the logarithm of the number of different secret inputs leading to the same public output. The bounding problem for minimal guessing entropy can be specified by the hyperproperty H

$$\Gamma \in H \text{ iff } \forall \Gamma' \subseteq \Gamma.\ (\forall \pi, \pi' \in \Gamma'.\ \pi =_{PO} \pi') \rightarrow |\Gamma_{SI}| \geq 2^n$$

which states that each public input should be mapped to at least 2^n secret inputs.

Excursion 5.1 (Shannon Entropy)**.** Shannon entropy, also known as information entropy, is the most prominent notion of entropy [137]. Introduced

by Claude Shannon in 1948, Shannon entropy was used to measure the uncertainty about information carried over a communication channel. Shannon entropy has also been widely used for measuring information leakage [139]. In our study, we concentrate on deterministic security-critical systems with a uniform distribution of inputs. For this case, Shannon entropy and min-entropy coincide [139].

In the following, we give a formalization of quantitative hyperproperties that expresses a bound on the number of traces that may appear in a certain relation. We call such hyperproperties by the name *counting hyperproperties*.

Definition 5.3 (Counting Hyperproperties)**.** For hyperproperties H and J defined over a set of atomic propositions AP, a set $A \subseteq AP$, bounds $n, k \in \mathbb{N}$, and a comparison operator $\triangleleft \in \{<, \leq, =, >, \geq\}$, a counting hyperproperty φ is defined by the expression

$$\varphi = \forall \pi_1 \ldots \forall \pi_k.\ J \rightarrow (\#_A\ \sigma.\ H) \triangleleft n.$$

A transition system T satisfies φ, if for each set of traces $\{t_1, \ldots, t_k\} \subseteq \mathit{Traces}(T)$, with $\{t_1, \ldots, t_k\} \models J$ the following holds:

$$|\{t_A \mid t \in \mathit{Traces}(T) \wedge \{t_1, \ldots, t_k, t\} \models H\}| \triangleleft n$$

where t_A represents the projection of trace t to the valuations of propositions in A.

Example 5.3 (Quantitative Noninterference)**.** A quantitative noninterference policy lays a bound on the number of different outputs produced by the system. For a set of atomic propositions AP, and a set of public outputs $O \subseteq AP$, and for a bound 2^c corresponding to a bound of c bits of information, a system must satisfy the policy defined by the counting hyperproperty

$$(\#_{PO}\ \sigma.\ \mathit{true}) \leq n$$

We may also limit the number of outputs produced by the system for every public input to the system by using the following policy

$$\forall \pi.\ (\#_{PO}\ \sigma.\ \square(\pi =_{PI} \sigma)) \leq n$$

for a set of public inputs PI. The counting hyperproperty ensures that for each trace π in the system, there are no more than 2^c traces σ that have the same observable input as π (With respect to the definition of counting hyperproperties, $J \equiv \mathit{true}$ and $H \equiv \square(\pi =_{PI} \sigma)$) but different observable outputs.

Example 5.4 (Quantitative Deniability)**.** A quantitative variant of deniability requires that the number of secret inputs corresponding to a public output is larger than a given threshold. Quantitative deniability can be defined by the following counting hyperproperty

$$\forall \pi.\, (\#_{SI}\, \sigma.\, \square(\pi =_{PO} \sigma)) > n$$

where SI are the secret inputs to the system. For all traces π of the system, we count the number of different sequences σ over 2^{SI} in the system with the same public output sequence as in π, i.e., for the fixed output sequence given by π we count the number of input sequences that lead to this output.

In the following, we study counting hyperproperties $\varphi = \forall \pi_1 \dots \forall \pi_k.\, J \rightarrow (\#_A\, \sigma.\, H) \triangleleft n$. where the hyperproperties J and H are defined by HyperLTL formulas.

5.4 Model Checking Counting Hyperproperties

In this section, we present two approaches to model checking counting hyperproperties. One based on encodings into HyperLTL and another one based on maximum model counting. We show that the maximum model counting based approach improves the complexity exponentially in the bound, and sometimes doubly exponentially depending on the counting hyperproperty, over the model checking algorithm of HyperLTL.

5.4.1 Encoding counting hyperproperties in HyperLTL

The idea of the encoding is to check a lower bound of n traces by existentially quantifying over n traces, and to check an upper bound of n traces by *universally* quantifying over $n+1$ traces. The resulting HyperLTL formula can be verified using the standard model checking algorithm for HyperLTL presented in Section 5.2.1.

Theorem 5.3. *Every counting hyperproperty $\forall \pi_1 \dots \pi_k.\, J \rightarrow (\#_A\, \sigma.\, H \triangleleft n)$ can be expressed as a HyperLTL formula. For $\triangleleft \in \{\leq\}(\{<\})$, the HyperLTL formula has $n+k+1$(resp. $n+k$) universal trace quantifiers in addition to the quantifiers in J*

and H. For $\triangleleft \in \{\geq, >\}$, the HyperLTL formula has k universal trace quantifiers and n (resp. $n+1$) existential trace quantifiers in addition to the quantifiers in J and H. For $\triangleleft \in \{=\}$, the HyperLTL formula has $n+k+1$ universal trace quantifiers and n existential trace quantifiers in addition to the quantifiers in J and H.

Proof. For $\triangleleft = \ \leq$, we encode the counting hyperproperty $\forall \pi_1, \dots, \pi_k.\ J \rightarrow (\#_A\ \sigma.\ H) \triangleleft n$ as the following HyperLTL formula

$$\forall \pi_1, \dots, \pi_k.\ \forall \pi'_1, \dots, \pi'_{n+1}.\ \left(J \wedge \bigwedge_{i \neq j} \Diamond(\pi'_i \neq_A \pi'_j) \right) \rightarrow \left(\bigvee_i \neg H[\sigma \mapsto \pi'_i] \right)$$

where $\psi[\sigma \mapsto \pi'_i]$ is the HyperLTL formula ψ with all occurrences of σ replaced by π'_i. The formula states that there is no tuple of $n+1$ traces $\pi'_1, \dots, \pi'_{n+1}$ different in the evaluations of A, that satisfies H. In other words, for every $n+1$ tuple of traces $\pi'_1, \dots, \pi'_{n+1}$ that differ in the evaluation of A, one of the paths must violate ψ. For $\triangleleft = \ <$, we use the same formula, with $\forall \pi'_1, \dots, \pi'_n$ instead of $\forall \pi'_1, \dots, \pi'_{n+1}$.

For $\triangleleft = \geq$, we encode the counting hyperproperty analogously as the HyperLTL formula

$$\forall \pi_1, \dots, \pi_k.\ \exists \pi'_1, \dots, \pi'_n.\ J \rightarrow \left(\bigwedge_{i \neq j} \Diamond(\pi'_i \neq_A \pi'_j) \right) \wedge \left(\bigwedge_i H[\sigma \mapsto \pi'_i] \right)$$

The formula states that there exist paths $\pi'_1, \dots, \pi'_n$ that differ in the evaluation of A, and that all satisfy H. For $\triangleleft \in \{>\}$, we use the same formula, with $\exists \pi'_1, \dots, \pi'_{n+1}$ instead of $\exists \pi'_1, \dots, \pi'_n$.

Lastly, for $\triangleleft \in \{=\}$, we encode the counting hyperproperty as a conjunction of the encodings for $\leq$ and for $\geq$. □

Example 5.5 (Quantitative noninterference in HYPERLTL)**.** As discussed in Example 5.3, quantitative noninterference is the counting hyperproperty

$$\forall \pi.\ (\#_{PO}\ \sigma.\ \Box(\pi =_{PI} \sigma)) \leq n$$

This can be encoded in HyperLTL as the requirement that there are no $n+1$ traces distinguishable in their output for the same input

$$\forall \pi_0.\ \forall \pi_1 \dots \forall \pi_n.\ \left(\bigwedge_i \Box(\pi_i =_{PI} \pi_0) \right) \rightarrow \left(\bigvee_{i \neq j} \Box(\pi_i =_{PO} \pi_j) \right).$$

This formula is equivalent to the formalization of quantitative noninterference given in [66].

Example 5.6 (Quantitative deniability in HYPERLTL)**.** As discussed in Example 5.4, quantitative deniability is the counting hyperproperty

$$\forall \pi.\, (\#_{SI}\, \sigma.\, \square(\pi =_{PO} \sigma)) > n$$

Quantitative deniability can be given in HYPERLTL by the formula

$$\forall \pi_0.\, \exists \pi_1 \ldots \exists \pi_n.\, \left(\bigwedge_i \square(\pi_i =_{PO} \pi_0) \right) \wedge \left(\bigwedge_{i \neq j} \diamondsuit(\pi_i \neq_{SI} \pi_j) \right)$$

which encodes the requirement that there are at least $n+1$ traces distinguishable in their secret inputs for the same public output.

Model checking quantitative hyperproperties via the reduction to HyperLTL is very expensive. In the best case, when $\triangleleft \in \{\leq, <\}$, J does not contain existential quantifiers, and ψ does not contain universal quantifiers, we obtain an HyperLTL formula without quantifier alternations, where the number of quantifiers grows linearly with the bound n. For m quantifiers, the HyperLTL model checking algorithm [66] constructs and analyzes the m-fold self-composition of the Kripke structure. The running time of the model checking algorithm is thus exponential in the bound. If $\triangleleft \in \{\geq, >, =\}$, the encoding additionally introduces a quantifier alternation. The model checking algorithm checks quantifier alternations via a complementation of Büchi automata, which adds another exponent, resulting in an overall doubly exponential running time.

The model checking algorithm we introduce in the next section avoids the n-fold self-composition. It needs only logarithmic space in the bound, and therefore improves, depending on the property, exponentially or even doubly exponentially over the model checking algorithm of HyperLTL.

5.4.2 Model checking counting hyperproperties using maximum model counting

In this section, we present an algorithm for solving the model checking problem for counting hyperproperties based on a reduction to maximum model counting for linear-time properties. The algorithm builds on the following.

Let a transition system $T = (AP, I, O, S, s_0, \tau, L)$, HYPERLTL formulas J and H over a set of atomic propositions AP, a set $A \subseteq AP$, bounds $n, k \in \mathbb{N}$, and a comparison operator $\triangleleft \in \{<, \leq, =, \neq, >, \geq\}$. Let $\varphi = \forall \pi_1 \ldots \forall \pi_k.\, J \rightarrow$

$(\#_A\ \sigma.\ H) \triangleleft n$ be a counting hyperproperty where J is defined over $AP_{[\{\pi_1,\ldots,\pi_k\}]}$, and H is defined over $AP_{[\{\pi_1,\ldots,\pi_k,\sigma\}]}$. To check whether T violates φ, we need to find a set of traces $\{t_1,\ldots,t_k\} \subseteq \mathit{Traces}(T)$ that satisfies the hyperproperty J, and such that the size of the set

$$\Gamma = \{t_A \mid t \in \mathit{Traces}(T) \wedge \{t_1,\ldots,t_k,t\} \models H\}$$

satisfies the relation $\overline{\triangleleft}\ n$ where $\overline{\triangleleft}$ is the opposite operator of $\triangleleft$. The model checking problem thus reduces to maximizing (when $\triangleleft \in \{\leq,<\}$), or minimizing (when $\triangleleft \in \{>,\geq\}$) the size of Γ over all sets of k traces in T that satisfy J. This can be done in three steps:

- First, we need to capture the language of sets of traces $\{t_1,\ldots,t_k,t\}$ that satisfy the hyperproperty $J \wedge H$. This is done by building a Büchi automaton $B_{J\wedge H}$ for the HyperLTL formula $J \wedge H$ following the construction in Section 5.2.1. The automaton $B_{J\wedge H}$ is defined over the alphabet $2^{AP_{[\{\pi_1,\ldots,\pi_k,\sigma\}]}}$.

- Second, we construct a product automaton P of $B_{J\wedge H}$ and $k+1$ copies of T, corresponding to the trace variables $\pi_1,\ldots,\pi_k$, and σ, as follows

  - $P_0 = B_{J\wedge H}$
  - For some $0 \leq i < k$, if $P_i = (Q_i, \{q_{0,i}\}, 2^{AP_{\{\pi_1,\ldots,\pi_k,\sigma\}}}, \delta_i, F_i)$, then $P_{i+1} = (Q_i \times S \cup \{q_{0,i+1}\}, q_{0,i+1}, 2^{AP_{\{\pi_1,\ldots,\pi_k,\sigma\}}}, \delta_{i+1}, F_i \times S)$ where $q_{0,i+1}$ is a fresh state, and δ_{i+1} is defined as follows

$$\delta_{i+1}(q_{0,i+1},\alpha) = \{(q',s') \mid\ q' \in \delta_i(q_{0,i},\alpha) \wedge\ L(s_0)_{[\pi_i]} = \alpha_{O_{\pi_i}} \wedge\ s' \in \tau(s_0, (\alpha_{I_{\pi_i}})_{[a_{\pi_i} \in I_{\pi_i} \mapsto a]})\}$$

    and

$$\delta_{i+1}((q,s),\alpha) = \{(q',s') \mid\ q' \in \delta(q,\alpha) \wedge\ L(s)_{[\pi_i]} = \alpha_{O_{\pi_i}} \wedge\ s' \in \tau(s, (\alpha_{I_{\pi_i}})_{[a_{\pi_i} \in I_{\pi} \mapsto a]})\}$$

    We define the product automaton P as the automaton P_k.

- Third, given the product automaton P, we define the following maximum (or minimum) model counting problem over P. Let AP_H be all the atomic propositions that appear H (with their trace variable indices). The the

maximum (minimum) model counting is defined over the counting set $A_{\{\sigma\}}$, and maximization (minimization) set $AP_H \setminus AP_{\{\sigma\}}$.

We illustrate the advantages of the new algorithm over the standard HYPERLTL model checking algorithm by comparing the complexities for model checking the counting hyperproperties of quantitative noninterference and deniability.

Example 5.7 (Model checking quantitative noninterference)**.** Quantitative noninterference is defined by the counting hyperproperty

$$\varphi = \forall \pi.\, (\#_{PO}\, \sigma.\, \square(\pi =_{PI} \sigma)) \leq n$$

which in turn can be encoded as the HYPERLTL formula

$$\varphi = \forall \pi_0.\, \forall \pi_1 \ldots \forall \pi_n.\, \left(\bigwedge_i \square(\pi_i =_I \pi_0) \right) \rightarrow \left(\bigvee_{i \neq j} \square(\pi_i =_O \pi_j) \right).$$

Using the standard model checking algorithm for HYPERLTL, we need to construct a n-self composition of the transition T system to be model checked against φ. The size of the self-composition is exponential in n, and thus the model checking algorithm runs in time exponential, and space polynomial in the n.

In contrast, the maximum-model-counting-based algorithm requires only space logarithmic, and time polynomial in the bound n. The reason for that is, that we only need to maintain binary encoded counter that counts up to n. As soon as the maximum model counting algorithm reaches this bound, we can terminate the procedure and return a violation for the counting property. For quantitative noninterference, there is however a tradeoff in the complexity in the size of the system T. The maximum model counting algorithms runs in space polynomial in the size of the product automaton (Section 3.6), which is of size polynomial in the size of T, in contrast to a logarithmic space complexity in T (Theorem 5.1).

Example 5.8 (Model checking quantitative deniability)**.** Quantitative deniability is defined by the counting hyperproperty

$$\forall \pi.\, (\#_{SI}\, \sigma.\, \square(\pi =_{PO} \sigma)) > n$$

which in turn can be encoded as the HYPERLTL formula

$$\forall \pi_0.\, \exists \pi_1 \ldots \exists \pi_n.\, \left(\square(\bigwedge_{o \in PO} o_\pi = o_{\pi'}) \right) \rightarrow \left(\Diamond(\bigvee_{i \in SI} i_{\pi_1} \neq i_{\pi_2}) \right)$$

Using the standard model checking algorithm for HYPERLTL, we need to a product automaton composed of the Büchi automaton for the formula

$$\left(\square(\bigwedge_{o \in PO} o_\pi = o_{\pi'})\right) \rightarrow \left(\diamondsuit(\bigvee_{i \in SI} i_{\pi_1} \neq i_{\pi_2})\right)$$

with n-self composition of the transition T, a complementation of this automaton, and computing a final product of the complement automaton and the transitions system T. The size of the resulting automaton is doubly-exponential in n, and thus the model checking algorithm runs in time doubly-exponential, and space exponential in the n.

In contrast, a minimum-model-counting-based algorithm requires only space logarithmic, and time polynomial in the bound n, as we only need to maintain a counter of $\log(n)$ bits. With respect to the complexity in the size of the transition system, the algorithm runs, as in the case of quantitative noninterference, only in polynomial space in the size of T. This is also the case for the standard model checking algorithm, because the alternation depth in the HYPERLTL formula is 1 (Theorem 5.2).

The complexity of the model checking algorithm is summarized in the following theorem.

Theorem 5.4. *Given a transition system T and a quantitative hyperproperty φ with bound n, deciding whether $T \models \varphi$ can be done in logarithmic space in the bound n, and in polynomial space in the size of T.*

5.5 Symbolic Approach to Model Checking Counting Properties

For existential HYPERLTL formulas J and H, we give a model checking approach by encoding the automaton-based construction presented into a propositional formula.

Given a transitions system $= (AP, I, O, S, s_0, \tau, L)$, and a quantitative hyperproperty $\varphi = \forall \pi_1, \ldots, \pi_k.\ J \rightarrow (\#_A \sigma.\ H) \triangleleft n$ over a set of atomic propositions $AP_\varphi \subseteq AP$ and bound μ (defined below), our algorithm constructs a propositional formula ϕ such that, every satisfying assignment of ϕ uniquely encodes a tuple of lassos $(\pi_1, \ldots, \pi_k, \sigma)$ in T of length μ, where $(\pi_1, \ldots, \pi_k)$ satisfies J and $(\pi_1, \ldots, \pi_k, \sigma)$ satisfies H.

To compute the values

$$\max_{(\pi_1, \ldots, \pi_k)} |\{\sigma_A \mid (\pi_1, \ldots, \pi_k, \sigma) \models J \wedge H\}|$$

in case $\triangleleft \in \{\leq, <\}$, or

$$\min_{(\pi_1,\ldots,\pi_k)} |\{\sigma_A \mid (\pi_1, \ldots, \pi_k, \sigma) \models J \wedge H\}|$$

in case $\triangleleft \in \{\geq, >\}$, we apply a maximum model counter or a minimal model counter on ϕ, respectively, with the appropriate sets of counting and maximization, or minimization propositions. From Lemma 3.2 we know that it is enough to consider lassos of length linear in the product automaton. The size of ϕ is thus exponential in the size of φ and polynomial in the size of T.

The construction resembles the encoding of the bounded model checking approach for LTL [22]. Let $J = \exists \pi'_1 \ldots \pi'_{k'}.\ J'$ and $H = \exists \pi''_1 \ldots \pi''_{k''}.\ H''$ and let AP_J and AP_H be the sets of atomic propositions that appear in J and H respectively. The propositional formula ϕ is given as a conjunction of the following propositional formulas:

$$\phi = \bigwedge_{i \leq k} [\![T]\!]^{\pi_i}_{\mu} \wedge [\![T]\!]^{\sigma}_{\mu} \wedge [\![J]\!]^{0}_{\mu} \wedge [\![H]\!]^{0}_{\mu}$$

where:

- μ is length of considered lassos and is equal to $\mu = 2^{|\psi'_i \wedge \psi''|} * |S|^{k+k'+k''+1} + 1$, which is one plus the size of the product automaton constructed from the $k + k' + k'' + 1$ self-composition and the automaton for $J \wedge H$. The "plus one" is to additionally to check whether the number of models is infinite.

- $[\![T]\!]^{\pi}_{k}$ is the encoding of the transition relation of the copy of T where atomic propositions are indexed with π and up to an unrolling of length k. Each state of T can be encoded as an evaluation of a vector of $\log |S|$ unique propositional variables. The encoding[3] is given by the propositional formula $I(\vec{v}^{\pi}_0) \wedge \bigwedge_{i=0}^{k-1} \tau(\vec{v}^{\pi}_i, \vec{v}^{\pi}_{i+1})$ which encodes all paths of T of length k. The formula $I(\vec{v}^{\pi}_0)$ defines the assignment of the initial state. The formulas $\tau(\vec{v}^{\pi}_i, \vec{v}^{\pi}_{i+1})$ define valid transitions in T from the ith to the $(i+1)$st state of a path.

- $[\![J]\!]^{0}_{k}$ and $[\![H]\!]^{0}_{k}$ are constructed using the following rules:

[3] We refer the reader to [22] for more information on the encoding

	$i < k$	$i = k$
$[\![a_\pi]\!]^i_k$	a^i_π	$\bigvee_{j=0}^{k-1}(l_j \wedge a^j_\pi)$
$[\![\neg a_\pi]\!]^i_k$	$\neg a^i_\pi$	$\bigvee_{j=0}^{k-1}(l_j \wedge \neg a^j_\pi)$
$[\![\bigcirc \varphi_1]\!]^i_k$	$[\![\varphi_1]\!]^{i+1}_k$	$\bigvee_{j=0}^{k-1}(l_j \wedge [\![\varphi_1]\!]^j_k)$
$[\![\varphi_1 \,\mathcal{U}\, \varphi_2]\!]^i_k$	$[\![\varphi_2]\!]^i_k \vee ([\![\varphi_1]\!]^i_k \wedge [\![\varphi_1 \,\mathcal{U}\, \varphi]\!]^{i+1}_k)$	$\bigvee_{j=0}^{k-1}(l_j \wedge \langle \varphi_1 \,\mathcal{U}\, \varphi_2 \rangle^j_k)$
$\langle \varphi_1 \,\mathcal{U}\, \varphi_2 \rangle^i_k$	$[\![\varphi_2]\!]^i_k \vee ([\![\varphi_1]\!]^i_k \wedge \langle \varphi_1 \,\mathcal{U}\, \varphi \rangle^{i+1}_k)$	false
$[\![\varphi_1 \,\mathcal{R}\, \varphi_2]\!]^i_k$	$[\![\varphi_2]\!]^i_k \wedge ([\![\varphi_1]\!]^i_k \vee [\![\varphi_1 \,\mathcal{R}\, \varphi]\!]^{i+1}_k)$	$\bigvee_{j=0}^{k-1}(l_j \wedge \langle \varphi_1 \,\mathcal{R}\, \varphi_2 \rangle^j_k)$
$\langle \varphi_1 \,\mathcal{R}\, \varphi_2 \rangle^i_k$	$[\![\varphi_2]\!]^i_k \wedge ([\![\varphi_1]\!]^i_k \vee \langle \varphi_1 \,\mathcal{R}\, \varphi \rangle^{i+1}_k)$	true

in case of an existential quantifier over a trace variable π, we add a copy of the encoding of T with new variables distinguished by π:

$[\![\exists \pi.\varphi_1]\!]^i_k$	$[\![T]\!]^\pi_k \wedge [\![\varphi_1]\!]^i_k$

We define sets $X = \{a^i_\sigma \mid a \in A, i \leq k\}$, $Y = \{a^i \mid a \in AP_H \setminus A, i \leq k\}$ and $Z = P \setminus X \cup Y$, where P is the set of all propositions in ϕ. The maximum model counting problem is then $\#^{\max}(\phi, X, Y, Z)$.

5.5.1 Evaluation

We have implemented the symbolic approach from the last section and compared it to the expansion-based approach of HyperLTL [66]. Our implementation uses the MaxCount tool [73]. We use the option in MaxCount that enumerates, rather than approximates, the number of assignments for the counting variables. We furthermore instrumented the tool so that it terminates as soon as a sample is found that exceeds the given bound. If no sample is found after one hour, we report a timeout.

Table 5.1 shows the results on a parameterized benchmark obtained from the implementation of an 8bit passcode checker. The parameter of the benchmark is the bound on the number of bits that is leaked to an adversary, who might, for example, enter passcodes in a brute-force manner. In all instances, a violation is found. The results show that the Max#Sat-based approach scales significantly better than the expansion-based approach.

5.6 Bibliographic Remarks

Quantitative information-flow has been studied extensively in the literature [139, 96, 45, 6, 79]. Multiple verification methods for quantitative information-flow

Specification	MCHyper				MCQHyper		
	#Latches	#Gates	#Quan.	Time(sec)	#max	#count	Time(sec).
1bit_leak	9	55	2	0.3	16	2	1
2bit_leak			4	0.4	32	4	1
3bit_leak			8	1.3	64	8	2
4bit_leak			16	97	128	16	4
5bit_leak			32	TO	256	32	8
6bit_leak			64	TO	512	64	335
8bit_leak			256	TO	2048	256	TO

Table 5.1: Comparison between the expansion-based approach (MCHyper) and the Max#Sat-based approach (MCQHyper). #max is the number of maximization variables (set Y). #count is the number of the counting variables (set X). TO indicates a time-out after 1 hour.

were proposed for sequential systems. For example, with static analysis techniques [41], approximation methods [98], equivalence relations [10, 48], and randomized methods [98]. Quantitative information-flow for multi-threaded programs was considered in [37].

The study of quantitative information-flow in a reactive setting gained a lot of attention recently after the introduction of hyperproperties [46] and the idea of verifying the self-composition of a reactive system [16] in order to relate traces to each other. There are several possibilities to measure the amount of leakage, such as Shannon entropy [137, 56, 41, 115], and the ones we used in this chapter, guessing entropy [96], min-entropy [139]. A classification of quantitative information-flow policies as safety and liveness hyperproperties was given in [155]. While several verification techniques for hyperproperties exists [15, 80, 117, 122], the literature was missing general approaches to quantitative information-flow control. SecLTL [57] was introduced as first general approach to model check (quantitative) hyperproperties, before HyperLTL [44], and its corresponding model checker [66], was introduced as a temporal logic for hyperproperties, which subsumes the previous approaches.

Among the already existing tools for computing the amount of information leakage, for example, `QUAIL` [24], which analyzes programs written in a specific while-language and `LeakWatch` [39], which estimates the amount of leakage in Java programs, `Moped-QLeak` [31] is closest to our approach. However, their approach of computing a symbolic summary as an Algebraic Decision Diagram is, in contrast to our approach, solely based on model counting, not maximum model counting.

Chapter 6

Synthesis of Approximate Implementations

6.1 Synthesis of Reactive Systems

In formal synthesis of reactive systems an implementation of a system is automatically constructed from its formal specification [63, 125]. The great advantage of synthesis is that the resulting implementation is correct by construction, and thus allows developers to focus on determining *what* a system should do rather than *how* it should do it.

In synthesis, the task of the developer is shifted from writing a program that implements the system to writing a specification for it. This comes with the main hurdle that the system designer has to provide the right formal specification, which as we have discussed earlier is often a difficult task [100]. In particular, since the system being synthesized is required to satisfy its requirements against all possible environments allowed by the specification, accurately capturing the designer's knowledge about the environment in which the system will execute is crucial for being able to successfully synthesize an implementation. Missing assumptions about the environment lead quickly to unrealizable specifications, i.e., specifications for which there is no implementation that satisfies them.

Traditionally, environment assumptions are included in the specification. One way to repair an unrealizable specification is by refining these assumptions or weakening the guarantees expected of the system [133]. There are, however less explored ways of incorporating information about the environment, for example, one of which is to consider a *bound on the size of the environment*, that is, a bound on the size of the state space of a transition system that describes the possible environment behaviors. Restricting the space of possible environments can render an unrealizable specification into a realizable one.

The synthesis under such bounded environments was first studied in [101], where the authors extensively study the problem, in several versions, from the complexity-theoretic point of view. In this chapter, we present another approach to providing environment assumptions, where instead of bounding the size of the state space of the environment, we bound the sequences of values of input signals produced by the environment. The infinite input sequences produced by a finite-state environment which interacts with a finite state system are ultimately periodic, and thus, each such infinite sequence $\sigma \in \Sigma_I^\omega$, over the input alphabet Σ_I, can be represented as a *lasso*, which is a pair (u, v) of finite words $u \in \Sigma_I^*$ and $v \in \Sigma_I^+$, such that $\sigma = u \cdot v^\omega$. It is the length of such sequences that we consider a bound on. More precisely, given a bound $k \in \mathbb{N}$, we consider the language of all infinite sequences of inputs that can be represented by a lasso (u, v) with $|u \cdot v| = k$. The goal of the *synthesis of lasso precise implementations* is then to synthesize a system for which each execution resulting from a sequence of environment inputs in that language, satisfies a given linear-time property.

Example. Consider an arbiter serving two client processes. Each client issues a request when it wants to access a shared resource, and keeps the request signal up until it is done using the resource. The goal of the arbiter is to ensure the classical mutual exclusion property, by not granting access to the two clients simultaneously. The arbiter has to also ensure that each client request is eventually granted. This, however, is difficult since, first, a client might gain access to the resource and never lower the request signal, and second, the arbiter is not allowed to take away a grant unless the request has been set to false, or the client never sets the request to false in the future (the client has become unresponsive). The last two requirements together make the specification unrealizable, as the arbiter has no way of determining if a client has become unresponsive, or will lower the request signal in the future. If, however, the length of the lassos of the input sequences is bounded, then, after a sufficient number of steps, the arbiter can assume that if the request has not been set to false, then it will not be low-

ered in the future either, as the sequence of inputs must already have run at least once through it's period that will be ultimately repeated from that point on.

Formally, we can express the requirements on the arbiter in LTL as follows. There is one input variable r_i (for *request*) and one output variable g_i (for grant) associated with each client. The specification is then given as the conjunction

$$\varphi = \varphi_{mutex} \wedge \varphi_{resp} \wedge \varphi_{rel}$$

where

$$\begin{aligned} \varphi_{mutex} &= \Box\neg(g_1 \wedge g_2), \\ \varphi_{resp} &= \Box\bigwedge_{i=1}^{2}(r_i \rightarrow \Diamond g_i), \\ \varphi_{rel} &= \Box\bigwedge_{i=1}^{2}(g_i \wedge r_i \wedge (\Diamond\neg r_i) \rightarrow \bigcirc g_i). \end{aligned}$$

Due to the requirement to not revoke grants stated in φ_{rel}, the specification φ is unrealizable. For any bound k on the length of the input lassos, however, φ is realizable. More precisely, there exists an implementation in which once client i has not lowered the request signal for k consecutive steps, the variable g_i is set to false. This example shows that when the system designer has knowledge about the resources available to the environment processes, taking this knowledge into account can enable us to synthesize a system that is correct under this assumption.

We formally define the synthesis problem for *lasso-precise implementations*, that is, implementations that are correct for input lassos of bounded size, and describe an automata-theoretic approach to this synthesis problem. We also consider the synthesis of *lasso-precise implementations of bounded size*, and provide a symbolic synthesis algorithm based on quantified Boolean satisfiability.

Bounding the size of the input lassos can render some unrealizable specifications realizable, but, similarly to bounding the size of the environment, comes at the price of higher computational complexity. To alleviate this problem, we further study the synthesis of *approximate implementations*, where we relax the synthesis problem further, and only require that for a given $\epsilon > 0$ the ratio of input lassos of a given size for which the specification is satisfied, to the total number of input lassos of that size is at least $1 - \epsilon$. We then propose an *approximate synthesis method* based on maximum model counting for Boolean formulas [73]. The benefits of the approximate approach are two-fold. Firstly, it can often deliver high-quality approximate solutions more efficiently than the lasso-precise synthesis method, and secondly, even when the specification is still unrealizable for a given lasso bound, we might be able to synthesize an implementation that is correct for a given fraction of the possible input lassos.

6.2 Lasso-precise implementations

We begin by formally defining the language of sequences of input values representable by lassos of a given length k. For the rest of the section, we consider linear-time properties defined over a set of atomic propositions AP. The subset $I \subseteq AP$ consists of the input atomic propositions controlled by the environment.

Definition 6.1 (Bounded Model Languages)**.** Let φ be a linear-time property over a set of atomic propositions AP, let $\Sigma = 2^{AP}$, and let $I \subseteq AP$.

We say that an infinite word $\sigma \in \Sigma^\omega$ is an I-k-model of φ, for a bound $k \in \mathbb{N}$, if and only if $\sigma \in L(\varphi)$, and there are words $u \in (2^I)^*$ and $v \in (2^I)^+$ such that $|u \cdot v| = k$ and $\sigma|_I = u \cdot v^\omega$. The language of I-k-models of the property φ is defined by the set $L_k^I(\varphi) = \{\sigma \in \Sigma^\omega \mid \sigma \text{ is a } I\text{-}k\text{-model of } \varphi\}$.

Note that a model of φ might be induced by lassos of different length and by more than one lasso of the same length, e.g, a^ω is induced by (a, a) and (ϵ, aa). The next lemma establishes that if a model of φ can be represented by a lasso of length k then it can also be represented by a lasso of any larger length.

Lemma 6.1. *For a linear-time property φ over $\Sigma = 2^{AP}$, subset $I \subseteq AP$ of atomic propositions, and bound $k \in \mathbb{N}$, we have $L_k^I(\varphi) \subseteq L_{k'}^I(\varphi)$ for all $k' > k$.*

Proof. Let $\sigma \in L_k^I(\varphi)$. Then, $\sigma \models \varphi$ and there exists $(u, v) \in (2^I)^* \times (2^I)^+$ such that $|u \cdot v| = k$ and $\sigma|_I = u \cdot v^\omega$. Let $v = v_1 \dots v_k$. Since $u \cdot v_1(v_2 \dots v_k v_1)^\omega = u \cdot (v_1 \dots v_k)^\omega = \sigma|_I$, we have $\sigma \in L_{k+1}^I(\varphi)$. The claim follows by induction. □

Using the definition of I-k-models, the language of infinite sequences of environment inputs representable by lassos of length k can be expressed as $L_k^I(\Sigma^\omega)$.

Definition 6.2 (k-lasso-precise Implementations)**.** For a linear-time property φ over $\Sigma = 2^{AP}$, subset $I \subseteq AP$ of atomic propositions, and bound $k \in \mathbb{N}$, we say that a transition system T is a k-lasso-precise implementation of φ, denoted $T \models_{k,I} \varphi$, if it holds that $L_k^I(\mathit{Traces}(T)) \subseteq \varphi$.

That is, in a k-lasso-precise implementation T, all the traces of T that belong to the language $L_k^I(\Sigma^\omega)$ are I-k-models of the specification φ.

Using the last definitions, we can formally define the synthesis problem of lasso-precise implementations as follows

Definition 6.3 (Synthesis of lasso-precise implementations)**.** Given a linear-time property φ over atomic propositions AP with input atomic propositions I, and given a bound $k \in \mathbb{N}$, construct an implementation T such that $T \models_{k,I} \varphi$, or determine that such an implementation does not exist.

We establish the relationship between the synthesis of lasso-precise implementations and synthesis under bounded environments. The *synthesis problem for bounded environments* asks for a given linear-time property φ and a bound $k \in \mathbb{N}$ to synthesize a transition system T such that for every possible environment E of size at most k, the transition system T satisfies φ under environment E, i.e., $T \models_E \varphi$. Intuitively, the two synthesis problems can be reduced to each other since an environment of a given size, interacting with a given implementation, can only produce ultimately periodic sequences of inputs representable by lassos of length determined by the sizes of the environment and the implementation. This intuition is formalized in the following proposition, stating the connection between the two problems.

Proposition 6.1. *Given a specification φ over a set of atomic propositions AP with subset $I \subseteq AP$ of atomic propositions controlled by the environment, and a bound $k \in \mathbb{N}$, for every transition system T the following statements hold:*

(1) If $T \models_E \varphi$ for all environments E of size at most k, then $T \models_{k,I} \varphi$.

(2) If $T \models_{k\cdot|T|,I} \varphi$, then $T \models_E \varphi$ for all environments E of size at most k.

Proof. For (1), let T be a transition system such that $T \models_E \varphi$ for all environments E of size at most k. Assume, for the sake of contradiction, that $T \not\models_{k,I} \varphi$. Thus, that there exists a word $\sigma \in \mathit{Traces}(T)$, such that $\sigma \in L_k^I(\Sigma^\omega)$ and $\sigma \not\models \varphi$. Since $\sigma \in L_k^I(\Sigma^\omega)$, we can construct an environment E of size at most k that produces the sequence of inputs $\sigma|_I$. Since E is of size at most k, we have that $T \models_E \varphi$. Thus, since $\sigma \in \mathit{Traces}(T \times E)$, we have $\sigma \models \varphi$, which is a contradiction.

For (2), let T be a transition system such that $T \models_{k\cdot|T|,I} \varphi$. Assume, for the sake of contradiction that there exists an environment E of size at most k such that $T \not\models_E \varphi$. Since $T \not\models_E \varphi$, there exists $\sigma \in \mathit{Traces}(T \times E)$ such that $\sigma \not\models \varphi$. As the number of states of E is at most k, the input sequences it generates can be represented as lassos of size $k \cdot |T|$. Thus, $\sigma \in L^I_{k\cdot|T|}(\Sigma^\omega)$. This is a contradiction with the choice of T, according to which $T \models_{k\cdot|T|,I} \varphi$. □

6.3 Automata-theoretic synthesis of lasso-precise implementations

We now provide an automata-theoretic algorithm for the synthesis of lasso-precise implementations. The underlying idea of this approach is to first construct an automaton over finite traces that accepts all finite prefixes of traces in $L_k^I(\Sigma^\omega)$. Then, combining this automaton and an automaton representing the property φ, we can construct an automaton whose language is non-empty if and only if there exists an k-lasso-precise implementation of φ.

The next theorem presents the construction of a deterministic finite automaton for the language $\mathit{Prefix}(L_k^I(\Sigma^\omega))$.

Theorem 6.1. *For any set AP of atomic propositions, subset $I \subseteq AP$, and bound $k \in \mathbb{N}$ there is a deterministic finite automaton A_k over alphabet $\Sigma = 2^{AP}$, with size $(2^{|I|}+1)^k \cdot (k+1)^k$, such that $L(A_k) = \{w \in \Sigma^* \mid \exists \sigma \in L_k^I(\Sigma^\omega).\ w < \sigma\}$.*

Idea & Construction. For given $k \in \mathbb{N}$ we first define an automaton $\widehat{A}_k = (Q, q_0, \delta, F)$ over $\widehat{\Sigma} = 2^I$, such that $L(\widehat{A}_k) = \{\widehat{w} \in \widehat{\Sigma}^* \mid \exists \widehat{\sigma} \in L_k^I(\widehat{\Sigma}^\omega).\ \widehat{w} < \widehat{\sigma}\}$. That, is $L(\widehat{A}_k)$ is the set of all finite prefixes of infinite words over $\widehat{\Sigma}$ that can be represented by a lasso of length k. We can then define the automaton A_k as the automaton that for each $w \in \Sigma^*$ simulates $\widehat{A}_k$ on the projection $w|_I$ of w. We define the automaton $\widehat{A}_k = (Q, q_0, \delta, F)$ such that

- $Q = (\widehat{\Sigma} \cup \{\#\})^k \times \{-, 1, \ldots, k\}^k$,
- $q_0 = (\#^k, (1, 2, \ldots, k))$,
- $\delta(q, \alpha) = \begin{cases} (w \cdot \alpha \cdot \#^{m-1}, t) & \text{if } q = (w \cdot \#^m, t) \text{ where } 1 \leq m \leq k, \\ & \quad w \in \widehat{\Sigma}^{(k-m)},\ t \in \{-, 1, \ldots, k\}^k \\ (w, (i'_1, \ldots, i'_k)) & \text{if } q = (w, (i_1, \ldots, i_k)) \text{ where } w \in \widehat{\Sigma}^k, \text{ and} \\ & \quad i'_j = \begin{cases} - & i_j \leq k \wedge w(i_j) \neq \alpha \text{ or } i_j = - \\ i_j + 1 & i_j < k \wedge w(i_j) = \alpha \\ j & i_j = k \wedge w(i_j) = \alpha \end{cases} \end{cases}$
- $F = Q \setminus \{(w, (-, \ldots, -)) \mid w \in \widehat{\Sigma}^k\}$.

Proof. States of the form $(w \cdot \alpha \cdot \#^m, t)$ with $m \geq 1$ store the portion of the input word read so far, for input words of length smaller than k. In states of this form we have $t = (1, 2, \ldots, k)$, which implies that all such states are accepting. In turn, this means that A_k accepts all words of length smaller or equal to k. This is justified by the fact that, each word of length smaller or equal to k is a prefix of an infinite word in $L_k^I(\widehat{\Sigma}^\omega)$, obtained by repeating the prefix infinitely often. Now, let us consider words of length greater than k.

In states of the form $(u, (i_1, \ldots, i_k))$, with $u \in \widehat{\Sigma}^*$, the word u stores the first k letters of the input word. Intuitively, the tuple $(i_1, \ldots, i_k)$ stores the information about the loops that are still possible, given the portion of the input word that is read thus far. To see this, let us consider a word $w \in \widehat{\Sigma}^*$ such that $|w| = l > k$, and let $q_0 q_1 \ldots q_l$ be the run of A_k on w. The state q_l is of the form $q_l = (w(1) \ldots w(k), (i_1^l, \ldots, i_k^l))$. It can be shown by induction on l that for each j we have $i_j^l \neq -$ if and only if w is of the form $w = w' \cdot w'' \cdot w'''$ where $w' = w(1) \ldots w(j-1)$, $w'' = (w(j) \ldots w(k))^k$ for some $k \geq 0$, and $w''' = (w(j) \ldots w(i_j^l - 1))$. Thus, if $i_j^l \neq -$, then it is possible to have a loop starting at position j, and i_j^l is such that $(w(j) \ldots w(i_j^l - 1))$ is the prefix of $w(j) \ldots w(k)$ appearing after the (possibly empty) sequence of repetitions of $w(j) \ldots w(k)$. This means, that if $i_j^l \neq -$, then w is a prefix of the infinite word $w' \cdot (w'')^\omega \in L_k^I(\widehat{\Sigma}^\omega)$. Therefore, if the run of A_k on a word w with $|w| > k$ is accepting, then there exists $\sigma \in L_k^I(\widehat{\Sigma}^\omega)$ such that $w < \sigma$.

For the other direction, suppose that for each j, we have $i_j^l = -$. Take any j, and consider the first position m in the run $q_0 q_1 \ldots q_l$ where $i_j^m = -$. By the definition of δ we have that $w(m) \neq w(i_j^{m-1})$. This means that the prefix $w(1) \ldots w(m)$ cannot be extended to the word $w(1) \ldots w(j-1)(w(j) \ldots w(k))^\omega$. Since for every $j \in \{1, \ldots, k\}$ we can find such a position m, it holds that there does not exist $\sigma \in L_k^I(\widehat{\Sigma}^\omega)$ such that $w < \sigma$. This concludes the proof. □

The automaton constructed in the previous theorem has size which is exponential in the length of the lassos. In the next theorem we show that this exponential blow-up is unavoidable. That is, we show that every nondeterministic finite automaton for the language $\mathit{Prefix}(L_k^I(\Sigma^\omega))$ is of size at least $2^{\Omega(k)}$.

Theorem 6.2. *For any bound $k \in \mathbb{N}$ and sets of atomic propositions AP and $\emptyset \neq I \subseteq AP$, every nondeterministic finite automaton N over the alphabet $\Sigma = 2^{AP}$ that recognizes $L = \{w \in \Sigma^* \mid \exists \sigma \in L_k^I(\Sigma^\omega).\ w < \sigma\}$ is of size at least $2^{\Omega(k)}$.*

Proof. Let $N = (Q, Q_0, \delta, F)$ be a nondeterministic finite automaton for L. For each $w \in \Sigma^k$, we have that $w \cdot w \in L$. Therefore, for each $w \in \Sigma^k$ there exists at least one accepting run $\rho = q_0 q_1 \dots q_f$ of N on $w \cdot w$. We denote with $q(\rho, m)$ the state q_m that appears at the position indexed m of a run ρ.

Let $a \in 2^I$ be a letter in 2^I, and let $\Sigma' = \Sigma \setminus \{a' \in \Sigma \mid a'|_I = a\}$. Let $L' \subseteq L$ be the language $L' = \{w \in \Sigma^k \mid \exists w' \in (\Sigma')^{k-1}, a' \in \Sigma : \ w = w' \cdot a' \text{ and } a'|_I = a\}$. That is, L' consists of the words of length k in which letters a' with $a'|_I = a$ appear in the last position and only in the last position.

Let us define the set of states

$$Q_k = \{q(\rho, k) \mid \exists w \in L' : \ \rho \text{ is an accepting run of } N \text{ on } w \cdot w\}.$$

That is, Q_k consists of the states that appear at position k on some accepting run on some word $w \cdot w$, where w is from L'. We will show that $|Q_k| \geq 2^{k-1}$.

Assume that this does not hold, i.e., $|Q_k| < 2^{k-1}$. Since $|L'| \geq 2^{k-1}$, this implies that there exist $w_1, w_2 \in L'$, such that $w_1|_I \neq w_2|_I$ and there exists accepting runs ρ_1 and ρ_2 of N on $w_1 \cdot w_1$ and $w_2 \cdot w_2$ respectively, such that $q(\rho_1, k) = q(\rho_2, k)$. That is, there must be two words in L' with $w_1|_I \neq w_2|_I$, which have accepting runs on $w_1 \cdot w_1$ and $w_2 \cdot w_2$ visiting the same state at position k.

We now construct a run $\rho_{1,2}$ on the word $w_1 \cdot w_2$ that follows ρ_1 for the first k steps on w_1, ending in state $q(\rho_1, k)$, and from there on follows ρ_2 on w_2. It is easy to see that $\rho_{1,2}$ is a run on the word $w_1 \cdot w_2$. The run is accepting, since ρ_2 is accepting. This means that $w_1 \cdot w_2 \in L$, which we will show leads to contradiction.

To see this, recall that $w_1 = w_1' \cdot a'$ and $w_2 = w_2' \cdot a''$, and $w_1|_I \neq w_2|_I$, and $a'|_I = a''|_I = a$. Since $w_1 \cdot w_2 \in L$, we have that $w_1' \cdot a' \cdot w_2' \cdot a'' < \sigma$ for some $\sigma \in L_k^I(\Sigma^\omega)$. That is, there exists a lasso for some word σ, and $w_1' \cdot a' \cdot w_2' \cdot a''$ is a prefix of this word. Since a does not appear in $w_2'|_I$, this means that the loop in this lasso is the whole word $w_1|_I$, which is not possible, since $w_1|_I \neq w_2|_I$.

This is a contradiction, which shows that $|Q| \geq |Q_k| \geq 2^{k-1}$. Since $\mathcal{N}$ was an arbitrary nondeterministic finite automaton for L, this implies that the minimal automaton for L has at least $2^{\Omega(k)}$ states, which concludes the proof. □

Using the automaton from Theorem 6.1, we can transform every property automaton A into an automaton that accepts words representable by lassos of length less than or equal to k if and only if they are in $L(A)$, and accepts all words that are not representable by lassos of length less than or equal to k.

Theorem 6.3. *Let AP be a set of atomic propositions, and let $I \subseteq AP$. For every (deterministic, nondeterministic or alternating) parity automaton A over $\Sigma = 2^{AP}$, and $k \in \mathbb{N}$, there is a (deterministic, nondeterministic or alternating) parity automaton A' of size $2^{O(k)} \cdot |A|$, s.t., $L(A') = (L_k^I(\Sigma^\omega) \cap L(A)) \cup (\Sigma^\omega \setminus L_k^I(\Sigma^\omega))$.*

Proof. The theorem is a consequence of Theorem 6.1 established as follows. Let $A = (Q, Q_0, \delta, \mu)$ be a parity automaton, and let $D = (\widehat{Q}, \widehat{q}_0, \widehat{\delta}, F)$ be the deterministic finite automaton for bound k defined as in Theorem 6.1. We define the parity automaton $A = (Q', Q_0', \delta', \mu')$ with the following components:

- $Q' = (Q \times \widehat{Q})$;
- $Q_0' = \{(q_0, \widehat{q}_0) \mid q_0 \in Q_0\}$ (when A is deterministic Q_0' is a singleton set);
- $\delta'((q, \widehat{q}), \alpha) = \delta(q, \alpha)_{[q'/(q', \widehat{\delta}(\widehat{q}, \alpha))]}$, where $\delta(q, \alpha)_{[q'/(q', \widehat{q}')]}$ is the Boolean expression obtained from $\delta(q, \alpha)$ by replacing every state q' by the state $(q', \widehat{q}')$;
- $\mu'((q, \widehat{q})) = \begin{cases} \mu(q) & \text{if } \widehat{q} \in F, \\ 0 & \text{if } \widehat{q} \notin F. \end{cases}$

Intuitively, the automaton A' is constructed as the product of A and D, where runs entering a state in D that is not accepting in D are accepting in A'. To see this, recall from the construction in Theorem 6.1 that once D enters a state in $\widehat{Q} \setminus \widehat{F}$ it remains in such a state forever. Thus, by setting the color of all states $(q, \widehat{q})$ where $\widehat{q} \notin F$ to 0, we ensure that words containing a prefix rejected by D have only runs in which the highest color appearing infinitely often is 0. Thus, we ensure that all words that are not representable by lassos of length less than or equal to k are accepted by A', while words representable by lassos of length less than or equal to k are accepted if and only if they are in $L(A)$. □

The following theorem is a consequence of the one above, and of the complexity of synthesis fo deterministic parity automata [125], and provides us with an automata-theoretic approach to solving the lasso-precise synthesis problem.

Theorem 6.4 (Synthesis)**.** *Let AP be a set of atomic propositions, and $I \subseteq AP$ be a subset of AP consisting of the atomic propositions controlled by the environment. For a specification, given as a deterministic parity automaton $\mathcal{P}$ over the alphabet $\Sigma = 2^{AP}$, and a bound $k \in \mathbb{N}$, finding an implementation T, such that, $T \models_{k,I} \mathcal{P}$ can be done in time polynomial in the size of the automaton $\mathcal{P}$ and exponential in the bound k.*

6.4 Bounded Synthesis of Lasso-precise Implementations

For a specification φ given as an LTL formula, a bound n on the size of the synthesized implementation and a bound k on the lassos of input sequences, *bounded synthesis of lasso-precise implementations* searches for an implementation T of size n, such that $T \models_{k,I} \varphi$. Using the automata constructions in the previous section we can construct a universal co-Büchi automaton for the language $L_k^I(\varphi) \cup (\Sigma^\omega \setminus L_k^I(\Sigma^\omega))$ and construct the constraint system as presented in [67]. This constraint system is exponential in both $|\varphi|$ and k. In the following we show how the problem can be encoded as a quantified Boolean formula of size polynomial in $|\varphi|$ and k.

Theorem 6.5. *For a specification given as an LTL formula φ, and bounds $k \in \mathbb{N}$ and $n \in \mathbb{N}$, there exists a quantified Boolean formula ϕ, such that, ϕ is satisfiable if and only if there is a transition system $T = (AP, I, O, S, s_0, \tau, L)$ of size n with $T \models_{k,I} \varphi$. The size of ϕ is in $O(|\varphi| + n^2 + k^2)$. The number of variables of ϕ is equal to $n \cdot (n \cdot 2^{|I|} + |O|) + k \cdot (|I| + 1) + n \cdot k(|O| + n + 1)$.*

Construction. We encode the bounded synthesis problem in the following quantified Boolean formula:

$$\exists\{\tau_{s,i,s'} \mid s, s' \in S, i \in 2^I\}.\ \exists\{o_s \mid s \in S, o \in O\}. \tag{6.1}$$

$$\forall\{i_j \mid i \in I, 0 \leq j < k\}.\ \forall\{l_j \mid 0 \leq j < k\}. \tag{6.2}$$

$$\forall\{o_j \mid o \in O, 0 \leq j < n \cdot k\}. \tag{6.3}$$

$$\forall\{s_j \mid s \in S, 0 \leq j < n \cdot k\}. \tag{6.4}$$

$$\forall\{l'_j \mid 0 \leq j < n \cdot k\}. \tag{6.5}$$

$$\varphi_{\text{det}} \wedge (\varphi_{\text{lasso}} \wedge \varphi_{\in T}^{n,k} \rightarrow [\![\varphi]\!]_0^{k,n \cdot k}) \tag{6.6}$$

which we read as: there is a transition system (1), such that, for all input sequences representable by lassos of length k (2) the corresponding sequence of outputs of the system (3) satisfies φ. The variables introduced in lines (4) and (5) are necessary to encode the corresponding output for the chosen input lasso.

An assignment to the variables satisfies the formula in line (6), if it represents a deterministic transition system (φ_{det}) in which lassos of length $n \cdot k$ ($\varphi_{\text{lasso} \wedge \varphi_{\in T}^{n,k}}$) satisfy the property φ ($[\![\varphi]\!]_0^{(k,n \cdot k)}$)). These constraints are defined as follows.

φ_{det}: A transition system is deterministic if for each state s and input i there is exactly one transition $\tau_{s,i,s'}$ to some state s': $\bigwedge_{s \in S} \bigwedge_{i \in 2^I} \bigvee_{s' \in S} (\tau_{s,i,s'} \wedge \bigwedge_{s' \neq s''} \overline{\tau_{s,i,s''}})$.

$\varphi^{n,k}_{\in T}$: for a certain input lasso of size k we can match a lasso in the system of size at most $n \cdot k$. A lasso of this size in the transition system matches the input lasso if the following constraints are satisfied.

$$\bigwedge_{0 \le j < n \cdot k} \bigwedge_{s \in S} (s_j \rightarrow \bigwedge_{o \in O} (o_j \leftrightarrow o_{s_j})) \tag{6.7}$$

$$\wedge\, s_{00} \tag{6.8}$$

$$\wedge \bigwedge_{0 \le j < n \cdot k-1} \bigwedge_{i \in 2^I} \bigwedge_{s,s' \in S} ((\bigwedge_{0 \le j' < k} l_{j'} \rightarrow i_{\Delta(j,k,j')}) \wedge s_j \rightarrow (\tau_{s,i,s'} \leftrightarrow s'_{j+1})) \tag{6.9}$$

$$\wedge \bigwedge_{i \in 2^I, s,s' \in S} ((\bigwedge_{0 \le j' < k} l_{j'} \rightarrow i_{\Delta(n \cdot k-1,k,j')}) \wedge s_{n \cdot k-1} \rightarrow (\tau_{s,i,s'} \leftrightarrow (\bigvee_{0 \le j < n \cdot k} l'_j \wedge s'_j))) \tag{6.10}$$

Lines (9) and (10) make sure that the chosen lasso follows the guessed transition relation τ. Line (10) handles the loop transition of the lasso, and makes sure that the loop of the lasso follows τ. Line (7) is a necessary requirement in order to match the output produced on the lasso with φ. If the output variables o_j satisfy the constraint $[\![\varphi]\!]^{(k,n \cdot k)}_0$, then the lasso satisfies φ. As the input lasso is smaller than its matching lasso in the system we need to make sure that the indices of the input variables are correct with respect to the chosen loop. This is computed using the function Δ which is given by:

$$\Delta(j,k,j') = \begin{cases} j & \text{if } j < k, \\ ((j-k) \mod (k-j')) + j' & \text{otherwise.} \end{cases}$$

φ_{lasso}: The formula encodes the additional constraint that exactly one of the loop variables can be true for a given variable valuation.

$[\![\varphi]\!]^{k,m}_0$: This constraint encodes the satisfaction of φ on lassos of size m. The encoding is similar to the encoding of bounded model checking [22] and the constraint we presented in the proof of Theorem 3.18, with the distinction of encoding the satisfaction relation of the atomic propositions, given below. As the inputs run with different indices than the outputs, we again, as in the lines (9) and (10), need to compute the correct indices using the function Δ.

	$h < m$	$h = m$
$[\![i]\!]^{k,m}_h$	$\bigwedge_{0 \le j' < k} (l_{j'} \rightarrow i_{\Delta(h,k,j')})$	$\bigvee_{j=0}^{m-1} (l'_j \wedge \bigwedge_{0 \le j' < k} (l_{j'} \rightarrow i_{\Delta(j,k,j')}))$
$[\![\neg i]\!]^{k,m}_h$	$\bigwedge_{0 \le j' < k} (l_{j'} \rightarrow \neg i_{\Delta(h,k,j')})$	$\bigvee_{j=0}^{m-1} (l'_j \wedge \bigwedge_{0 \le j' < k} (l_{j'} \rightarrow \neg i_{\Delta(j,k,j')}))$
$[\![o]\!]^{k,m}_h$	o_h	$\bigvee_{j=0}^{m-1} (l'_j \wedge o_j)$
$[\![\neg o]\!]^{k,m}_h$	$\neg o_h$	$\bigvee_{j=0}^{m-1} (l'_j \wedge \neg o_j)$

6.5 Synthesis of Approximate Implementations

In some cases, specifications remain unrealizable even when considered under bounded environments. Nevertheless, one might still be able to construct implementations that satisfy the specification in almost all input sequences of the environment. Consider for example the following simplified arbiter specification:

$$\square(\overline{w} \rightarrow \bigcirc \overline{g}) \wedge \square(r \rightarrow \diamondsuit g)$$

The specification defines an arbiter that should give grants g upon requests r, but is not allowed to provide these grants unless a signal w is true. The specification is unrealizable, because a sequence of inputs where the signal w is always false prevents the arbiter from answering any request. Bounding the environment does not help in this case as a lasso of size 1 already suffices to violate the specification (the one where w is always false). Nevertheless, one can still find reasonable implementations that satisfy the specification for a large fraction of input sequences. In particular, the fraction of input sequences where w remains false forever is less probable.

Definition 6.4 (ϵ-k-Approximation)**.** For a specification φ, a bound k, and an error rate ϵ, we say that a transition system T approximately satisfies φ with an error rate ϵ for lassos of length at most k, denoted by $T \models^{\epsilon}_{k,I} \varphi$, if and only if, $\frac{|\{\sigma | \sigma \in L^I_k(\mathit{Traces}(T))), \sigma \models \varphi\}|}{|L^I_k((2^I)^\omega)|} \geq 1 - \epsilon$. We call T an ϵ-k-approximation of φ.

Theorem 6.6. *For a specification given as a deterministic parity automaton P, a bound k and a error rate $0 \leq \epsilon \leq 1$, checking whether there is an implementation T, such that, $T \models^{\epsilon}_{k,I} P$ can be done in time polynomial in $|P|$ and exponential in k.*

Proof. For a given ϵ and k, we construct a nondeterministic parity tree automaton N that accepts all ϵ-k-approximations with respect to $L(P)$. For ϵ, we can compute the minimal number m of lassos from $L^I_k((2^I)^\omega)$ for which an ϵ-k-approximation has to satisfy the specification. In its initial state, the automaton N guesses m many lassos and accepts a transition system if it does not violate the specification on any of these lassos. The latter check is done by following the structure of the automaton constructed for P using Theorem 6.3. In order to check whether there is an ϵ-k-approximation for P, we solve the emptiness game of N. The size of N is $(2^k)^{m+1} \cdot |P|$.

□

Symbolic Approach

In the following, we present a symbolic approach for finding ϵ-k-approximations based on maximum model counting. We show that we can build a constraint system and apply a maximum model counting algorithm to compute a transition system that satisfies a specification for a maximum number of input sequences.

For a specification φ, bounds k and n on the length of the lassos and size of the system, respectively, we can compute an ϵ-k-approximation for φ by applying a maximum model counting algorithm to the constraint system given below. It encodes transition systems of size n that have an input lasso of length k that satisfies φ.

$$\exists\{\tau_{s,i,s'} \mid s, s' \in S, i \in 2^I\}.\ \exists\{o_t \mid s \in S, o \in O\}. \tag{6.11}$$

$$\exists\{i_j \mid i \in I, 0 \leq j < k\}.\ \exists\{l_j \mid 0 \leq j < k\}. \tag{6.12}$$

$$\exists\{x_j^i \mid x \in I, 0 \leq i, j < k\} \tag{6.13}$$

$$\exists\{o_j \mid o \in O, 0 \leq j < n \cdot k\}. \tag{6.14}$$

$$\exists\{s_j \mid s \in S, 0 \leq j < n \cdot k\}. \tag{6.15}$$

$$\exists\{l'_j \mid 0 \leq j < n \cdot k\}. \tag{6.16}$$

$$\varphi_{\text{det}} \wedge \varphi_{\text{lasso}} \wedge \varphi_{\in T}^{n,k} \wedge [\![\varphi]\!]_0^{k,n\cdot k} \wedge [\![k]\!]_0 \tag{6.17}$$

To check the existence of a ϵ-k-approximation, we maximize over the set of assignment to variables that define the transition system (line 11) and count over variables that define input sequences of the environment given by lassos of length k. As two input lassos of the same length may induce the same infinite input sequence, we count over auxiliary variables that represent unrollings of the lassos instead of counting over the input propositions themselves (line 13).

The formulas φ_{det}, φ_{lasso}, $\varphi_{\in T}^{n,k}$ and $[\![\varphi]\!]_0^{k,n\cdot k}$ are defined as in the previous section. The formula $[\![k]\!]_0$ is defined over that variables in line (13) and makes sure that input lasso that represent the same infinite sequence are not counted twice by unrolling the lasso to size $2k$. It is defined using the following recursive

rules:

$$\text{for } 0 \leq j < k-1 \quad [\![k]\!]_j = \bigwedge_{x \in I} (x_j^0 \leftrightarrow x_j) \wedge \tag{6.18}$$

$$\bigwedge_{0<i<k} (x_j^i \leftrightarrow x_{j+1}^{i-1}) \wedge \tag{6.19}$$

$$[\![k]\!]_{j+1} \tag{6.20}$$

$$[\![k]\!]_{k-1} = \bigwedge_{x \in I} (x_{k-1}^0 \leftrightarrow x_{k-1}) \wedge \tag{6.21}$$

$$\bigwedge_{0 \leq j < k} (l_j \rightarrow \bigwedge_{0<i<k} (x_{k-1}^i \leftrightarrow x_j^{i-1})) \tag{6.22}$$

Theorem 6.7. *For a specification given as an LTL formula φ, and bounds k and n, and an error rate ϵ, the propositional formula ϕ defined above is of size $O(|\varphi| + n^2 + k^2)$. The number of variables of ϕ is equal to $n \cdot (n \cdot 2^{|I|} + |O|) + k \cdot (k \cdot |I| + |I| + 1) + n \cdot k(|O| + n + 1)$.*

We implemented the symbolic encodings for the exact and approximate synthesis methods, and evaluated our approach on a bounded version of the greedy arbiter specification given at the beginning of the chapter, and another specification of a round-robin arbiter. The round-robin arbiter is defined by the specification:

$$\square\diamondsuit w \rightarrow \square\diamondsuit g_1 \wedge \square\diamondsuit g_2 \wedge \square(\neg w \rightarrow \bigcirc(\neg g_1 \wedge \neg g_2)) \wedge \square(\neg g_1 \vee \neg g_2)$$

This specification is realizable, with transition systems of size at least 4. We used our implementation to check whether we can find approximative solutions with smaller sizes. We used the tool CAQE [130] for solving the QBF instances and the tool MaxCount [73] for solving the approximate synthesis instances.

The results are presented in Table 6.1. As usual in synthesis, the size of the instances grows quickly as the size bound and number of processes increase. Inspecting the encoding constraints shows that the constraint for the specification is responsible for more than 80% of the number of gates in the encoding. The results show that, using the approach we proposed, we can synthesize implementations for unrealizable specifications by bounding the environment. The results for the approximate synthesis method further demonstrate that for the unrealizable cases one can still obtain approximative implementations that satisfy the specification on a large number of input sequences.

instance					QBF				MaxCount			
Spec.	Proc.	#States	Bound	Result	#Gates	∀	∃	time	#Max	#Count	rate	time
Round-Robin Arbiter	2	2	4	Unreal.	15556	48	12	9.91s	12	8	0.5	26s
	2	3	2	Unreal.	5338	40	24	2.45s	24	4	0.88	161s
	2	4	2	Real.	13414	60	12	12.15s	40	4	0.88	283s
Greedy Arbiter	1	2	2	Real.	1597	20	10	0.41s	10	4	1.0	0.79s
	1	2	3	Unreal.	4749	30	10	1.95s	10	6	0.88	3.86s
	1	3	3	Unreal.	16861	48	21	17.26s	21	6	0.88	20.83s
	1	4	3	Real.	43692	78	36	3m7.44s	36	6	1.0	2m43s
	1	4	4	-	169829	104	36	TO	36	8	-	TO
	2	4	2	Real.	24688	62	72	1m.24s	72	6	-	TO
	2	4	3	Unreal.	103433	93	72	27m15.2	72	12	-	TO
	3	2	2	Unreal.	3985	93	72	1.39s	38	8	0.65	4.18s

Table 6.1: Experimental results for the symbolic approaches. The rate in the approximate approach is the rate of input lassos on which the specification is satisfied.

6.6 Bibliographic Remarks

Providing good-quality environment specifications (typically in the form of assumptions on the allowed behaviors of the environment) is crucial for the synthesis of implementations from high-level specifications. Formal specifications, and thus also environment assumptions, are often hard to get right, and have been identified as one of the bottlenecks in formal methods and autonomy [133]. It is therefore not surprising, that there is a plethora of approaches addressing the problem of how to revise inadequate environment assumptions in the cases when these are the cause of unrealizability of the system requirements.

Most approaches in this direction build upon the idea of analyzing the cause of unrealizability of the specification and extracting assumptions that help eliminate this cause. The method proposed in [36] uses the game graph that is used to answer the realizability question in order to construct a Büchi automaton representing a minimal assumption that makes the specification realizable. The authors of [112, 113] provide an alternative approach where the environment assumptions are gradually strengthened based on counterstrategies for the environment. The key ingredient for this approach is using a library of specification templates and user scenarios for the mining of assumptions, in order to generate good-quality assumptions. A similar approach is used in [4], where, however, assumption patterns are synthesized directly from the counterstrategy without the need for the user to provide patterns. A different line of work focuses on giving feedback to the user or specification designer about the reason for unrealizability, so that they can, if possible, revise the specification accordingly. The key challenge addressed there lies in providing easy-to-understand feedback to users, which relies on finding a minimal cause for why the requirements are not achievable and generating a natural language explanation of this cause [131].

In the above mentioned approaches, assumptions are provided or constructed in the form of a temporal logic formula or an omega-automaton. Thus, it is on the one hand often difficult for specification designers to specify the right assumptions, and on the other hand special care has to be taken by the assumption generation procedures to ensure that the constructed assumptions are simple enough for the user to understand and evaluate. The work [101] takes a different route, by making assumptions about the *size* of the environment. That is, including as an additional parameter to the synthesis problem a bound on the state space of the environment. Similarly to temporal logic assumptions, this relaxation of the synthesis problem can render unrealizable specifications into realizable ones. From the system designer point of view, however, it might be significantly easier to estimate the size of environments that are feasible in practice than to express the implications of this additional information in a temporal logic formula. We take a similar route to [101], and consider a bound on the cyclic structures in the environment's behavior. Thus, the closest to our work is the temporal synthesis for bounded environments studied in [101]. In fact, we have shown that the synthesis problem for lasso-precise implementations and the synthesis problem under bounded environments can be reduced to each other. However, while the focus in [101] is on the computational complexity of the bounded synthesis problems, here we provide both automata-theoretic, as well as symbolic approaches for solving the synthesis problem for environments with bounded lassos. We further consider an *approximate version of this synthesis problem*. The benefits of using approximation are two-fold. Firstly, as shown in [101], while bounding the environment can make some specifications realizable, this comes at a high computational complexity price. In this case, approximation might be able to provide solutions of sufficient quality more efficiently. Furthermore, even after bounding the environment's input behaviors, the specification might still remain unrealizable, in which case we would like to satisfy the requirements for as many input lassos as possible. In that sense, we get closer to synthesis methods for probabilistic temporal properties in probabilistic environments [106]. However, we consider non-probabilistic environments (i.e., all possible inputs are equally likely), and provide probabilistic guarantees with desired confidence by employing maximum model counting techniques. Maximum model counting has previously been used for the synthesis of approximate non-reactive programs [73]. Here, on the other hand we are concerned with the synthesis of reactive systems from temporal specifications.

Bounding the size of the synthesized system implementation is a complementary restriction of the synthesis problem, which has attracted a lot of atten-

tion in recent years [67]. The computational complexity of the synthesis problem when both the system's and the environment's size is bounded has been studied in [101]. We provided a symbolic synthesis procedure for bounded synthesis of lasso-precise implementations based on quantified Boolean satisfiability.

In the nonreactive setting, propositional maximum model counting has been used for the synthesis of approximate programs [73]. Here, given a sketch of the program [3], the synthesis process searches for realizations of the program sketch such that the specification is satisfied on a maximum number of inputs.

Chapter 7

Discussion

In this thesis, we have introduced the model counting problem for linear-time properties, and presented algorithms for solving the model counting problem for different classes of properties, defined using different types of formalisms. We provided a thorough complexity analysis of the problem analyzing the time and space complexity of each algorithm, and established lower and upper complexity bounds in terms of counting complexity classes.

In addition to studying the algorithmic complexity of the model counting problem, we showed its connection to quantitative verification problems, explaining the relation of the problem to probabilistic model checking problems, quantitative information-flow control problems, and the synthesis of approximate implementations. For probabilistic model checking, we showed how model counting can be used to establish lower and upper bounds on the probability of a linear-time property being satisfied by a system. For quantitative information-flow control, we showed that using model counting we can introduce specialized efficient algorithm, for solving model checking problems for quantitative security policies, which significantly improve over the traditional model checking algorithms. Finally, in synthesis, we showed although a specification is unrealizable, using model counting we can synthesize approximate implementations for these specifications.

This thesis constitutes a significant step towards a general framework for quantitative verification problems based on model counting. In the next section

we show a first possible framework that captures the problems investigated in this thesis. We conclude our discussion and the thesis with an outlook on possible works that build the results of this thesis.

7.1 Relation to Model Measuring

In a more general view, we can see model checking of linear-time properties as a procedure of two operations that are consecutively applied to the system under scrutiny. Given a system and a trace property, the first operation assigns values 0 or 1 to the traces of the system, with respect to whether they violate or satisfy the property. The second operation is then applied on the results of the first operation by computing the minimum of all values and checking whether it is equal to 0. We can adapt this generalization for the computation of more involved quantitative measures. Instead of mapping traces of the system to 0 or 1, we can define a distance on traces and evaluate them against a specification according to this distance. Using an aggregation on the individual computed values, we can then determine a measure for the whole system, and compare the different systems by means of this measure.

The computation of the distance between a system and a specification is known as the *model measuring problem* [86, 88]. A model measuring problem is defined by a triple $(T, \delta_\varphi, \delta_{agg})$, where T is the system under scrutiny, $\delta_\varphi \colon \mathit{Traces}(T) \rightharpoonup \mathbb{R}$ is the *distance function*, a partial function that assigns traces of T values in $\mathbb{R}$, and $\delta_{agg} \colon \mathit{Multi}(\mathbb{R}) \to \mathbb{R}$, where $\mathit{Multi}(\mathbb{R})$ is the set of all multisets over $\mathbb{R}$, is the *measuring function* that computes an aggregate over all defined traces values computed by the distance function δ_φ. Solving a model measuring problem given by an instance $(T, \delta_\varphi, \delta_{agg})$ is then the problem of computing the value

$$\delta_{agg}([\delta_\varphi(t) \in \mathbb{R} \mid t \in \mathit{Traces}(T)])$$

where $[.]$ denotes a multiset over values in $\mathbb{R}$. Model checking a system T against a trace property φ is thus the model measuring problem with the distance function $\delta_\varphi(t) = (t \in \varphi)$, and the measuring function $\delta_{agg}(\Gamma) = \min(\Gamma)$ for some multi set Γ over $\{0, 1\}$.

The way we define model measuring instances above is characteristic to model measuring problems where the underlying model is a linear-time property, i.e., the distance function is defined over traces of the system. Many model measuring problems are nevertheless not captured by this formulation. Think of information-flow policies like deniability, where we want to compute the number of secret inputs that induce a single output. A distance function that only

measures single traces of the system is not enough for solving this measuring problem. What we need is a distance function that computes for each output the size of the largest set of traces where this output is produced, thus computing the number of secret inputs that produce this output. A distance function of this form is one that measures the distance of a set of traces of the system rather than the one for just a single trace, i.e., a distance function that generalizes hyperproperties .

To define model measuring instances over hyperproperties we need to generalize the distance function from a function that measures traces to a function that measure sets of traces. For an instance $(T, \delta_H, \delta_{agg})$ the model measuring problem becomes thus the problem of computing the value

$$\delta_{agg}([\delta_H(\Gamma) \in \mathbb{R} \mid \Gamma \subseteq \mathit{Traces}(T)])$$

for a transition system T, a distance function $\delta_H \colon 2^{\mathit{Traces}(T)} \rightharpoonup \mathbb{R}$, and a measuring function $\delta_{agg} \colon \mathit{Multi}(\mathbb{R}) \to \mathbb{R}$.

Using this definition of model measuring, we can now define the measuring problem of computing the minimal number of secret inputs that lead to the same output by the model measuring instance $(T, \delta_H, \delta_{agg})$ where

- $\delta_H(\Gamma) = \begin{cases} |\Gamma| & \forall t, t' \in \Gamma.\ t =_{\Sigma_O} t' \\ \text{undefined} & \text{otherwise} \end{cases}$
- $\delta_{agg}(\Gamma) = \min(\Gamma)$

Each set of traces Γ is mapped to its size if all the traces Γ share the same sequence of public output ($t =_{\Sigma_O} t'$), otherwise the set is not relevant and its distance value is undefined.

Based on this notion of model measuring, quantitative verification problems that can be solved using model counting can be formalized as model measuring instances where the distance function is defined over the size of sets of traces that satisfy a certain hyperproperty, i.e., model measuring problems of the form

$$\delta_H(\Gamma) = \begin{cases} \Delta(|\Gamma|) & \Gamma \in H \\ \text{undefined} & \text{otherwise} \end{cases}$$

for some function $\Delta : 2^{\mathit{Traces}(T)} \to \mathbb{R} \cup \{\infty\}$ and a hyperproperty H. This formalization captures all the model measuring problems addressed in this thesis. Whether there exists model-counting-based quantitative verification problems beyond this formalization is a question we leave for future work.

7.2 Future Work

7.2.1 Model counting Implementations

The model counting problem discussed in this thesis is based on a verification point of view, where we check the quality of the system with respect to a specification. We can also consider the model counting problem from the perspective of evaluating the specification itself. In many cases, specifications tend to underspecify a system. In this case, although a system satisfies the specification, it might still not fulfill the intent of the designer, as certain aspect on the functionality of the system have not been considered by the specification. Counting the number of implementations for a given specification, gives a measure on the quality of specification that determines the completeness of a specification. Specifications that are satisfied by a large number of implementation, tend to allow behaviors that where not intended by the designer of the system. A first approach for counting implementations is based on counting the number of finite trees that satisfy a formula [68].

7.2.2 Model counting for software verification

Debugging software is a tedious task and automated-tool support is necessary to quickly and effectively find bugs in a program. A possible way to increase the efficiency in debugging programs is one where we point out the fragments of the program that contain the majority of bugs. Using model counting, we might be able to mark the states of the program that lead to large number of error in the program.

7.2.3 Quantitative hyperlogics

As we have seen above, one possible formalization of model counting problems is one based on the notion of model measuring. Another possible way would be to check whether there is a logical formalization of counting problems for linear-time properties by introducing a logic that captures, for example, all the quantitative verification problems discussed in this thesis.

7.2.4 Optimizing model counting tools

Current state-of-the-art model counters cannot handle more than approximately 1000 to 10000 propositional variables [17, 135, 144]. Our propositional instances in our experiments where much larger and thus could not all be solved using the

current model counting tools in reasonable time and with reasonable memory. It is thus vital to look into optimizations specialized for counting problems of linear-time properties. For example, whether heuristics for smartly exploring the search space used in SAT-solvers can effectively be adapted for our model counting problems, or how techniques such symmetry breaking can be used to boost the time performance of model counters.

Bibliography

[1] Bowen Alpern and Fred B. Schneider. Defining liveness. *Inf. Process. Lett.*, 21(4):181–185, 1985.

[2] Bowen Alpern and Fred B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3):117–126, 1987.

[3] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 1–8, 2013.

[4] Rajeev Alur, Salar Moarref, and Ufuk Topcu. Counter-strategy guided refinement of GR(1) temporal logic specifications. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 26–33, 2013.

[5] Carme Àlvarez and Birgit Jenner. A very hard log-space counting class. *Theor. Comput. Sci.*, 107(1):3–30, 1993.

[6] Mário S. Alvim, Miguel E. Andrés, and Catuscia Palamidessi. Quantitative information flow in interactive systems. *Journal of Computer Security*, 20(1):3–50, 2012.

[7] Eugene Asarin, Michel Blockelet, Aldric Degorre, Catalin Dima, and Chunyan Mu. Asymptotic behaviour in temporal logic. In Henzinger and Miller [87], pages 10:1–10:9.

[8] Rehan Abdul Aziz, Geoffrey Chu, Christian Muise, and Peter Stuckey. #∃ sat: Projected model counting. In Marijn Heule and Sean Weaver, editors, *Theory and Applications of Satisfiability Testing – SAT 2015*, pages 121–137, Cham, 2015. Springer International Publishing.

[9] F. Bacchus, S. Dalmao, and T. Pitassi. Algorithms and complexity results for #sat and bayesian inference. In *44th Annual IEEE Symposium on Foundations of Computer Science, 2003. Proceedings.*, pages 340–351, Oct 2003.

[10] Michael Backes, Boris Kopf, and Andrey Rybalchenko. Automatic discovery and quantification of information leaks. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, SP '09, pages 141–153, Washington, DC, USA, 2009. IEEE Computer Society.

[11] Christel Baier, Lucia Cloth, Boudewijn R. Haverkort, Holger Hermanns, and Joost-Pieter Katoen. Performability assessment by model checking of markov reward models. *Formal Methods in System Design*, 36(1):1–36, 2010.

[12] Christel Baier and Clemens Dubslaff. From verification to synthesis under cost-utility constraints. *SIGLOG News*, 5(4):26–46, 2018.

[13] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008.

[14] Christel Baier, Joachim Klein, Sascha Klüppelholz, and Sascha Wunderlich. Weight monitoring with linear temporal logic: complexity and decidability. In Henzinger and Miller [87], pages 11:1–11:10.

[15] Anindya Banerjee and David A. Naumann. Stack-based access control and secure information flow. *J. Funct. Program.*, 15(2):131–177, 2005.

[16] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. Secure information flow by self-composition. *Mathematical Structures in Computer Science*, 21(6):1207–1252, 2011.

[17] Roberto J. Bayardo, Jr., and J. D. Pehoushek. Counting models using connected components. In *In AAAI*, pages 157–162, 2000.

[18] Richard Bellman. A markovian decision process. *Journal of Mathematics and Mechanics*, 6(5):679–684, 1957.

[19] Mordechai Ben-Ari, Amir Pnueli, and Zohar Manna. The temporal logic of branching time. *Acta Informatica*, 20(3):207–226, Sep 1983.

[20] Michael Benedikt, Rastislav Lenhardt, and James Worrell. LTL model checking of interval markov chains. In Nir Piterman and Scott A. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7795 of *Lecture Notes in Computer Science*, pages 32–46. Springer, 2013.

[21] Olivier Bernardi and Omer Giménez. A linear algorithm for the random sampling from regular languages. *Algorithmica*, 62(1-2):130–145, 2012.

[22] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in Computers*, 58:117–148, 2003.

[23] Vincent Bindschaedler, Reza Shokri, and Carl A. Gunter. Plausible deniability for privacy-preserving data synthesis. *PVLDB*, 10(5):481–492, 2017.

[24] Fabrizio Biondi, Axel Legay, Louis-Marie Traonouez, and Andrzej Wasowski. QUAIL: A quantitative security analyzer for imperative code. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, pages 702–707, 2013.

[25] Elazar Birnbaum and Eliezer L. Lozinskii. The good old davis-putnam procedure helps counting models. *J. Artif. Intell. Res.*, 10:457–477, 1999.

[26] Roderick Bloem, Krishnendu Chatterjee, Karin Greimel, Thomas A. Henzinger, Georg Hofferek, Barbara Jobstmann, Bettina Könighofer, and Robert Könighofer. Synthesizing robust systems. *Acta Inf.*, 51(3-4):193–220, 2014.

[27] Manuel Bodirsky, Tobias Gärtner, Timo von Oertzen, and Jan Schwinghammer. Effciently computing the density of regular languages. In *LATIN*, 2004.

[28] Benedikt Bollig, Normann Decker, and Martin Leucker. Frequency linear-time temporal logic. In *Sixth International Symposium on Theoretical Aspects of Software Engineering, TASE 2012, 4-6 July 2012, Beijing, China*, pages 85–92, 2012.

[29] J. Richakd Büchi. *Weak Second-Order Arithmetic and Finite Automata*, pages 398–424. Springer New York, New York, NY, 1990.

[30] Olivier Carton and Max Michel. Unambiguous büchi automata. *Theoretical Computer Science*, 297(1):37 – 81, 2003. Latin American Theoretical Informatics.

[31] Rohit Chadha, Umang Mathur, and Stefan Schwoon. Computing information flow using symbolic model-checking. In Raman and Suresh [132], pages 505–516.

[32] Anrin Chakraborti, Chen Chen, and Radu Sion. Datalair: Efficient block storage with plausible deniability against multi-snapshot adversaries. *PoPETs*, 2017(3):179, 2017.

[33] Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. A scalable approximate model counter. In *CP*, volume 8124 of *Lecture Notes in Computer Science*, pages 200–216. Springer, 2013.

[34] Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. Algorithmic improvements in approximate counting for probabilistic inference: From linear to logarithmic sat calls. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, IJCAI'16, pages 3569–3576. AAAI Press, 2016.

[35] Edward Chang, Zohar Manna, and Amir Pnueli. The safety-progress classification. In Friedrich L. Bauer, Wilfried Brauer, and Helmut Schwichtenberg, editors, *Logic and Algebra of Specification*, pages 143–202, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.

[36] Krishnendu Chatterjee, Thomas A. Henzinger, and Barbara Jobstmann. Environment assumptions for synthesis. In Franck van Breugel and Marsha Chechik, editors, *CONCUR 2008 - Concurrency Theory, 19th International Conference, CONCUR 2008, Toronto, Canada, August 19-22, 2008. Proceedings*, volume 5201 of *Lecture Notes in Computer Science*, pages 147–161. Springer, 2008.

[37] Han Chen and Pasquale Malacaria. Quantitative analysis of leakage for multi-threaded programs. In *Proceedings of the 2007 Workshop on Programming Languages and Analysis for Security, PLAS 2007, San Diego, California, USA, June 14, 2007*, pages 31–40, 2007.

[38] Noam Chomsky and George A. Miller. Finite state languages. *Information and Control*, 1(2):91–112, 1958.

[39] Tom Chothia, Yusuke Kawamoto, and Chris Novakovic. Leakwatch: Estimating information leakage from java programs. In *Computer Security - ESORICS 2014 - 19th European Symposium on Research in Computer Security, Wroclaw, Poland, September 7-11, 2014. Proceedings, Part II*, pages 219–236, 2014.

[40] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings*, volume 2404 of *Lecture Notes in Computer Science*, pages 359–364. Springer, 2002.

[41] David Clark, Sebastian Hunt, and Pasquale Malacaria. A static analysis for quantifying information flow in a simple imperative language. *Journal of Computer Security*, 15(3):321–371, 2007.

[42] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logics of Programs, Workshop, Yorktown Heights, New York, USA, May 1981*, pages 52–71, 1981.

[43] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In Dexter Kozen, editor, *Logics of Programs*, pages 52–71, Berlin, Heidelberg, 1982. Springer Berlin Heidelberg.

[44] Michael R. Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K. Micinski, Markus N. Rabe, and César Sánchez. Temporal logics for hyperproperties. In Martín Abadi and Steve Kremer, editors, *Principles of Security and Trust - Third International Conference, POST 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, volume 8414 of *Lecture Notes in Computer Science*, pages 265–284. Springer, 2014.

[45] Michael R. Clarkson, Andrew C. Myers, and Fred B. Schneider. Quantifying information flow with beliefs. *Journal of Computer Security*, 17(5):655–701, 2009.

[46] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *J. Comput. Secur.*, 18(6):1157–1210, September 2010.

[47] R. Clausius and Thomas Archer Hirst. *The mechanical theory of heat, with its applications to the steam-engine and to the physical properties of bodies. English translation*. J. Van Voorst, London, 1867.

[48] Ellis S Cohen. Information transmission in sequential programs. *Foundations of Secure Computation*, pages 297–335, 1978.

[49] Costas Courcoubetis and Mihalis Yannakakis. Verifying temporal properties of finite-state probabilistic programs. In *29th Annual Symposium on Foundations of Computer Science, White Plains, New York, USA, 24-26 October 1988*, pages 338–345. IEEE Computer Society, 1988.

[50] Jean-Michel Couvreur, Nasser Saheb, and Grégoire Sutre. An optimal automata approach to LTL model checking of probabilistic systems. In Moshe Y. Vardi and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 10th International Conference, LPAR 2003, Almaty, Kazakhstan, September 22-26, 2003, Proceedings*, volume 2850 of *Lecture Notes in Computer Science*, pages 361–375. Springer, 2003.

[51] Adnan Darwiche. Decomposable negation normal form. *J. ACM*, 48(4):608–647, 2001.

[52] Adnan Darwiche. New advances in compiling CNF into decomposable negation normal form. In Ramón López de Mántaras and Lorenza Saitta, editors, *Proceedings of the 16th Eureopean Conference on Artificial Intelligence, ECAI'2004, including Prestigious Applicants of Intelligent Systems, PAIS 2004, Valencia, Spain, August 22-27, 2004*, pages 328–332. IOS Press, 2004.

[53] Normann Decker, Peter Habermehl, Martin Leucker, Arnaud Sangnier, and Daniel Thoma. Model-checking counting temporal logics on flat structures. In Roland Meyer and Uwe Nestmann, editors, *28th International Conference on Concurrency Theory, CONCUR 2017, September 5-8, 2017, Berlin, Germany*, volume 85 of *LIPIcs*, pages 29:1–29:17. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017.

[54] Erik D. Demaine, Alejandro López-Ortiz, and J.Ian Munro. On universally easy classes for np-complete problems. *Theoretical Computer Science*, 2003.

[55] Dorothy E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, 1976.

[56] Dorothy E. Denning. *Cryptography and Data Security*. Addison-Wesley, 1982.

[57] Rayna Dimitrova, Bernd Finkbeiner, Máté Kovács, Markus N. Rabe, and Helmut Seidl. Model checking information flow in reactive systems. In *Verification, Model Checking, and Abstract Interpretation - 13th International Conference, VMCAI 2012, Philadelphia, PA, USA, January 22-24, 2012. Proceedings*, pages 169–185, 2012.

[58] Rayna Dimitrova, Bernd Finkbeiner, and Hazem Torfah. Synthesizing approximate implementations for unrealizable specifications. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*, volume 11561 of *Lecture Notes in Computer Science*, pages 241–258. Springer, 2019.

[59] Gerry Eisman and B. Ravikumar. Approximate recognition of non-regular languages by finite automata. In *Proceedings of the 28th Australasian Conference on Computer Science - Volume 38*, ACSC '05, Darlinghurst, Australia, 2005.

[60] E. Allen Emerson and Edmund M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In J. W. de Bakker and Jan van Leeuwen, editors, *Automata, Languages and Programming, 7th Colloquium, Noordweijkerhout, The Netherlands, July 14-18, 1980, Proceedings*, volume 85 of *Lecture Notes in Computer Science*, pages 169–181. Springer, 1980.

[61] E. Allen Emerson and Joseph Y. Halpern. "sometimes" and "not never" revisited: on branching versus linear time temporal logic. *J. ACM*, 33(1):151–178, 1986.

[62] Rachel Faran and Orna Kupferman. Spanning the spectrum from safety to liveness. In *ATVA 2015, Shanghai, China, Proceedings*.

[63] Bernd Finkbeiner. Synthesis of reactive systems. In Javier Esparza, Orna Grumberg, and Salomon Sickert, editors, *Dependable Software Systems Engineering*, volume 45 of *NATO Science for Peace and Security Series, D: Information and Communication Security*, pages 72–98. IOS Press, 2016.

[64] Bernd Finkbeiner, Lennart Haas, and Hazem Torfah. Canonical representations of k-safety hyperproperties. In *32nd IEEE Computer Security Foundations Symposium, CSF 2019, Hoboken, NJ, USA, June 25-28, 2019*, pages 17–31. IEEE, 2019.

[65] Bernd Finkbeiner, Christopher Hahn, and Hazem Torfah. Model checking quantitative hyperproperties. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*, volume 10981 of *Lecture Notes in Computer Science*, pages 144–163. Springer, 2018.

[66] Bernd Finkbeiner, Markus N. Rabe, and César Sánchez. Algorithms for model checking HyperLTL and HyperCTL. In Daniel Kroening and Corina S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, volume 9206 of *Lecture Notes in Computer Science*, pages 30–48. Springer, 2015.

[67] Bernd Finkbeiner and Sven Schewe. Bounded synthesis. *STTT*, 15(5-6):519–539, 2013.

[68] Bernd Finkbeiner and Hazem Torfah. Counting models of linear-time temporal logic. In Adrian-Horia Dediu, Carlos Martín-Vide, José Luis Sierra-Rodríguez, and Bianca Truthe, editors, *Language and Automata Theory and Applications - 8th International Conference, LATA 2014, Madrid, Spain, March 10-14, 2014. Proceedings*, volume 8370 of *Lecture Notes in Computer Science*, pages 360–371. Springer, 2014.

[69] Bernd Finkbeiner and Hazem Torfah. The density of linear-time properties. In Deepak D'Souza and K. Narayan Kumar, editors, *Automated Technology for Verification and Analysis - 15th International Symposium, ATVA 2017, Pune, India, October 3-6, 2017, Proceedings*, volume 10482 of *Lecture Notes in Computer Science*, pages 139–155. Springer, 2017.

[70] Philippe Flajolet. Analytic models and ambiguity of context-free languages. *Theoretical Computer Science*, 49(23):283 – 309, 1987.

[71] Vojtech Forejt, Marta Z. Kwiatkowska, Gethin Norman, and David Parker. Automated verification techniques for probabilistic systems. In *Formal Methods for Eternal Networked Software Systems - 11th International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM 2011, Bertinoro, Italy, June 13-18, 2011. Advanced Lectures*, pages 53–113, 2011.

[72] Martin Fränzle, James Kapinski, and Pavithra Prabhakar. Robustness in cyber-physical systems (dagstuhl seminar 16362). *Dagstuhl Reports*, 6(9):29–45, 2016.

[73] Daniel J. Fremont, Markus N. Rabe, and Sanjit A. Seshia. Maximum model counting. In *AAAI*, pages 3885–3892. AAAI Press, 2017.

[74] Vibhav Gogate and Rina Dechter. Approximate counting by sampling the backtrack-free search space. In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence, July 22-26, 2007, Vancouver, British Columbia, Canada*, pages 198–203, 2007.

[75] J. A. Goguen and J. Meseguer. Security policies and security models. In *1982 IEEE Symposium on Security and Privacy*, pages 11–11, April 1982.

[76] Carla P. Gomes, Jörg Hoffmann, Ashish Sabharwal, and Bart Selman. From sampling to model counting. In *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, pages 2293–2299, 2007.

[77] Carla P. Gomes, Ashish Sabharwal, and Bart Selman. Model counting. In *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 633–654. IOS Press, 2009.

[78] Erich Grädel, Wolfgang Thomas, and Thomas Wilke, editors. *Automata, Logics, and Infinite Games: A Guide to Current Research*. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[79] James W. Gray, III. Toward a mathematical foundation for information flow security. In *Proc. IEEE Symposium on Security and Privacy*, pages 210–34, May 1991.

[80] Christian Hammer and Gregor Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *Int. J. Inf. Sec.*, 8(6):399–422, 2009.

[81] D. Harel and A. Pnueli. On the development of reactive systems. In Krzysztof R. Apt, editor, *Logics and Models of Concurrent Systems*, pages 477–498, Berlin, Heidelberg, 1985. Springer Berlin Heidelberg.

[82] Sergiu Hart, Micha Sharir, and Amir Pnueli. Termination of probabilistic concurrent program. *ACM Trans. Program. Lang. Syst.*, 5(3):356–380, 1983.

[83] Michael Hartwig. On the density of regular and context-free languages. In *Computing and Combinatorics*. Springer Berlin Heidelberg, 2010.

[84] Lane A. Hemaspaandra and Heribert Vollmer. The satanic notations: counting classes beyond #p and other definitional adventures. *SIGACT News*, 26(1):2–13, 1995.

[85] Thomas A. Henzinger. Quantitative fitness measures for embedded systems. In *PECCS 2013 - Proceedings of the 3rd International Conference on Pervasive Embedded Computing and Communication Systems, Barcelona, Spain, 19-21 February, 2013*, 2013.

[86] Thomas A. Henzinger. Quantitative reactive modeling and verification. *Computer Science - R&D*, 28(4):331–344, 2013.

[87] Thomas A. Henzinger and Dale Miller, editors. *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014*. ACM, 2014.

[88] Thomas A. Henzinger and Jan Otop. From model checking to model measuring. In Pedro R. D'Argenio and Hernán Melgratti, editors, *CONCUR 2013 – Concurrency Theory*, pages 273–287, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[89] Gerard J. Holzmann. The model checker spin. *IEEE Trans. Softw. Eng.*, 23(5):279–295, May 1997.

[90] Michael Huth and Marta Z. Kwiatkowska. Quantitative analysis and model checking. In *Proceedings, 12th Annual IEEE Symposium on Logic in Computer Science, Warsaw, Poland, June 29 - July 2, 1997*, pages 111–122. IEEE Computer Society, 1997.

[91] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning About Systems*. Cambridge University Press, New York, NY, USA, 2004.

[92] Simon Jantsch, David Müller, Christel Baier, and Joachim Klein. From LTL to unambiguous büchi automata via disambiguation of alternating automata. In *Formal Methods - The Next 30 Years - Third World Congress, FM 2019, Porto, Portugal, October 7-11, 2019, Proceedings*, pages 262–279, 2019.

[93] Sampath Kannan, Z. Sweedyk, and Stephen R. Mahaney. Counting and random generation of strings in regular languages. In Kenneth L. Clarkson, editor, *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, 22-24 January 1995. San Francisco, California, USA.*, pages 551–557. ACM/SIAM, 1995.

[94] Joost-Pieter Katoen. The probabilistic model checking landscape. In Martin Grohe, Eric Koskinen, and Natarajan Shankar, editors, *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, July 5-8, 2016*, pages 31–45. ACM, 2016.

[95] John G Kemeny, James Laurie Snell, et al. *Finite markov chains*, volume 356. van Nostrand Princeton, NJ, 1960.

[96] Boris Köpf and David A. Basin. An information-theoretic model for adaptive side-channel attacks. In *Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007, Alexandria, Virginia, USA, October 28-31, 2007*, pages 286–296, 2007.

[97] Boris Köpf and David A. Basin. Automatically deriving information-theoretic bounds for adaptive side-channel attacks. *Journal of Computer Security*, 19(1):1–31, 2011.

[98] Boris Köpf and Andrey Rybalchenko. Approximation and randomization for quantitative information-flow analysis. In *Proceedings of the 23rd IEEE Computer Security Foundations Symposium, CSF 2010, Edinburgh, United Kingdom, July 17-19, 2010*, pages 3–14, 2010.

[99] Boris Köpf and Andrey Rybalchenko. *Automation of Quantitative Information-Flow Analysis*, pages 1–28. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[100] Hadas Kress-Gazit and Hazem Torfah. The challenges in specifying and explaining synthesized implementations of reactive systems. In Bernd Finkbeiner and Samantha Kleinberg, editors, *Proceedings 3rd Workshop on formal reasoning about Causation, Responsibility, and Explanations in Science and Technology, CREST@ETAPS 2018, Thessaloniki, Greece, 21st April 2018.*, volume 286 of *EPTCS*, pages 50–64, 2018.

[101] Orna Kupferman, Yoad Lustig, Moshe Y. Vardi, and Mihalis Yannakakis. Temporal synthesis for bounded systems and environments. In Thomas Schwentick and Christoph Dürr, editors, *28th International Symposium on*

Theoretical Aspects of Computer Science, STACS 2011, March 10-12, 2011, Dortmund, Germany, volume 9 of *LIPIcs*, pages 615–626. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2011.

[102] Orna Kupferman and Moshe Y. Vardi. Weak alternating automata and tree automata emptiness. In *Proceedings of the Thirtieth Annual ACM Symposium on the Theory of Computing, Dallas, Texas, USA, May 23-26, 1998*, pages 224–233, 1998.

[103] Andrey Kupriyanov and Bernd Finkbeiner. Causal termination of multi-threaded programs. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 814–830. Springer, 2014.

[104] Marta Kwiatkowska, Gethin Norman, and David Parker. Probabilistic model checking: advances and applications. In Rolf Drechsler, editor, *Formal System Verification: State-of the-Art and Future Trends*, pages 73–121. Springer Verlag, 2017.

[105] Marta Z. Kwiatkowska. Quantitative verification: models, techniques and tools. In Ivica Crnkovic and Antonia Bertolino, editors, *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, September 3-7, 2007, Companion Papers*, pages 449–458. ACM, 2007.

[106] Marta Z. Kwiatkowska and David Parker. Automated verification and strategy synthesis for probabilistic systems. In Dang Van Hung and Mizuhito Ogawa, editors, *Automated Technology for Verification and Analysis - 11th International Symposium, ATVA 2013, Hanoi, Vietnam, October 15-18, 2013. Proceedings*, volume 8172 of *Lecture Notes in Computer Science*, pages 5–22. Springer, 2013.

[107] Jean-Marie Lagniez and Pierre Marquis. A recursive algorithm for projected model counting. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019.*, pages 1536–1543, 2019.

[108] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Softw. Eng.*, 3(2):125–143, March 1977.

[109] L. H. Landweber. Decision problems forω-automata. *Mathematical systems theory*, 3(4):376–384, Dec 1969.

[110] François Laroussinie, Antoine Meyer, and Eudes Petonnet. Counting CTL. *Logical Methods in Computer Science*, 9(1), 2012.

[111] Thomas Lengauer and Klaus W. Wagner. The correlation between the complexities of the nonhierarchical and hierarchical versions of graph problems. *J. Comput. Syst. Sci.*, 44(1):63–93, 1992.

[112] Wenchao Li, Lili Dworkin, and Sanjit A. Seshia. Mining assumptions for synthesis. In Satnam Singh, Barbara Jobstmann, Michael Kishinevsky, and Jens Brandt, editors, *9th IEEE/ACM International Conference on Formal Methods and Models for Codesign, MEMOCODE 2011, Cambridge, UK, 11-13 July, 2011*, pages 43–50. IEEE, 2011.

[113] Wenchao Li, Dorsa Sadigh, S. Shankar Sastry, and Sanjit A. Seshia. Synthesis for human-in-the-loop control systems. In *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, pages 470–484, 2014.

[114] Peter A. Lindsay. On alternating -automata. *Journal of Computer and System Sciences*, 36(1):16 – 24, 1988.

[115] Pasquale Malacaria. Assessing security threats of looping constructs. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, pages 225–235, 2007.

[116] Kuldeep Meel. *Constrained Counting and Sampling: Bridging the Gap between Theory and Practice*. PhD thesis, Rice University, 2017.

[117] Dimiter Milushev and Dave Clarke. Incremental hyperproperty model checking via games. In *Secure IT Systems - 18th Nordic Conference, NordSec 2013, Ilulissat, Greenland, October 18-21, 2013, Proceedings*, pages 247–262, 2013.

[118] Satoru Miyano and Takeshi Hayashi. Alternating finite automata on omega-words. *Theor. Comput. Sci.*, 32:321–330, 1984.

[119] Daniel Morwood and Daniel Bryce. Evaluating temporal plans in incomplete domains. In *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence*, AAAI'12, pages 1793–1801. AAAI Press, 2012.

[120] D. E. Muller, A. Saoudi, and P. E. Schupp. Weak alternating automata give a simple explanation of why most temporal and dynamic logics are decidable in exponential time. In *[1988] Proceedings. Third Annual Symposium on Logic in Computer Science*, pages 422–427, July 1988.

[121] David E. Muller. Infinite sequences and finite machines. In *Proceedings of the 1963 Proceedings of the Fourth Annual Symposium on Switching Circuit Theory and Logical Design*, SWCT '63, pages 3–16, Washington, DC, USA, 1963. IEEE Computer Society.

[122] Andrew C. Myers. Jflow: Practical mostly-static information flow control. In *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*, pages 228–241, 1999.

[123] Luan Viet Nguyen, James Kapinski, Xiaoqing Jin, Jyotirmoy V. Deshmukh, and Taylor T. Johnson. Hyperproperties of real-valued signals. In Jean-Pierre Talpin, Patricia Derler, and Klaus Schneider, editors, *Proceedings of the 15th ACM-IEEE International Conference on Formal Methods and Models for System Design, MEMOCODE 2017, Vienna, Austria, September 29 - October 02, 2017*, pages 104–113. ACM, 2017.

[124] Christos H. Papadimitriou. *Computational complexity*. Addison-Wesley, 1994.

[125] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 179–190, New York, NY, USA, 1989. ACM.

[126] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.

[127] Amir Pnueli and Lenore D. Zuck. Probabilistic verification by tableaux. In *Proceedings of the Symposium on Logic in Computer Science (LICS '86), Cambridge, Massachusetts, USA, June 16-18, 1986*, pages 322–331, 1986.

[128] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, UK, 1982. Springer-Verlag.

[129] Markus N. Rabe. *A temporal logic approach to iInformation-flow control*. PhD thesis, Saarland University, 2016.

[130] Markus N. Rabe and Leander Tentrup. Caqe: A certifying qbf solver. In *Proceedings of the 15th Conference on Formal Methods in Computer-aided Design (FMCAD'15)*, pages 136–143, September 2015.

[131] Vasumathi Raman, Constantine Lignos, Cameron Finucane, Kenton C. T. Lee, Mitchell P. Marcus, and Hadas Kress-Gazit. Sorry dave, i'm afraid I can't do that: Explaining unachievable robot tasks using natural language. In Paul Newman, Dieter Fox, and David Hsu, editors, *Robotics: Science and Systems IX, Technische Universität Berlin, Berlin, Germany, June 24 - June 28, 2013*, 2013.

[132] Venkatesh Raman and S. P. Suresh, editors. *34th International Conference on Foundation of Software Technology and Theoretical Computer Science, FSTTCS 2014, December 15-17, 2014, New Delhi, India*, volume 29 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2014.

[133] Kristin Yvonne Rozier. Specification: The biggest bottleneck in formal methods and autonomy. In *Verified Software. Theories, Tools, and Experiments - 8th International Conference, VSTTE 2016, Toronto, ON, Canada, July 17-18, 2016, Revised Selected Papers*, pages 8–26, 2016.

[134] S. Safra. On the complexity of omega -automata. In *Proceedings of the 29th Annual Symposium on Foundations of Computer Science*, SFCS '88, pages 319–327, Washington, DC, USA, 1988. IEEE Computer Society.

[135] Tian Sang, Fahiem Bacchus, Paul Beame, Henry A. Kautz, and Toniann Pitassi. Combining component caching and clause learning for effective model counting. In *Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing*, 2004.

[136] Sven Schewe and Thomas Varghese. Determinising parity automata. In Erzsébet Csuhaj-Varjú, Martin Dietzfelbinger, and Zoltán Ésik, editors, *Mathematical Foundations of Computer Science 2014*, pages 486–498, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[137] Claude E. Shannon. A mathematical theory of communication. *Mobile Computing and Communications Review*, 5(1):3–55, 2001.

[138] A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. *J. ACM*, 32(3):733–749, July 1985.

[139] Geoffrey Smith. On the foundations of quantitative information flow. In Luca de Alfaro, editor, *Foundations of Software Science and Computational Structures*, pages 288–302, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[140] Andrew Szilard, Sheng Yu, Kaizhong Zhang, and Jeffrey Shallit. Characterizing regular languages with polynomial densities. In *Mathematical Foundations of Computer Science 1992*. Springer Berlin Heidelberg, 1992.

[141] Paulo Tabuada, Ayca Balkan, Sina Y. Caliskan, Yasser Shoukry, and Rupak Majumdar. Input-output robustness for discrete systems. In Ahmed Jerraya, Luca P. Carloni, Florence Maraninchi, and John Regehr, editors, *Proceedings of the 12th International Conference on Embedded Software, EMSOFT 2012, part of the Eighth Embedded Systems Week, ESWeek 2012, Tampere, Finland, October 7-12, 2012*, pages 217–226. ACM, 2012.

[142] Paulo Tabuada, Sina Yamac Caliskan, Matthias Rungger, and Rupak Majumdar. Towards robustness for cyber-physical systems. *IEEE Trans. Automat. Contr.*, 59(12):3151–3163, 2014.

[143] Wolfgang Thomas. Handbook of theoretical computer science (vol. b). chapter Automata on Infinite Objects, pages 133–191. MIT Press, Cambridge, MA, USA, 1990.

[144] Marc Thurley. sharpSAT: Counting models with advanced component caching and implicit bcp. In *Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing*, SAT'06, pages 424–429, Berlin, Heidelberg, 2006. Springer-Verlag.

[145] Hazem Torfah and Martin Zimmermann. The complexity of counting models of linear-time temporal logic. In Raman and Suresh [132], pages 241–252.

[146] Hazem Torfah and Martin Zimmermann. The complexity of counting models of linear-time temporal logic. *Acta Informatica*, 55(3):191–212, 2018.

[147] Leslie G. Valiant. The complexity of computing the permanent. *Theor. Comput. Sci.*, 8:189–201, 1979.

[148] Leslie G. Valiant. The complexity of enumeration and reliability problems. *SIAM J. Comput.*, 8(3):410–421, 1979.

[149] Moshe Y. Vardi. Automatic verification of probabilistic concurrent finite-state programs. In *26th Annual Symposium on Foundations of Computer Science, Portland, Oregon, USA, 21-23 October 1985*, pages 327–338. IEEE Computer Society, 1985.

[150] Moshe Y. Vardi. *Alternating automata and program verification*, pages 471–485. Springer Berlin Heidelberg, Berlin, Heidelberg, 1995.

[151] Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification (preliminary report). In *Proceedings of the Symposium on Logic in Computer Science (LICS '86), Cambridge, Massachusetts, USA, June 16-18, 1986*, pages 332–344, 1986.

[152] M.Y. Vardi and P. Wolper. Reasoning about infinite computations. *Inf. Comput.*, 115(1):1–37, November 1994.

[153] Wei Wei and Bart Selman. A new approach to model counting. In Fahiem Bacchus and Toby Walsh, editors, *Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005, St. Andrews, UK, June 19-23, 2005, Proceedings*, volume 3569 of *Lecture Notes in Computer Science*, pages 324–339. Springer, 2005.

[154] H. Yasuoka and T. Terauchi. On bounding problems of quantitative information flow. In *Proc. European Symposium on Research in Computer Security*, pages 357–372, September 2010.

[155] Hirotoshi Yasuoka and Tachio Terauchi. Quantitative information flow as safety and liveness hyperproperties. *Theor. Comput. Sci.*, 538:167–182, 2014.

List of Figures

List of Tables