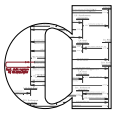


On the Connection of Probabilistic Model Checking, Planning, and Learning for System Verification

A dissertation submitted towards the degree Doctor of Engineering (Dr.-Ing.) of the Faculty of Mathematics and Computer Science of Saarland University

submitted by

Michaela Klauck

Saarbrücken

2022

Dean of the Faculty	Prof. Dr. rer. nat. Jürgen Steimle
Date of the Colloquium	28.06.2022
Chair of the Committee	Prof. Dr. rer. nat. Verena Wolf
Commission	Prof. Dr.-Ing. Holger Hermanns
	Prof. Dr. rer. nat. Jörg Hoffmann
	Prof. Dr. Benoît Delahaye
Academic Assistant	Dr. rer. nat. Martin Bromberger

Abstract

This thesis presents approaches using techniques from the model checking, planning, and learning community to make systems more reliable and perspicuous. First, two heuristic search and dynamic programming algorithms are adapted to be able to check extremal reachability probabilities, expected accumulated rewards, and their bounded versions, on general Markov decision processes (MDPs). Thereby, the problem space originally solvable by these algorithms is enlarged considerably. Correctness and optimality proofs for the adapted algorithms are given, and in a comprehensive case study on established benchmarks it is shown that the implementation, called Modysh, is competitive with state-of-the-art model checkers and even outperforms them on very large state spaces. Second, Deep Statistical Model Checking (DSMC) is introduced, usable for quality assessment and learning pipeline analysis of systems incorporating trained decision-making agents, like neural networks (NNs). The idea of DSMC is to use statistical model checking to assess NNs resolving nondeterminism in systems modeled as MDPs. The versatility of DSMC is exemplified in a number of case studies on Racetrack, an MDP benchmark designed for this purpose, flexibly modeling the autonomous driving challenge. In a comprehensive scalability study it is demonstrated that DSMC is a lightweight technique tackling the complexity of NN analysis in combination with the state space explosion problem.

Zusammenfassung

Diese Arbeit präsentiert Ansätze, die Techniken aus dem Model Checking, Planning und Learning Bereich verwenden, um Systeme verlässlicher und klarer verständlich zu machen. Zuerst werden zwei Algorithmen für heuristische Suche und dynamisches Programmieren angepasst, um Extremwerte für Erreichbarkeitswahrscheinlichkeiten, Erwartungswerte für Kosten und beschränkte Varianten davon, auf generellen Markov Entscheidungsprozessen (MDPs) zu untersuchen. Damit wird der Problemraum, der ursprünglich mit diesen Algorithmen gelöst wurde, deutlich erweitert. Korrektheits- und Optimalitätsbeweise für die angepassten Algorithmen werden gegeben und in einer umfassenden Fallstudie wird gezeigt, dass die Implementierung, namens Modysh, konkurrenzfähig mit den modernsten Model Checkern ist und deren Leistung auf sehr großen Zustandsräumen sogar übertrifft. Als Zweites wird Deep Statistical Model Checking (DSMC) für die Qualitätsbewertung und Lernanalyse von Systemen mit integrierten trainierten Entscheidungsgenten, wie z.B. neuronalen Netzen (NN), eingeführt. Die Idee von DSMC ist es, statistisches Model Checking zur Bewertung von NNs zu nutzen, die Nichtdeterminismus in Systemen, die als MDPs modelliert sind, auflösen. Die Vielseitigkeit des Ansatzes wird in mehreren Fallbeispielen auf Racetrack gezeigt, einer MDP Benchmark, die zu diesem Zweck entwickelt wurde und die Herausforderung des autonomen Fahrens flexibel modelliert. In einer umfassenden Skalierbarkeitsstudie wird demonstriert, dass DSMC eine leichtgewichtige Technik ist, die die Komplexität der NN-Analyse in Kombination mit dem State Space Explosion Problem bewältigt.

Acknowledgements

This work was partially supported by the German Research Foundation (DFG) under grant No. 389792660, as part of TRR 248 – CPEC (Center for Perspicuous Computing), see https://perspicuous-computing.science, and by the ERC Advanced Investigators Grant 695614 (POWVER),

Some Personal Words

My endeavor of pursuing a PhD at the Dependable Systems and Software chair started already in August 2017 before even signing a contract. After several tutor positions and a lot of fun and work in the mathematics preparatory course team, I already knew Felix, Gereon, and Sebastian who promised me that working here would be a lot of fun.

To get to know also the others in the team, Holger offered me the opportunity to take part in the chair's retreat in Tanna, a very small city in Thüringen. Nobody expected that this trip would turn into a literal crash landing for me. But after surviving this first shock, returning home without getting lost at a toilet on a service station, and knowing that Vahid has a big heart for goats, I started officially in November and shared my office with Gilles and Yuliya.

Dear Gilles, I have to confess, that at first glance I was misled like the grandma who changed to the other side of the street but very soon we turned into office mates who know the habits of each other very well. We had quite some philosophical and deep discussions about our work and doing a PhD in general. All this helped me so much in overcoming some doubts and I learned a lot from you. Thank you very much!

Soon we established Game and Movie Nights in our new leisure room with the very extravagant orange zebra sofa which is exactly 1 m(ichaela) long.

At CONFESTA 2018 I got the chance to go with Daniel, Gereon, and Sebastian on the greatest trip ever. We visited Beijing for the conference and the final CAP workshop. We made it without getting hacked, or thrown into prison, but with hurting legs from all the sightseeing, climbing up the wall, and especially from the soccer match Gereon, Rob van Glabbeek, Uwe Nestmann, and me won against Kim Larsen.

Especially in Christmas time, the Depend Band, often in different line-ups with Felix, Gereon, Maxi, and me, filled the whole building with music. I always enjoyed it even though there were some wrong notes. Sorry for those who had to listen.

I really enjoyed our trips to Phantasia Land and Europapark but I have to say that doing Hula Hoop on the street was more fun than riding Silverstar next to Holger. Thanks for noticing, Florian.

Furthermore, I am proud to be able to maintain that I am the first and only one who ever rode the unicycle inside university and especially inside the lecture hall during a concurrent programming mini-test. This fact gets even more amusing when remembering my first adventure with the chair where I convincingly demonstrated that I am not able to ride an e-bike.

Organizing the SFB review for CPEC showed that if Felix and me should get tired of doing science at some point, we can directly start an event management service with Christa as our secretary.

That also remembers me of Christa's 60th birthday where Felix and me played hide and seek with her in E1 1 and E1 3 to hide her present. Of course, we did not expect Erich Reindel to cross our way next to the elevator. But then there was no way out except for drawing him into the game without him noticing it.

Apart from these highlights there were more small events like the business runs, and other sport sessions at the chair. Thanks Florian for being my training partner. We also enjoyed a lot of Al Bacio and billiard evenings on Wednesdays. Another highlight we will never forget are the delicious Premium Dinners Maxi served us.

One of the most formative events for me was the summer school in Marktoberdorf in 2019. I learned so much about other areas of research in our field and met so many other PhD students of our community all sharing the same way. That created such an exciting, motivating, and unique atmosphere, which I never experienced afterwards. It was such a pleasure to feel the team spirit and openness of so many young researchers from all over the world who did not know each other before. We trained together even in heavy rain for the final traditional soccer match, climbed up Hahnenkamm in Austria in the monsoon, and had a nice trip to Neuschwanstein and Hohenschwangau Castle. But the most impressive experience for me was the final evening. We were asked to organize an entertainment program. After thinking twice, I had the courage to start a choir, in the hope that two or three others who are better singers than me, would help me perform a song while I play the guitar. In the end, we were a choir of 12 singers, a trombone, and a guitar. I will never forget our performance of Lemon Tree, Yesterday, and Mein kleiner grüner Kaktus. I did not believe before that forming a choir of people from all over the world and practicing twice can bring so much joy

to all MOD participants. Needlessly to say that the evening continued with more than four hours of playing the guitar and singing with more than 20 people until my fingers got numb.

Unfortunately, this was one of the best but also the last trip of my road to a PhD because on Wednesday, March 11, 2020 we all had our last normal working day together in the office. Since then, social events, conferences, and other leisure activities got very rare ... I am sure more anecdotes would have followed, and for sure will follow at the craziest chair at university, soon.

Before coming to a long list of names, I have one request to all of you: Even though I am used to being called Michael(a)-Manuel(a)-Claudia-Mitchella by many mail senders, please do not forget my name and keep me and our adventures in good memories.

During my time at the chair I met so many people who influenced my work and gave inspiration to me that I can probably not remember all of them anymore but I want to thank as many as possible (mostly in alphabetical order): Kevin Baum, Yuliya Butkova, Pedro D'Argenio, Juan Fraire, Vahid Hashemi, Kathrin Stark, and Sarah Sterz for the time together at and around the chair.

Special thanks go to Sebastian Biewer for giving me insights into car emissions and the dCar project, Felix Freiberger for the endless technical support, and funny game and billiard nights, Gereon Fox for deep discussions about the right way to do a PhD, for billiard, and dancing events, Alexander Graf-Brill for funny discussions in Saarland-dialect, telling me about his scientific experiences, and always believing in me, Timo P. Gros for political discussions, and of course for working together in a lot of projects which would not have been realizable without his expertise in NN learning, Arnd Hartmanns for the insights in the Modest Toolset, Ute Hornung for nice conversations on the corridor, Nikolai Käfer for being my neighbor in the Programming 1 tutorial, and for afterwards attending many lectures together, for having a lot of fun in the mathematics preparatory course, and at cultural events, Maximilian Köhl for the legendary Premium Dinners, for rescuing me when weird junkies show up in my office before 8 am, for pushing the red emergency button in my office just for scientific curiosity, and of course also for deep scientific debates on many papers, the opportunity to work on many projects together, and the opportunity to work on and with Momba, Sabine Nermerich for taking care about the bureaucracy, Gilles Nies for being my office mate, Christa Schäfer for always taking care of everything and everyone at the chair, Florian Schießl for the hardware support and his technical handicrafts, Daniel Stan for the help in doing proofs, and for nice skate and billiard events, Marcel Steinmetz for the introduction into the planning world, and Gregory Stock for several years of hard work together in the team of the mathematics preparatory course.

In addition, I want to thank Jörg Hoffmann who impressively improved all our papers by rewriting them, developed great new ideas, and always took care to have a next paper target. I learned so much from you and working together with you always felt very productive.

Of course, a huge thank you goes to Holger Hermanns for supervising me during the hard times of my road to a PhD, for giving me the opportunity to work at his chair in such a special, unique, funny, and sometimes also crazy group, but especially for teaching me how to do research, write papers, and for the brainstorming sessions on what to do next. You are the one who made it possible for me to do a PhD and stand where I am now. I enjoyed doing science with you and working on projects together with you as much as I enjoyed the relaxed atmosphere you created at social events at the chair or at conferences. For all of this I am very grateful.

I also would like to thank the reviewers of my thesis, Benoît Delahaye, Holger Hermanns, and Jörg Hoffmann.

The biggest thanks go to my family, first of all to my mother, my father, and my sister who always believed in me, and supported me in pursuing my special way even in the hardest and most difficult times of our lives. You gave me the opportunity to be where I am today by giving me the chance to be the first in our family who studies, and teaching me how important education is. I also want to thank my grandma and my grandpa for their interest in what I am doing. Finally, the one who guided and influenced my way the most is my uncle. He is the one who brought me to computer science. And: Who knows where I would stand now without him and many hours of discussions about my studies in the first semesters?

To all of you: I know that your help and support in whichever form, may it be topic related discussions at the chair, may it be that you believed in me when I did not anymore, or may it be the fun we had in our free time, is not a matter of course, and I hope that I could and can give back something to you.

Thank you all!

Table of Contents

List of Figures

List of Publications

This is the complete list of publications of the author written during their time as a PhD student, as requested in the Doctoral Degree Regulations of the Faculty of Mathematics and Computer Science at Saarland University. The publications are listed chronologically. Authors of publications in the model checking community are ordered alphabetically (with two exceptions ordered by contribution). In the planning and visualization community authors are listed by contribution.

Conference Papers

- Michaela Klauck, Marcel Steinmetz, Jörg Hoffmann, and Holger Hermanns. Compiling Probabilistic Model Checking into Probabilistic Planning. In Mathijs de Weerdt, Sven Koenig, Gabriele Röger, and Matthijs T. J. Spaan, editors, *Proceedings of the Twenty-Eighth International Conference on Automated Planning and Scheduling, ICAPS 2018, Delft, The Netherlands, June 24-29, 2018*, pages 150–154. AAAI Press, 2018

- Arnd Hartmanns, Michaela Klauck, David Parker, Tim Quatmann, and Enno Ruijters. The Quantitative Verification Benchmark Set. In Tomás Vojnar and Lijun Zhang, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part I*, volume 11427 of *Lecture Notes in Computer Science*, pages 344–350. Springer, 2019

- Ernst Moritz Hahn, Arnd Hartmanns, Christian Hensel, Michaela Klauck, Joachim Klein, Jan Křetínský, David Parker, Tim Quatmann, Enno Ruijters, and Marcel Steinmetz. The 2019 Comparison of Tools for the Analysis of Quantitative Formal Models - (QComp 2019 Competition Report). In Dirk Beyer, Marieke Huisman, Fabrice Kordon, and Bernhard Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part III*, volume 11429 of *Lecture Notes in Computer Science*, pages 69–92. Springer, 2019

- Jörg Hoffmann, Holger Hermanns, Michaela Klauck, Marcel Steinmetz, Erez Karpas, and Daniele Magazzeni. Let's Learn Their Language? A Case for Planning with Automata-Network Languages from Model Checking. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pages 13569–13575. AAAI Press, 2020

- Timo P. Gros, Holger Hermanns, Jörg Hoffmann, Michaela Klauck, and Marcel Steinmetz. Deep Statistical Model Checking. In Alexey Gotsman and Ana Sokolova, editors, *Formal Techniques for Distributed Objects, Components, and Systems - 40th IFIP WG 6.1 International Conference, FORTE 2020, Held as Part of the 15th International Federated Conference on Distributed Computing Techniques, DisCoTec 2020, Valletta, Malta, June 15-19, 2020, Proceedings*, volume 12136 of *Lecture Notes in Computer Science*, pages 96–114. Springer, 2020

- Christel Baier, Maria Christakis, Timo P. Gros, David Groß, Stefan Gumhold, Holger Hermanns, Jörg Hoffmann, and Michaela Klauck. Lab Conditions for Research on Explainable Automated Decisions. In Fredrik Heintz, Michela Milano, and Barry O'Sullivan, editors, *Trustworthy AI - Integrating Learning, Optimization and Reasoning - First International Workshop, TAILOR 2020, Virtual Event, September 4-5, 2020, Revised Selected Papers*, volume 12641 of *Lecture Notes in Computer Science*, pages 83–90. Springer, 2020

- Carlos E. Budde, Arnd Hartmanns, Michaela Klauck, Jan Křetínský, David Parker, Tim Quatmann, Andrea Turrini, and Zhen Zhang. On Correctness, Precision, and Performance in Quantitative Verification - QComp 2020 Competition Report. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Tools and Trends - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part IV*, volume 12479 of *Lecture Notes in Computer Science*, pages 216–241. Springer, 2020

- Timo P. Gros, David Groß, Stefan Gumhold, Jörg Hoffmann, Michaela Klauck, and Marcel Steinmetz. TraceVis: Towards Visualization for Deep Statistical Model Checking. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Tools and Trends - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part IV*, volume 12479 of *Lecture Notes in Computer Science*, pages 27–46. Springer, 2020

- Christel Baier, Clemens Dubslaff, Holger Hermanns, Michaela Klauck, Sascha Klüppelholz, and Maximilian A. Köhl. Components in Probabilistic Systems: Suitable by Construction. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part I*, volume 12476 of *Lecture Notes in Computer Science*, pages 240–261. Springer, 2020

- Rasha Faqeh, Christof Fetzer, Holger Hermanns, Jörg Hoffmann, Michaela Klauck, Maximilian A. Köhl, Marcel Steinmetz, and Christoph Weidenbach. Towards Dynamic Dependable Systems Through Evidence-Based Continuous Certification. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Engineering Principles - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October*

20-30, 2020, Proceedings, Part II, volume 12477 of *Lecture Notes in Computer Science*, pages 416–439. Springer, 2020

- Maximilian A. Köhl, Michaela Klauck, and Holger Hermanns. Momba: JANI Meets Python. In Jan Friso Groote and Kim Guldstrand Larsen, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part II*, volume 12652 of *Lecture Notes in Computer Science*, pages 389–398. Springer, 2021

- Michaela Klauck and Holger Hermanns. A Modest Approach to Dynamic Heuristic Search in Probabilistic Model Checking. In Alessandro Abate and Andrea Marin, editors, *Quantitative Evaluation of Systems - 18th International Conference, QEST 2021, Paris, France, August 23-27, 2021, Proceedings*, volume 12846 of *Lecture Notes in Computer Science*, pages 15–38. Springer, 2021

- Timo P. Gros, Daniel Höller, Jörg Hoffmann, Michaela Klauck, Hendrik Meerkamp, and Verena Wolf. DSMC Evaluation Stages: Fostering Robust and Safe Behavior in Deep Reinforcement Learning. In Alessandro Abate and Andrea Marin, editors, *Quantitative Evaluation of Systems - 18th International Conference, QEST 2021, Paris, France, August 23-27, 2021, Proceedings*, volume 12846 of *Lecture Notes in Computer Science*, pages 197–216. Springer, 2021

- David Groß, Michaela Klauck, Timo P. Gros, Marcel Steinmetz, Jörg Hoffmann, and Stefan Gumhold. Glyph-based visual analysis of q-learning based action policy ensembles on racetrack. In *26th International Conference on Information Visualisation (IV)*, 2022

- Timo P. Gros, Holger Hermanns, Jörg Hoffmann, Michaela Klauck, Maximilian A. Köhl, and Verena Wolf. MoGym: Using Formal Models for Training and Verifying Decision-making Agents. In *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings*, 2022

Journal Papers

- Michaela Klauck, Marcel Steinmetz, Jörg Hoffmann, and Holger Hermanns. Bridging the Gap Between Probabilistic Model Checking and Probabilistic Planning: Survey, Compilations, and Empirical Comparison. *Journal of Artificial Intelligence Research*, 68:247–310, 2020

- Timo P. Gros, Holger Hermanns, Jörg Hoffmann, Michaela Klauck, and Marcel Steinmetz. Analyzing Neural Network Behavior through Deep Statistical Model Checking, 2022. under submission

- Timo P. Gros, Joschka Groß, Daniel Höller, Jörg Hoffmann, Michaela Klauck, Hendrik Meerkamp, and Verena Wolf. DSMC Evaluation Stages: Fostering Robust and Safe Behavior in Deep Reinforcement Learning, 2022. under submission

1. Introduction

Already today our life is full of *cyber-physical systems (CPS)*, i.e., computing systems in which the control software is meant to govern a physical and mechanical part [8, 223]. We use them all day long, when brushing our teeth in the morning with an intelligent electric tooth brush, or when interacting with all our small smart home devices which make our daily life much easier [70, 80, 231]. But they also play a major role in large industry machines [162], in robotics [72, 237], in medical monitoring [79, 196], in traffic management systems guiding cars which drive more and more autonomously [145], and in air traffic [254]. We are in the process of developing Industry 4.0, autonomous cars, smart homes, and smart cities. Who knows which innovations we can expect in the future?

Many of the innovations in recent years have been enabled by *neural networks (NNs)*. We often interact with *intelligent systems* which use NNs to take decisions for us or recommend which choice seems to be optimal [213, 269]. These techniques are used, e.g., in personalized advertisements [151, 232], but also in CPS, like industry robots [72], robot assisted surgeries [237], computer guided rescue missions [41], and autonomous cars [40]. All these examples demonstrate that increasingly complex software participates in actions and decisions that affect humans directly.

1.1. System Verification

We interact with all of these small and large intelligent and cyber-physical devices very closely, and often without thinking about any safety risks. In some cases that is fine because failures are not critical and do not harm. In other cases small imprecisions or errors can lead to losses of lives, e.g., in accidents with self-driving cars [265], or to huge economic damage, e.g., when software is not tested adequately [222]. An exemplary question of a specific use case which will be with us throughout the thesis is: Shall we really trust a neural network in the driving seat of our car?
More generally, the main question is: *How can we achieve that we trust the systems we are interacting with in our daily lives?*

This shows that there is a growing need for verification of cyber-physical and intelligent systems. We often do not understand how these systems interact with their components,

and why certain decisions are taken by the systems. One big reason is that the systems become increasingly complex which makes it ever harder to verify them. All these devices, machines, and systems consist of multiple components all interacting with each other, and sometimes even with other systems in their surrounding. Nevertheless, it is urgently required for software applications in, e.g., an autonomous car, that all components the car consists of are guaranteed to work as intended. If this is the case, other parts of the system can rely on the correctness of the hardware and software systems in use. In the example of an autonomous car this is then also true for the passengers who can feel safe upon entering the car.

To summarize, the consequences of loosing the understanding and control over systems surrounding us become more and more severe during the technological progress. This shows the need for *perspicuous systems*, i.e., systems which operate lucidly, understandably, transparently, comprehensibly, and clearly. For perspicuous systems, it should be possible to describe their behavior using formal mathematical or computational constructs. In addition, perspicuity is provided by interactive visualizations and verbalizations allowing potential users of the system with different background knowledge to explore the system's behavior and understand it. This also opens the systems for usage in multiple disciplines. It should be possible to make the behavior of systems plausible to the users, such that they can gain trust into the systems' operation and decisions.

Our *first goal* in this thesis is to contribute research which enables us to tell how high the safety risk of a given complex system is and what properties it has. Since before being able to explain why systems behave in a certain way, it is extremely important to verify that they do what they are intended to do, and especially that they operate in a safe manner. If that is the case, other systems, but first and foremost we as humans interacting with CPS, can rely on them and be sure to be safe during interaction. The definition of safety always depends on the specific context.

On top of that, as a *second goal*, we want to provide approaches helping in explaining and in giving reasons why a certain decision has been taken in a system or what exactly the cause for an error was, i.e., we want to provide strategies to make systems more perspicuous. This does not only help users in gaining trust in the systems by understanding their decisions but also gives system engineers deeper insights usable to enhance the functionality and quality.

But before being able to verify a system or to assess its quality by understanding what it does, it has to be specified unambiguously what its functionality should be. This can, for example, be done by designing a *formal model* specifying its behavior, on which the verification and inspection can be conducted later. Most systems can be modeled as networks of parallel probabilistic automata in one of the established modeling languages, like the JANI-model format [53], the Prism language [190], or Modest [123]. The systems we are

talking about in this thesis are often composed of several components working together in parallel. The communication and interaction of these parts has to be modeled accurately. In such systems, regularly decisions between multiple options for actions have to be taken. Often probabilities specify how likely it is for a behavior to happen, e.g., when (parts of) the system can be modeled as a stochastic process, when randomization is used, or when unreliable or unpredictable events occur. This behavior is modeled by *probabilistic actions*. If the system is underspecified, it can only be modeled abstractly. This is sometimes also done to leave implementation freedom or to model multiple options at once. When the probability distribution over actions is not clear in such cases, the states in which these actions occur are modeled *nondeterministically*. If this is the case, we say that *nondeterminism* occurs in the model. This exemplifies that the modeling process of concurrent probabilistic systems is often quite complex.

After having build a model, interesting properties can be specified for it. Often the performance or the reliability of the entire system is of interest and properties like the failure probability, or the estimated resources consumed to fulfill a predefined task are calculated.

The solution approaches of *formal methods* [94, 157] rely on such precise and unambiguous formal specifications of the system and the property under investigation, which need to be verified to match each other to be sure that the system does exactly what it is supposed to do.

One possible verification approach is called *theorem proving* [39], where given a formal model and the required properties, one tries to verify that the property holds with the help of an interactive proof assistant. This, however, can be very challenging and time consuming because the key steps often rely on manual proof steps.

Another approach, sometimes more, sometimes less decoupled from formal methods, is to conduct *tests* on the CPS itself to check if the behavior matches the expectations. But there are several limitations to this approach: Finding no errors during a test does not imply that the system always behaves correctly. Being sure that the system is entirely error-free is only possible after exhaustive tests of the full system. Testing all possible scenarios to be sure that the system is always behaving correctly is often infeasible because there are too many options. In the presence of nondeterminism the approach is infeasible because the nondeterminism is not controllable to reproduce all possible scenarios. Sometimes also an infinite number of scenarios would need to be checked. Even if a testing strategy is applicable, it is quite costly, e.g., for large industrial systems, because it takes excessively long and in some cases uses a lot of resources [183].

Studies have shown that model checking is a better technique in this respect, because it is less time consuming, requires less adjustments, and finds substantially more bugs [38]. *Model checking* [22, 64, 68, 86, 242, 261] is a set of fully automated techniques to check a

given formal property, i.e., a formal specification of the requirements, on a formal model of the system.

There are two relevant flavors of model checking, both appearing in this thesis on probabilistic models called *Markov Decision Processes (MDPs)* [226] with probabilistic actions and nondeterministic behavior. These are (i) *exhaustive probabilistic model checking* [15, 17, 19, 22, 69, 90, 168, 188, 260] and (ii) *statistical model checking (SMC)* [28, 44, 51, 141, 197, 198, 240, 273, 271, 275]. The former works on the entire state space of the model and checks for all possible executions, i.e., across the reachable part of the state space, whether the property under consideration holds. This approach of checking a property for all possible executions has similarities to exhaustive testing but is done on the formal model of the system instead of the system itself, which in many cases makes it feasible to check all possible scenarios in a reasonable amount of time with low costs in comparison to real testing [38]. In addition, probabilistic model checking is applicable to nondeterministic models in contrast to testing. But even for exhaustive probabilistic model checking some CPS are too complex, the model gets too large, and the approach is not feasible anymore w.r.t. time or memory consumption. This phenomenon of issues arising when trying to treat increasingly large state spaces is called the *state space explosion problem* [66, 67].

The second approach for model checking, *statistical model checking*, is in general only applicable to deterministic models. It is not exhaustive and employs efficient sampling techniques to statistically check the validity of a certain formal property. In essence, it builds up one run through the system model's state space after the other until enough sample runs are at hand to provide a result within predefined statistical bounds, i.e., required memory resources are much lower. This means, hypothesis testing on finitely many simulations of the system are performed to get a statistical evidence for property satisfaction or violation. SMC is in general only applicable to models without nondeterminism because during the execution of the sample runs through the system, it would otherwise not be clear how to resolve nondeterminism and how to deal with the individual results afterwards. Only in specific cases under specific assumptions SMC can be used to analyse nondeterministic models like MDPs.

To meet our goals in this thesis, which we defined above, we build up on both of the introduced variants of probabilistic model checking. We first present a new approach for probabilistic model checking by making use of techniques proven successful in the automated planning community, which do not necessarily explore the whole state space but concentrate on regions relevant for the property under investigation. Thereby, they often operate more efficiently w.r.t. time and memory than exhaustive model checkers. The resulting dynamic heuristic search model checker Modysh [173] shows promising performance in comparison

to other classical state-of-the-art model checkers and even outperforms them on very large benchmarks.

Second, we present *Deep Statistical Model Checking (DSMC)* [106], an approach based on statistical model checking, usable to analyze and verify NNs and other decision-making agents, which take decisions resolving nondeterminism in MDPs modeling their environment. We demonstrate how DSMC can help people from different backgrounds, like domain engineers, learning experts, engineers in system approval or certification as well as end users to assess the quality of the neural network's decisions, to gain an understanding why the system using the NN behaves in a certain way, and why the NN takes a certain decision. This analysis makes NNs and systems using such decision-making agents much more perspicuous.

Both approaches have in common that they avoid the exploration of the whole state space of the system under investigation at once by only considering single executions and a small part of the state space of the model relevant to answer the property of interest. Thereby, they often avoid issues related to the state space explosion problem. The contributions have both been implemented as part of the well-established Modest Toolset[1] [128].

To summarize, this thesis presents different approaches on how to verify cyber-physical systems and systems incorporating trained decision-making agents, like NNs, automatically with the help of recent methods known from the model checking community in combination with ideas of the planning and the learning community.

1.2. Using Planning and Heuristic Search for Model Checking

Automated probabilistic planning [98, 184, 203, 212, 270] operates as follows: Given an initial state and a goal state description together with a set of possible actions, planners try to find a sequence of actions which transform the initial state description in such a way that the outcome meets the requirements of the goal state specification. Often the focus also lies on how to reach a certain goal in an optimal way, which could be the fastest, safest, or cheapest way. To find an optimal sequence as fast as possible, often heuristic search methods [27, 45, 46, 125] are used. In those approaches, approximations or other hints, e.g., provided by functions called *heuristics*, are exploited to guide the search, instead of trying out every possibility.

In general, a model checking problem with a certain property under investigation can be turned into a planning task in which goal states are states violating this property [62]. If a plan is found for this task, it constitutes a counterexample which shows that the property does

[1] https://www.modestchecker.net/

not hold. This demonstrates that there is quite some similarity between planning and model checking, which has already been exploited in several facets. For example, compilations from planning into model checking languages and vice versa [24, 150, 175, 176, 206, 218] have been used, and algorithmic ideas have been exchanged between the communities [7, 84, 185]. More details on that are given in Section 2.3 and 4.3.

In this thesis however, we do not make use of the fact that model checking tasks can be compiled into planning problems, but we adapt algorithmic techniques known from automated probabilistic planning to solve model checking tasks directly, and in a more efficient way w.r.t. time and memory consumption than other state-of-the-art model checkers. We present a new approach for probabilistic model checking inspired by *probabilistic planning* methods [81, 184, 203, 248], heuristic search, and a variation of *asynchronous value iteration*, which tries to compute optimal values for reachability probabilities and expected accumulated rewards based on only a small fraction of the states in the system model's state space.

In the last years, a large part of these works in the planning community concentrated on reachability analysis of *MaxProb* properties [180, 251, 256], which is the calculation of maximal reachability probabilities. The algorithms we build upon also stem from this area of research. While contributions to this research line are manyfold in the literature, as discussed in Chapter 4, they are quite fragmented with respect to the assumptions on property types and model characteristics they make.

In our contribution, which is the algorithmic design and the implementation of the probabilistic model checker Modysh, we instead take care to efficiently support all established property types, from reachability probabilities to accumulated reward expectations (but no long-run averages and nested properties), also including bounded versions of these property types, on MDPs with positive and zero-valued rewards. We harvest and extend a collection of modified versions of asynchronous value iteration based on heuristic search methods. The core components of our approach are the *Labeled Real-Time Dynamic Programming (LRTDP)* [45] and *Find-Revise-Eliminate-Traps (FRET)* [180] procedures. The implementation of Modysh is based on the combination of these algorithms with several modifications to make the approach work for MDPs with positive and zero-valued rewards on all established property types (except long-run averages and nested properties) listed above. As a result, our tool Modysh considerably enlarges the property types and problem structures supported by heuristic search methods. The new elements of the algorithms and their integration in the basic versions are described in detail in Chapter 4. We also give correctness and optimality proofs for our modified algorithms. A large empirical evaluation, based on the concept of the QComp competition [54, 121], shows that Modysh is competitive relative to state-of-the-art

model checkers and is able to solve benchmark instances which are too large to be solved by any other tool.

MODYSH is shipped as an extension component to the MODEST TOOLSET [128] inside which it can be considered as an alternative to MCSTA [119, 122, 129], which is an exhaustive explicit-state probabilistic model checker based on traditional value iteration. The functionality and architecture of the MODEST TOOLSET is described in Section 3.4. The details on the algorithmic adaptions, the implementation in MODYSH, and the comprehensive benchmarking and scalability study are the subject of Chapter 4.

1.3. Quality Assessment of Trained Decision-Making Agents in Systems

Neural networks, in particular deep neural networks, are the most prominent representatives of a large and diverse set of trained decision-making agents developed in the AI community. Trained decision-making agents in general, but especially NNs, promise astounding advances across a manifold of computing applications in domains as diverse as image classification [182], natural language processing [146], and game playing [244]. Over the last years, it has been shown that *reinforcement learning (RL)* algorithms can approximate optimal decision policies by training deep NNs with very good performances on various complex tasks [209]. As a result, NNs are pervading the technical core of ever more intelligent systems, created to assist or replace humans in decision-making. They are being proposed for automated decision-making under uncertainty in safety-critical cyber-physical contexts, for example, for the purpose of navigating autonomous vehicles through city traffic.

Against this background, techniques are urgently needed which can assert that the decisions made by NNs meet crucial requirements w.r.t. robustness, explainability, perspicuity, and dependability. This development comes with the urgent need to devise methods to analyze and verify desirable behavioral properties of such systems. Unlike for traditional programming methods, this endeavor is hampered by the nature of NNs, whose complex function representation is not suited to human inspection and is highly resistant to mechanical analysis and thus is highly complex to analyze automatically.

As a matter of fact, remarkable progress is being made towards automated NN analysis, be it through specialized reasoning methods of the SAT-modulo-theories family [85, 156, 170], through suitable variants of abstract interpretation [99, 201, 202, 245], or quantitative analysis [71, 268]. All these works thus far focus on the verification of individual NN decision episodes, i.e., the behavior of a single input/output function call of the NN.

In contrast, the verification of NNs being the decisive (in the literal sense of the word) authorities inside larger systems placed in possibly uncertain contexts, i.e., the verification of

the overall intelligent systems encompassing NNs, is wide-open scientific territory. Methods to systematically analyze and ideally verify behavioral properties of such systems are needed. This in particular requires the analysis of *all possible situations that may result from sequences of NN decisions*, which is extremely challenging as it combines the complexity of analyzing the NN with that of analyzing the induced system behaviors and interactions. That is, we have to address the combination of neural network intricacy *and* the state space explosion problem.

One step into this direction has been done by computing predicate abstractions of the subgraph of the state space induced by an NN action policy to verify safety properties [264]. Another approach presents an integrated combination of program analysis together with the analysis of NN decisions with the help of abstract domains [59].

We present a different approach, called *Deep Statistical Model Checking (DSMC)*, concentrating on formal models and the verification of trained decision-making agents, with a strong focus on NN decisions.

The focus on formal models to describe the environment of decision-making agents has a clear benefit. During training, these environments are typically specified implicitly in the form of simulation code in the learning community, like in the Arcade Learning Environment for Atari games [30]. This makes it even harder to check consistency properties on the agents and is an impediment to the use of such representations in safety-critical applications [262]. If one strives for a principled understanding of the power of RL algorithms or of the properties of a specific learned agent in a possibly uncertain environment, a formal, mathematically precise, and unambiguous description of the *training environment* appears as a central asset. A very natural formal model for studying the principles, requirements, efficacy, and robustness of a trained decision-making agent is the model family of Markov decision processes (MDPs) [226]. Employing a formal, precise, well-specified, and unambiguous model of the environment instead of an informally programmed one, bears the promise of enabling rigorous assessment of decision-making agent properties via formal methods.

Deep Statistical Model Checking has been developed by us for settings where one is facing a problem which can be described in terms of an MDP, for which a decision-making entity, like an NN, has been developed by a different party. With DSMC it is then possible to use the MDP as a context to study properties of the NN acting in the environment. Concretely, the NN will be put to use as a determinizer of the otherwise nondeterministic choices in the MDP, so that altogether a Markov chain results, which in turn can be evaluated by statistical model checking [141, 273]. This combination of the NN and the MDP given to SMC results in a scalable approach able to tackle the complexity of analyzing the participating NN *in combination with* that of analyzing the induced system behaviors and interactions.

The idea can be further extended by making the technology available to a certification authority responsible for NN system approval, or to the party designing the NN, as a valuable feedback mechanism in the design process.

For the sake of experimentation and for use by third parties, we have implemented a generic connection between NNs as well as arbitrary decision-making agents connected via a Python function, and the state-of-the-art statistical model checker MODES [43, 51], part of the MODEST TOOLSET [128]. This extension of MODES gives the possibility to use a decision-making agent as a determinizer and to analyze the resulting Markov chain by SMC. We thus *established the first DSMC tool infrastructure*. This infrastructure is also part of MOGYM [107], the toolbox enabling the training and verification of decision-making agents based on formal models.

DSMC as a scalable method for verification and quality assessment of trained decision-making agents is the core idea Chapter 5 proposes, where also the tooling in MODES and MOGYM is described. In addition, we demonstrate in a comprehensive scalability study in Section 5.3 that DSMC is indeed a lightweight approach and scales well even on extremely large state spaces. That DSMC makes intelligent systems including NNs more perspicuous is shown in Section 5.4, where the TRACEVIS tool is demonstrated which visualizes data collected during the DSMC analysis to make the NN decisions more transparent.

Racetrack. The running example of the chapter centered around DSMC are autonomous vehicles, e.g., self-driving cars, in a variant of the Racetrack benchmark, which we introduce in detail in Section 3.3. Racetrack is an instance of many further examples representing real-world phenomena that involve randomness and decision-making. This is the natural scenario where NNs are taking over ever more duties. Racetrack constitutes a benchmark originating in AI autonomous decision-making [27, 221], contains basic features of automated decision-making contexts and can be extended with various further features, ultimately encompassing the scope of autonomous driving. In full generality, the *automated driving challenge* is a vision of (i) swarms of autonomous vehicles, (ii) navigating nimbly through dense traffic, (iii) using advanced object and position recognition systems, (iv) respecting all speed limits and other traffic regulations, (v) operating with minimal fuel consumption, (vi) dynamically adapting to weather and road conditions, (vii) all that in order to get passengers swiftly and safely to their individual destinations. This autonomous driving vision is among the most acclaimed applications of future intelligent systems.

To study the potential of DSMC, we use the Racetrack case study family in a variant that abstractly resembles the autonomous driving challenge albeit with some drastic restrictions relative to the grand vision. These restrictions are: (i) We consider a single vehicle, there is no traffic otherwise. (ii) No object or position sensing is in use, instead the vehicle is aware

of its exact position and speed as well as a description of its environment in form of a *map*. (iii) No speed limits or other traffic regulations are in place. (iv) Fuel consumption is not optimized for. (vi) Weather and road conditions are constant. (vii) The entire problem is discretized in a coarse manner. What remains after all these restrictions (apart from inducing a roadmap of further works beyond what we study) is the problem of navigating a vehicle from start to goal on a discrete map, with actions allowing acceleration and deceleration in discrete directions, subject to a probabilistic risk of an action failing to take effect in each step. The objective is to reach the goal without bumping into a boundary wall. In formal terms, each environment description in form of a map and parameter combination induces an MDP.

Racetrack is a simple problem, simple enough to put a neural network in the driver seat: This NN is then the central authority in the vehicle control loop. It needs to take action decisions with the objective to navigate the vehicle safely towards the goal. There are a good number of scientific proposals on how to construct and train an NN for mastering such tasks, and this thesis is not aiming to innovate in this respect. Instead, the central contribution of DSMC is a scalable method to verify the effectiveness of an NN trained externally for its task. This technique is by no means bound to the Racetrack problem domain, instead it is generally applicable. We evaluate it in the context of Racetrack because we think that this is a crisp formal model family, which is of value in ongoing activities to systematize our understanding of NNs that are supposed to take over important decisions from humans.

1.4. Outline and Contributions

Outline. In Chapter 2 we introduce the theoretical background and lay the mathematical foundations to dive deeper into the world of probabilistic model checking techniques as well as probabilistic planning and heuristic search. In addition, we give a brief introduction to neural networks and learning techniques.

Chapter 3 describes the modeling context of the benchmarks used in case studies throughout the thesis and the context of the tool implementations. It introduces the JANI modeling format in which the models of the Quantitative Verification Benchmark Set (QVBS), we make use of, are given. In addition, it briefly describes the QVBS and the QComp competition, which is based on these benchmarks, together with the concepts of the competition we use. In addition, the Racetrack benchmark is discussed in detail. In the end, the functionality and architecture of the Modest Toolset is presented, in which all tool contributions of the thesis are integrated.

In Chapter 4, Modysh, a new engine of the Modest Toolset, adapting dynamic heuristic search for model checking, is presented together with all modifications and extensions made

to the underlying theory. We also give correctness as well as optimality proofs for the adapted algorithms. A comprehensive benchmarking and scalability study on MODYSH is performed based on the concepts of the QComp competition.

The Deep Statistical Model Checking approach is discussed in detail in Chapter 5. We first explain the theoretical contribution and procedure of the whole DSMC approach by introducing NNs as MDP action oracles, and demonstrating how DSMC can be applied for quality assurance and learning pipeline analysis in multiple case studies. In addition, the DSMC tooling infrastructure implemented in MODES and integrated in MOGYM is presented. In a comprehensive benchmarking and scalability study we show that DSMC scales very well. At the end of the chapter, TRACEVIS, a visualization tool to make DSMC analysis results on Racetrack more perspicuous is presented briefly.

Chapter 6 summarizes the contributions of the thesis and gives an outlook on open topics and future works for which we layed the foundation and paved the way.

The chapters of the main part of the thesis each start with a short summary paragraph as well as a survey of the main contributions and an outline of the chapter together with a clear and detailed statement about the contributions of the author and the origins of the content. A summary and overview regarding the contributions of the thesis and the contributions of the author for the whole thesis is already given in the following.

Contributions. The main contributions of the thesis are part of Chapter 3, and especially Chapter 4 and 5.

We introduce the Quantitative Verification Benchmark Set, which was inspired by previous works of the author in the planning community, which has been set up with the help of the author mainly together with Arnd Hartmanns and Tim Quatmann, and to which the author contributed a lot of benchmarks. In addition, the author was part of a team, consisting of the same collaborators, initiating the QComp competition based on this benchmark collection, and was responsible for the technical setup, tool execution, and result evaluation in the second edition of the competition. Furthermore, we introduce the Racetrack benchmark as a JANI MDP model and briefly summarize the variants of it developed by the author for different use cases in multiple projects. We expect this benchmark to be essential in many future works on the autonomous driving challenge and also in general in works on verification techniques for cyber-physical systems.

With MODYSH, we present a new model checking engine integrated in the MODEST TOOLSET, which is based on modified and extended versions of well known probabilistic planning approaches using dynamic heuristic search. These modifications and extensions together with the implementation of the resulting algorithms have been done by the author. Discussions

with Holger Hermanns, Jörg Hoffmann, and Marcel Steinmetz about the theory were very helpful. The implementation is applicable to MDP structures with positive and zero-valued rewards on all established property types (except long-run averages and nested properties), i.e., maximal and minimal reachability probabilities, expected rewards, and bounded versions thereof. We demonstrate in a comprehensive benchmarking study, conducted by the author, that MODYSH is more efficient w.r.t. time and memory consumption than other state-of-the-art model checkers, especially on very large benchmarks with symmetric structures. With this benchmarking study, we basically present a new edition of the *default probably ε-correct track* of QComp 2020 with the newest versions of the participating tools.

The contributions of the works around Deep Statistical Model Checking are manyfold. We present DSMC to assess the quality of trained decision-making agents, like NNs, in systems formalized as MDPs. DSMC uses statistical model checking to evaluate the connection of the decision-making agent determinizing the MDP, which gives insights into the quality of the agent's decisions. This approach has been developed by the author in collaboration with Holger Hermanns, Jörg Hoffmann, Timo P. Gros, and Marcel Steinmetz. The author established tool infrastructure for DSMC within MODES to connect to NNs and to general decision-making agents, i.e., arbitrary oracles, resolving the nondeterminism in MDP environments. With MOGYM, the author together with Maximilian A. Köhl and Timo P. Gros, provides an integrated tool for training decision-making agents on formal models and their verification with DSMC. How to use DSMC for quality assurance and learning pipeline assessment is demonstrated in case studies on the Racetrack benchmark, conducted by the author. In an exhaustive scalability and performance evaluation of DSMC along multiple dimensions, also done by the author, we show that DSMC is really a lightweight approach which scales well. To perform all these case studies and experiments, the author established infrastructure for Racetrack benchmarking, including parsing maps, JANI model generation and export, and comparison of the decision-making agent's performance with optimal behavior. Furthermore, the author extended the standard DSMC implementation to be able to extract additional analysis data to get deeper insights into the NN decisions and to make them more perspicuous with the help of the visualization in the TRACEVIS tool.

1.5. Origins of the Chapters

In the following, the publications of the author are ordered w.r.t. the chapters their content appears in this thesis. For each of the publications, it is stated what the contributions of the author to the paper are. More details are given at the beginning of each of the chapters.

- Arnd Hartmanns, Michaela Klauck, David Parker, Tim Quatmann, and Enno Ruijters. The Quantitative Verification Benchmark Set. In Tomás Vojnar and Lijun Zhang,

editors, *Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part I*, volume 11427 of *Lecture Notes in Computer Science*, pages 344–350. Springer, 2019

The Quantitative Verification Benchmark Set has been built up with the help of the author and was motivated by works of the author not covered in this thesis [148, 175, 176]. It is introduced in Section 3.2.

- Ernst Moritz Hahn, Arnd Hartmanns, Christian Hensel, Michaela Klauck, Joachim Klein, Jan Křetínský, David Parker, Tim Quatmann, Enno Ruijters, and Marcel Steinmetz. The 2019 Comparison of Tools for the Analysis of Quantitative Formal Models - (QComp 2019 Competition Report). In Dirk Beyer, Marieke Huisman, Fabrice Kordon, and Bernhard Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part III*, volume 11429 of *Lecture Notes in Computer Science*, pages 69–92. Springer, 2019

  The author was part of the team organizing the first QComp competition, which was partially influenced by previous works of the author [175, 176]. It is covered in Section 3.2 together with the publication below.

- Carlos E. Budde, Arnd Hartmanns, Michaela Klauck, Jan Křetínský, David Parker, Tim Quatmann, Andrea Turrini, and Zhen Zhang. On Correctness, Precision, and Performance in Quantitative Verification - QComp 2020 Competition Report. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Tools and Trends - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part IV*, volume 12479 of *Lecture Notes in Computer Science*, pages 216–241. Springer, 2020

  In the second QComp competition, the author was responsible for the technical setup, tool execution, and result evaluations.

- Christel Baier, Maria Christakis, Timo P. Gros, David Groß, Stefan Gumhold, Holger Hermanns, Jörg Hoffmann, and Michaela Klauck. Lab Conditions for Research on Explainable Automated Decisions. In Fredrik Heintz, Michela Milano, and Barry O'Sullivan, editors, *Trustworthy AI - Integrating Learning, Optimization and Reasoning - First International Workshop, TAILOR 2020, Virtual Event, September 4-5, 2020,*

Revised Selected Papers, volume 12641 of *Lecture Notes in Computer Science*, pages 83–90. Springer, 2020

This publication summarizes works in CPEC[2] centered around the Racetrack use case presented in Section 3.3. Many of them have been co-authored by the author of this thesis. The corresponding website[3] has been coordinated by the author.

- Michaela Klauck and Holger Hermanns. A Modest Approach to Dynamic Heuristic Search in Probabilistic Model Checking. In Alessandro Abate and Andrea Marin, editors, *Quantitative Evaluation of Systems - 18th International Conference, QEST 2021, Paris, France, August 23-27, 2021, Proceedings*, volume 12846 of *Lecture Notes in Computer Science*, pages 15–38. Springer, 2021

  The paper introduces the Modysh tool and the base algorithmic innovations of it as detailed in Chapter 4. All modifications and extensions to the underlying algorithms and their implementation have been done by the author.

- Timo P. Gros, Holger Hermanns, Jörg Hoffmann, Michaela Klauck, and Marcel Steinmetz. Deep Statistical Model Checking. In Alexey Gotsman and Ana Sokolova, editors, *Formal Techniques for Distributed Objects, Components, and Systems - 40th IFIP WG 6.1 International Conference, FORTE 2020, Held as Part of the 15th International Federated Conference on Distributed Computing Techniques, DisCoTec 2020, Valletta, Malta, June 15-19, 2020, Proceedings*, volume 12136 of *Lecture Notes in Computer Science*, pages 96–114. Springer, 2020

  This is the publication introducing Deep Statistical Model Checking (Chapter 5) and especially the case studies on Racetrack presented in Section 5.2. The author was responsible to implement the DSMC approach and to conduct the case studies on it, which resulted in the heat maps for evaluation purposes. The NN training was done by Timo P. Gros.

- Timo P. Gros, Holger Hermanns, Jörg Hoffmann, Michaela Klauck, Maximilian A. Köhl, and Verena Wolf. MoGym: Using Formal Models for Training and Verifying Decision-making Agents. In *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings*, 2022

  The paper introduces the MoGym framework which comprises the newest version of the DSMC implementation and provides a fully integrated tool chain from learning to verification, as described in Section 5.1.2. The author implemented extended DSMC functionality for MoGym and helped to connect the individual parts of the tool.

[2] https://perspicuous-computing.science
[3] https://racetrack.perspicuous-computing.science/

Timo P. Gros was responsible for the learning infrastructure and Maximilian A. Köhl provided the implementations in Momba.

- Timo P. Gros, Holger Hermanns, Jörg Hoffmann, Michaela Klauck, and Marcel Steinmetz. Analyzing Neural Network Behavior through Deep Statistical Model Checking, 2022. under submission

  This journal article summarizes the works on DSMC. The extensive scalability study of DSMC done by the author, which is the core new contribution of that publication, is part of Section 5.3.

- Timo P. Gros, David Groß, Stefan Gumhold, Jörg Hoffmann, Michaela Klauck, and Marcel Steinmetz. TraceVis: Towards Visualization for Deep Statistical Model Checking. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Tools and Trends - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part IV*, volume 12479 of *Lecture Notes in Computer Science*, pages 27–46. Springer, 2020

  The TraceVis tool is presented in this paper in the version discussed in Section 5.4. The tool was implemented by David Groß and Stefan Gumhold. The author implemented an extended version of DSMC to be able to deliver the data needed for the visualization. The author ran all the experiments for the paper, processed the data, and gave advice, which data of interest to select for the visualization.

Further publications not explicitly included, but often related to topics covered in this thesis, are:

- Michaela Klauck, Marcel Steinmetz, Jörg Hoffmann, and Holger Hermanns. Compiling Probabilistic Model Checking into Probabilistic Planning. In Mathijs de Weerdt, Sven Koenig, Gabriele Röger, and Matthijs T. J. Spaan, editors, *Proceedings of the Twenty-Eighth International Conference on Automated Planning and Scheduling, ICAPS 2018, Delft, The Netherlands, June 24-29, 2018*, pages 150–154. AAAI Press, 2018

  The paper presents the translation from JANI to PPDDL and is part of the author's Master's thesis, and therefore only cited in this dissertation.

- Michaela Klauck, Marcel Steinmetz, Jörg Hoffmann, and Holger Hermanns. Bridging the Gap Between Probabilistic Model Checking and Probabilistic Planning: Survey, Compilations, and Empirical Comparison. *Journal of Artificial Intelligence Research*, 68:247–310, 2020

The comprehensive journal article includes the translation between JANI and PPDDL in both directions as well as an exhaustive comparison of the most popular model checking and planning techniques, and their implementation in different tools. The experiments and the comparisons of all the approaches have been done by the author together within Marcel Steinmetz in close collaboration.

- Jörg Hoffmann, Holger Hermanns, Michaela Klauck, Marcel Steinmetz, Erez Karpas, and Daniele Magazzeni. Let's Learn Their Language? A Case for Planning with Automata-Network Languages from Model Checking. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pages 13569–13575. AAAI Press, 2020

  The paper shows how attractive JANI is for expressing planning tasks because of its various features. The work was mainly lead by Jörg Hoffmann.

- Christel Baier, Clemens Dubslaff, Holger Hermanns, Michaela Klauck, Sascha Klüppelholz, and Maximilian A. Köhl. Components in Probabilistic Systems: Suitable by Construction. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part I*, volume 12476 of *Lecture Notes in Computer Science*, pages 240–261. Springer, 2020

  In this paper the Racetrack benchmark is used to demonstrate newly introduced notions of suitability for system components. The author adapted the use case for this purpose and contributed to the development of the theory and to the evaluation.

- Rasha Faqeh, Christof Fetzer, Holger Hermanns, Jörg Hoffmann, Michaela Klauck, Maximilian A. Köhl, Marcel Steinmetz, and Christoph Weidenbach. Towards Dynamic Dependable Systems Through Evidence-Based Continuous Certification. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Engineering Principles - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part II*, volume 12477 of *Lecture Notes in Computer Science*, pages 416–439. Springer, 2020

  This publication presents a process for continuous certification of systems consisting of multiple components which get independent updates from time to time. The author implemented the simulation and the model.

- Maximilian A. Köhl, Michaela Klauck, and Holger Hermanns. Momba: JANI Meets Python. In Jan Friso Groote and Kim Guldstrand Larsen, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part II*, volume 12652 of *Lecture Notes in Computer Science*, pages 389–398. Springer, 2021

  Momba, the flexible Python framework for constructing, exploring, and verifying formal models is introduced in this paper. Momba was designed and implemented mainly by Maximilian A. Köhl. The implementation work was supported by the author of this thesis. Components of Momba are used in MoGym for learning and DSMC analysis on general formal models.

- Timo P. Gros, Daniel Höller, Jörg Hoffmann, Michaela Klauck, Hendrik Meerkamp, and Verena Wolf. DSMC Evaluation Stages: Fostering Robust and Safe Behavior in Deep Reinforcement Learning. In Alessandro Abate and Andrea Marin, editors, *Quantitative Evaluation of Systems - 18th International Conference, QEST 2021, Paris, France, August 23-27, 2021, Proceedings*, volume 12846 of *Lecture Notes in Computer Science*, pages 197–216. Springer, 2021

  This work integrates DSMC into a feedback-loop for deep reinforcement learning to determine state space regions in which more training is needed. The work was mainly done by Timo P. Gros and Hendrik Meerkamp. The author supported them by giving advice on how to use DSMC.

- Timo P. Gros, Joschka Groß, Daniel Höller, Jörg Hoffmann, Michaela Klauck, Hendrik Meerkamp, and Verena Wolf. DSMC Evaluation Stages: Fostering Robust and Safe Behavior in Deep Reinforcement Learning, 2022. under submission

  This paper is an invited journal article on an extended version of the QEST paper with the same title described above.

- David Groß, Michaela Klauck, Timo P. Gros, Marcel Steinmetz, Jörg Hoffmann, and Stefan Gumhold. Glyph-based visual analysis of q-learning based action policy ensembles on racetrack. In *26th International Conference on Information Visualisation (IV)*, 2022

  In this paper we extend TraceVis to visualize the Q-values used during learning to get deeper insights about the training progress. Again, the implementation was done by David Groß and Stefan Gumhold. The author conducted the user study and gave advice during the tool design and evaluation phase.

2. Theoretical Background

We start in Section 2.1 by laying the foundations of the following chapters, introduce the formal background, and the notions used throughout the thesis. This encompasses the mathematical concepts needed to reason about probabilistic systems, especially *Markov Decision Processes (MDP)*, on which all approaches of this thesis are based. Different measures of interest, i.e., different kinds of property types of the models, are discussed, and probabilistic as well as statistical model checking techniques to calculate these properties are introduced in Section 2.2. Since verification techniques presented in this thesis also make use of probabilistic planning approaches, especially heuristic search methods, the theoretical underpinning of this area is discussed in Section 2.3. Furthermore, a brief introduction into neural networks and Q-learning methods is given in Section 2.4, because we will later present a method to analyze and verify neural networks or trained decision-making agents with the prospect to improve training and learning strategies.

2.1. Mathematical Foundations

First, we introduce the mathematical notation used to represent standard mathematical concepts and recall the theory behind them.

Sets

Sets are collections of *elements*. An object x can be an element of the set X, formally written as $x \in X$. The negation, i.e., x not being in the set X, is expressed by $x \notin X$. A set Y is a *subset* of X if and only if all elements of Y are also elements of X, denoted by $Y \subseteq X$. If X contains additional elements which are not in Y, Y is a *proper subset* of X, written $Y \subsetneq X$.

Sets can be described in two notations. First, if feasible w.r.t. the number of elements, they can be listed, like $X = \{1, 2, 3, 4\}$. Second, a set Y can be defined by a basic set X, and a property A which has to hold on elements of X to be elements of Y, written $Y = \{x \in X \mid A(x)\}$, which means that Y consists of all elements x of X for which $A(x)$ holds, and no more. If it is clear from the context, the basic set can be omitted in the notation.

Throughout the thesis $\mathbb{N}$ is the set of *natural numbers* including 0, $\mathbb{Z}$ denotes the *integers*, $\mathbb{Q}$ are the *rational numbers*, $\mathbb{R}$ is the set of *real numbers*, and $\mathbb{B}$ are the *boolean values True* and *False*.

$\emptyset$ is the *empty set*, which does not contain any elements. The *powerset* $Pow(X) = \{X' \subseteq X\}$ is the set containing all subsets X' of X.

In general, sets are unordered. But for sets of elements for which a certain order can be defined, the notion of *ordered sets* has been introduced. A prominent example of ordered sets are intervals of numbers. We denote by the interval $[a, b]$ over the set X the *closed interval* from a to b, including the boundary values a and b, and all elements of X lying in between a and b according to the order on X. (a, b) is the *open interval* excluding the boundary values. Half closed intervals are defined analogously. Depending on the context it should often be clear if, e.g., an interval of real, or natural numbers, or something else is meant in mathematical notations. It is possible to shrink sets by specifying an interval or bounds in the index, like $\mathbb{N}_{[0,10]}$ or $\mathbb{R}_{\geq 0}$, which describes the natural numbers between 0 and 10 including the bounds, and the real numbers greater or equal to 0, respectively.

In addition, we introduce notations for sums and products over specific sets of numbers. The elements x_i of the set X are indexed over $\mathbb{N}$. For example $\sum_{i=1}^{n} x_i := x_1 + x_2 + \cdots + x_n$ denotes the sum of all elements x_i with $i \in [1, n]$ over $\mathbb{N}$. In the same sense, we use $\prod_{i=1}^{n} x_i$ to express the product over all x_i with $i \in [1, n]$ over $\mathbb{N}$. $\prod_{i=1}^{n} i$ would multiply all natural numbers from 1 to n: $1 \cdot 2 \cdot 3 \cdot \cdots \cdot n$.

The number of elements in a set ist denoted by $|X|$ and is called the *cardinality* of X. A *finite set* has less elements than the natural numbers, i.e., $|X| < |\mathbb{N}|$. A *countably infinite set* has the same cardinality as $\mathbb{N}$, i.e., $|X| = |\mathbb{N}|$. If the set contains more elements than the natural numbers, i.e., $|X| > |\mathbb{N}|$, it is called *uncountable*.
We use the following set operations:

- *Set difference*: $X \setminus Y := \{x \in X \mid x \notin Y\}$, describes the set of all elements which are in X and not in Y. In this context the set difference $X \setminus X_1$ of a subset X_1 of set X is called *complement* of X_1 w.r.t. X, often written as $\overline{X_1}$ if it is clear from the context what X is.

- *Intersection*: $X \cap Y := \{x \in X \mid x \in Y\}$, is the set of all elements which are in X and also in Y.

- *Union*: $X \cup Y := \{x \mid x \in X \vee x \in Y\}$, describes the set of all elements which are in X or $(\vee)$ in Y.

Sets are called *disjoint* if their intersection results in the empty set, i.e., if there is no common element.
The *Cartesian product* $X_0 \times X_1 \times \cdots \times X_n$ of non-empty sets $X_0, X_1, \ldots, X_n$ consists of all *n-tuples* $(x_0, x_1, \ldots, x_n)$ where $x_0 \in X_0, x_1 \in X_1, \ldots, x_n \in X_n$. This means $X_0 \times \cdots \times X_n =$

$\{(x_0, \ldots, x_n) \mid x_i \in X_i, \text{ for } i \in [0, n]\}$. We use X^n for tuples of length n where all components are elements of X. The elements of a Cartesian product of two sets are called *pairs*.

It is also possible to build ordered *sequences* of elements x_i of a set X, e.g., the sequence $x_0\, x_1\, x_2$ containing these three elements in exactly this order. A *prefix* $pref(seq)$ of a sequence seq is a part, or more precisely a subsequence, of seq starting with the first element $seq[0]$ of seq, such that $pref(seq)$ can be extended to seq by adding the missing elements at the end. If $pref(.)$ is applied to a set of sequences X, the result is a set where $pref(.)$ is applied to each element of X.

A finite, non-empty sequence of elements in X of undefined length is denoted by $x_0\, x_1\, x_2 \cdots \in X^+$. Infinite sequences are of the form $(x_i)_{i \in \mathbb{N}} \in X^\omega$.

A *σ-algebra* $A \subseteq Pow(X)$ over a non-empty set X is a subset A of the power set $Pow(X)$ of X, which fulfills the following conditions:

- $X \in A$.
- A is *closed under complement* in X, i.e., if $A_1 \in A$ then its complement $X \setminus A_1$ is also in A.
- A is *closed under countable unions*, which means that any union of subsets of A is again in A, i.e., if the sets $A_1, A_2, A_3, \ldots$ are elements of A then $\bigcup_{n \in \mathbb{N}} A_n$ is also contained in A.

Elements of the σ-algebra A, $A_i \in A$, are called *measurable sets*. The tuple (X, A) is then called *measurable space*.

Relations and Functions

A binary *relation* R is defined by a set of pairs $R \subseteq S_1 \times S_2$ over two sets S_1 and S_2. If every element of S_1 is at most related to a single element of S_2, i.e., if $(s_1, s_2) \in R$ and $(s_1, s_2') \in R$ implies that $s_2 = s_2'$, the binary relation is called *functional*, i.e., the relation is a *function*.

Relations can also be defined over more than two sets. A relation over three sets is called a ternary relation and the elements are called triples. A relation over n sets is called n-ary and the elements are n-tuples.

A *total function* $f : X \to Y$ is a binary relation which assigns to each element of the *domain* X exactly one element of the *codomain* Y. This means, for each $x \in X$ there exists exactly one $y \in Y$ such that $f(x) = y$. y is the image of x, and x is the preimage or inverse image of y. In comparison, a *partial function* $f : X \rightharpoonup Y$ does not have to define a value of the codomain for each element of the domain. In this case, we write $f(x) = \perp$ if there is no image of x in Y (under the assumption that $\perp \notin Y$).

A function f is called *injective* if two different elements of X are never mapped to the same element in Y: $x_1 \neq x_2 \implies f(x_1) \neq f(x_2)$, where $\implies$ denotes a logical implication. A function is called *surjective* if for every $y \in Y$ there is an $x \in X$ with $f(x) = y$, i.e., every y is an image of some x. If a function is injective and surjective, it is called *bijective*.

$f^{-1} : Y \to X$ is the *inverse function* of $f : X \to Y$. Not all functions have an inverse. An injective function is invertible. In this case, it holds $f^{-1}(y) = x \Leftrightarrow f(x) = y$ ($\Leftrightarrow$ means *if and only if*, an equivalence, an implication in both directions). The *preimage* of Y under f is always defined as $f^{-1}(Y) := \{x \in X \mid f(x) \in Y\}$.

Let X and Y be two sets with their respective σ-algebras χ and γ. A function $f : X \to Y$ defined on the two measurable spaces (X, χ) and (Y, γ) is called a *χ-γ-measurable function* if and only if the preimage of every set $G \in \gamma$ under f is an element of χ, i.e., $f^{-1}(G) \in \chi$, for all $G \in \gamma$.

Probability Theory and Probabilistic Models

This thesis revolves around processes with probabilistic behavior. We first recap the basics of probability theory, and later apply these concepts on the concrete systems and models which we investigate. Unless otherwise stated, the theory is based on standard textbooks on probability theory [10, 60, 88, 234, 252].

Definition 1: Probability Space

A *probability space* $(\Omega, \Sigma, \mathcal{P})$ consists of:

- A non-empty set Ω, the *sample space*, which consists of all possible *outcomes* of the probabilistic process under inspection.
- A set of *events* Σ, the *event space*, which contains sets of outcomes, called events, where Σ is a *σ-algebra* over Ω.
- A *probability measure* often also called *probability function* $\mathcal{P} : \Sigma \to [0, 1]$, which assigns to each event in Σ a probability in such a way that the Kolmogorov Axioms hold.

The *Kolmogorov Axioms* on a probability space as defined above state:

- $\mathcal{P}$ is a non-negative function into $\mathbb{R}_{\geq 0}$.
- $\mathcal{P}(\Omega) = 1$.
- For disjoint events E_i of Σ it holds that $\mathcal{P}(\bigcup_i E_i) = \sum_i \mathcal{P}(E_i)$.

For a finite or countably infinite set X, $Pow(X)$ is the default σ-algebra. A *discrete probability distribution* over a finite or countably infinite sample space Ω is a function $\mu : \Omega \to [0, 1]$ such that $\sum_{x \in \Omega} \mu(x) = 1$. If μ is applied on sets of elements of Ω, the sum of μ applied to each element is meant, i.e., $\mu(X) = \sum_i \mu(x_i)$ with $X \subseteq \Omega$ and $x_i \in X$. Thereby, μ induces a probability space with the sample space Ω, the event space $\Sigma = Pow(\Omega)$, and a probability measure $\mathcal{P}$ with $\mathcal{P}(X) = \sum_i \mu(x_i)$, where $x_i \in X$ and $X \in \Sigma$.

We denote by $\mathcal{D}(\Omega)$ the set of all discrete probability distributions over Ω. We write $\delta_x : \Omega \to [0, 1]$ for the *Dirac distribution* that assigns probability 1 only to the element $x \in \Omega$, and 0 to all others.

The *support* of a probability distribution μ is defined as the set of elements of Ω which have a positive probability under μ, i.e., $supp(\mu) := \{x \in \Omega \mid \mu(x) > 0\}$.

Definition 2: Random Variable, Probability Mass & Density Function

Assume a probability space $(\Omega, \Sigma, \mathcal{P})$ as defined above.

$\mathcal{B}(\mathbb{R})$ denotes the *Borel σ-algebra* over $\mathbb{R}$, which is the σ-algebra containing all open subsets of $\mathbb{R}$. *Open subsets* in this context are defined as having no element of $\mathbb{R}$ on their boundaries.

This means, we have the two measurable spaces (Ω, Σ) and $(\mathbb{R}, \mathcal{B}(\mathbb{R}))$.

A *random variable* RV for the probability space is a Σ-$\mathcal{B}(\mathbb{R})$-measurable function $RV : \Omega \to \mathbb{R}$ assigning to each outcome of Ω a value in $\mathbb{R}$.

Its *cumulative distribution function* can be defined as $F : \mathbb{R} \to [0, 1]$, $F(r) := \mathcal{P}(\{\omega \in \Omega \mid RV(\omega) \leq r\})$.

For discrete random variables RV, for which the domain of F is finite or countably infinite, the *probability mass function* is defined as $f(r) := \mathcal{P}(\{\omega \in \Omega \mid RV(\omega) = r\})$.

In case RV is continuous, the cumulative distribution function can also be specified in the following way:

$$F(r) := \int_{-\infty}^{r} h(u)\,du$$

for $r \in \mathbb{R}$ and an integrable function $h : \mathbb{R} \to \mathbb{R}_{\geq 0}$. This function h is called the *probability density function* of RV.

Intuitively, the cumulative distribution function gives the probability with which outcomes of the probabilistic process occur, whose values assigned by the random variable are smaller or equal to the given threshold.

For random variables, it is often of interest which value the variable has *in expectation*, i.e., what the average of the results obtained when executing an experiment an infinite number of times is. Hence, the expectation is also called *mean*. The value depends on the probability distribution.

Definition 3: Expected Value

For discrete random variables RV on a probability space as above, the *expected value* $E(RV)$ is the sum over all results r_i of RV of the product of each possible result r_i and its probability p_i.

$$E(RV) := \sum_i p_i \cdot r_i = \sum_i \mathcal{P}(\{\omega \in \Omega \mid RV(\omega) = r_i\}) \cdot r_i.$$

The sum does not have to be convergent.

If RV is a continuous random variable and h is its probability density function, then we can define the following if the integral exists:

$$E(RV) := \int_{-\infty}^{\infty} r \cdot h(r)\, dr,$$

Probabilities occur very often in processes in the real world, e.g., in biology or chemistry, where reactions or other processes happen with a certain likelihood. But there are also cyber-physical systems or other processes where randomization is involved, e.g., in games, to either distribute things equally or to avoid that always the same order is kept. In addition, probabilistic models are in use if unreliable or unpredictable events should be represented, like package losses or failure rates.

These systems and processes can be modeled using states and transitions describing the transition from one state of the system into another. The very basic type of such models, still abstracting from probabilities, is called *Transition System (TS)*.

Definition 4: Transition System (TS)

A *transition system* TS [22] is a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{T}, I, \text{AP}, L)$, where:

- $\mathcal{S}$ is the set of *states*.
- $\mathcal{A}$ is the set of *actions*.
- $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{A} \times \mathcal{S}$ is the *transition relation*.
- $I \subseteq \mathcal{S}$ is the set of *initial states*.
- AP is the set of *atomic propositions*.
- $L : \mathcal{S} \to Pow(\text{AP})$ is the *labeling function*.

Transition systems are used to model systems, where the states describe information of a certain configuration of the system at some point in time. *Transitions*, given as the elements of the transition relation, describe the process of changes from one system state to another. This means, a transition happens if a certain action characterized by its name is *taken*, i.e., *applied*, in a *start state* s. During the transition the current state of the system is manipulated in a specific way, defined by the action, and changed to another system state, the *destination* or *successor* state, as defined by the element of the transition relation. An action $a \in \mathcal{A}$ is called *applicable* in a state $s \in \mathcal{S}$ if $(s, a, t) \in \mathcal{T}$ for some $t \in \mathcal{S}$. In this case, we also say that s is *connected by a to t* via the transition (s, a, t). Note that not every action has to be applicable in every state and that the same action applied in a certain state can lead to different destination states depending on the transition it belongs to. A state t is called *reachable* from state s if there is a set of intermediate states $\mathcal{S}_{inter}$, such that they are connected via transitions $(s, a_0, s_1), (s_1, a_1, s_2), \ldots, (s_n, a_n, t) \in \mathcal{T}$ with $s_i \in \mathcal{S}_{inter}$ and $a_i \in \mathcal{A}$, finally ending in t. We say that a state t is *reached* if after taking one or multiple transitions one finally ends in t. To be able to describe a system state by known facts characterizing it, atomic propositions are used. The labeling function assigns to each system state the characterizing and describing atomic propositions.

By replacing the possibility to transition with different or even the same action from a state to different successor states with a probabilistic distribution over the successor states, this simplest form of system models can be enhanced with probabilities. The fundamental probabilistic model based on the notion of transition systems only consists of states and probabilistic transitions which lead from a start state to a set of destination states, where for each destination state a probability to reach this state by taking the transition is given. In contrast to transition systems, there is no distinction between different actions anymore. This model is called *Markov Chain* if it fulfills the *Markov Property*, which states that the probability distributions of transitions starting in a state only depend on this state, and not on previous decisions and visited states, i.e., it is *history-independent*, also called *memory-less*.

Definition 5: Finite Markov Chain

A finite *Markov Chain* [153, 158] is a tuple $C = \langle \mathcal{S}, \mathcal{T}, s_0, \mathcal{S}_* \rangle$ consisting of a finite set of *states* $\mathcal{S}$, a *transition probability function* $\mathcal{T} : \mathcal{S} \to \mathcal{D}(\mathcal{S})$, and an *initial state* $s_0 \in \mathcal{S}$ as well as a set of *goal states* $\mathcal{S}_* \subseteq \mathcal{S}$.

We assume that there is no state without an outgoing transition. If there are no transitions leading to another state, we assume that there is a self-loop back to the state itself, which does not change the types of system behavior we are investigating later because the system will stay in this state forever. Such states are called *terminal*. All goal states are assumed to

be terminal. This is no restriction because for the purposes considered in this thesis it is only important if a goal is reached and not what happens afterwards.

To make the models even more realistic for real world and especially cyber-physical systems, *nondeterministic* behavior can be added. *Nondeterminism* means that there is a choice between multiple transition options but it is not specified or quantified at all which one will be taken. Nondeterminism is used, e.g., to represent incompletely or not specified parts of a system, where the probability distribution over multiple options is not known, or to model different instantiations of the system at once, i.e., for abstraction purposes to allow implementation freedom. In cyber-physical systems often multiple components work together concurrently and their unknown scheduling can be modeled with the help of nondeterminism. In addition, nondeterminism is used when there are multiple possible choices and a separate entity not part of the model but of its environment has to decide which choice to take. By adding the possibility to not only have probabilistic transitions but also nondeterministic states, i.e., by adding states with multiple outgoing transitions, where it has to be resolved nondeterministically which one to take, we obtain the models this thesis is centered around, *Markov Decision Processes*.

Definition 6: Finite Markov Decision Process (MDP)

A finite *Markov Decision Process (MDP)* [154, 226] is a tuple $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, s_0, \mathcal{S}_* \rangle$ consisting of a finite set of *states* $\mathcal{S}$, a finite set of *actions* $\mathcal{A}$, the *partial transition probability function* $\mathcal{T} : \mathcal{S} \times \mathcal{A} \rightharpoonup \mathcal{D}(\mathcal{S})$, a reward function $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}_{\geq 0}$ assigning a reward (or cost) value to each triple of state, action, state, a single *initial state* $s_0 \in \mathcal{S}$, and a set of *goal states* $\mathcal{S}_* \subseteq \mathcal{S}$.

We use non-negative reward structures, including zero-valued rewards, like it is often the standard in the probabilistic model checking community [22, 127, 164, 226], because otherwise expected rewards may be unbounded or not even well defined (for more details see [226, Chapter 7]).

An action $a \in \mathcal{A}$ is *applicable* in a state $s \in \mathcal{S}$ if $\mathcal{T}(s, a)$ is defined. In this case, we denote by $\mathcal{T}(s, a, t)$ the probability $\mu(t)$ of going to the *successor state* t according to $\mathcal{T}(s, a) = \mu$. $\mathcal{A}(s) \subseteq \mathcal{A}$ is the set of all actions that are applicable in state s. $\mathcal{T}(s, a, t)$ denotes the probability of going to state t when applying action a in state s. If according to the transition probability function, an action leads with probability 1 to a state, it is called a *Dirac action*. A *nondeterministic state* is a state s where $|\mathcal{A}(s)| > 1$. Similar to what has been introduced for transition systems above, we say that there is a *transition* from state s to state t if there is an action a applicable in s for which $\mathcal{T}(s, a, t) > 0$. In this case, s is *connected to* t *by* action a via this transition, i.e., there exists a *connection* from s to t. Analogously to transition systems, we also say that state t is *reachable* from state s if there is a set of intermediate states

$\mathcal{S}_{inter}$, connected via transitions $\mathcal{T}(s, a_0, s_1) > 0, \mathcal{T}(s_1, a_1, s_2) > 0, \ldots, \mathcal{T}(s_n, a_n, t) > 0$ with $s_i \in \mathcal{S}_{inter}$ and $a_i \in \mathcal{A}$, finally ending in t, i.e., *reaching* t.

The reward function assigns to every transition from one state to another a specific reward depending on the start and destination state as well as the applied action. This enables us, e.g., to reason about the sum of the rewards obtained when taking multiple transitions in a row.

Similar to what we defined for Markov chains, we base our work on MDPs where for each state s, $\mathcal{A}(s)$ is non-empty. If there are no transitions to another state, a self-loop with reward 0 is assumed to exist. This is possible because self-loops do not change the system behavior we are interested in later. A state s is called *terminal* if $|\mathcal{A}(s)| = 1$ and for this $a \in \mathcal{A}(s)$ it holds that $\mathcal{T}(s, a, s) = 1$ and $\mathcal{R}(s, a, s) = 0$. All goal states g are assumed to be terminal, which forces to stay in g forever without accumulating further reward when taking this self-loop. Fixing these constraints makes sure that the self-loop in these states has no effect on the properties we are interested in later. Terminal states not contained in $\mathcal{S}_*$ are called *dead-ends*.

Definition 7: Paths in an MDP

For a given MDP $\mathcal{M}$ an infinite sequence of states connected via transitions, $\zeta = (s_i)_{i \in \mathbb{N}}$, is called a *path*. We often argue about finite paths τ which are finite prefixes of infinite paths.

$\zeta[0, i]$ denotes the finite prefix of ζ of length $i + 1$ from state s_0 to s_i. $\zeta[i]$ means the state s_i at position i in the path.

The length of a finite path τ is given by $|\tau|$. Analogously to the notation for infinite paths, $\tau[0, i]$ is the finite prefix of τ from s_0 to s_i and $\tau[i]$ denotes the state s_i of the path, both under the assumption that $|\tau| > i$.

$Paths(\mathcal{M})$ denotes the set of all infinite paths through $\mathcal{M}$ rooted in its initial state s_0. Accordingly, $Paths(s)$ is the set of all infinite paths starting in state s.

To get a more concrete impression on how the theory of Markov decision processes can be applied in practice, we give an illustrative example which we will use throughout the next sections to demonstrate how interesting properties can be calculated over MDP models of systems.

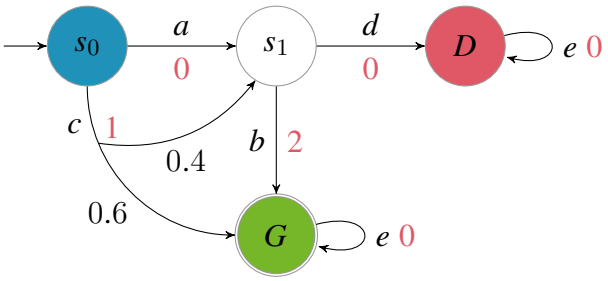
Example 1: Markov Decision Process

In this example we discuss the components an MDP consists of. The MDP below consists of four states, $\mathcal{S} = \{s_0, s_1, D, G\}$, where s_0 is the initial state, G is the goal state, i.e., $\mathcal{S}_* = \{G\}$, and D is a dead-end state. In addition, there are five actions, $\mathcal{A} = \{a, b, c, d, e\}$. a, b, d, and e are Dirac actions. c is a probabilistic action leading

with a probability of 60% to the goal state and with 40% to s_1. The states s_0 and s_1 are nondeterministic. It has to be decided nondeterministically which action to choose in s_0, where a and c could be taken, and in s_1, where b and d are applicable actions.

There are two paths leading from the initial state to the goal: $\tau_1 = s_0\ s_1\ G$ and $\tau_2 = s_0\ G$. τ_1 can be induced by taking action a in s_0 and b in s_1 or by taking action c in s_0 and b in s_1 if c lead to s_1. τ_2 can only be induced by taking action c in s_0 when ending directly in G. The last two options depend on the transition probability function.

Rewards are displayed in red. The reward obtained when taking action a, d, or e is 0. When taking b, a reward of 2 is achieved. For c a reward of 1 is obtained independently of the destination.



Some properties of MDPs or sets of specific states can already be precomputed on an abstracted version of the MDP without taking probabilities into account. This abstracted version is called the *underlying graph of the MDP*.

Definition 8: Underlying Graph of an MDP

A directed *graph* consists of a set V of *vertices* and a set E of directed *edges* of the form $(v_i, v_j) \in V \times V$ spanned between vertices. Two vertices v_i, v_j are called *connected* if there exists an edge (v_i, v_j) from v_i to v_j. The *underlying graph* $G = \langle V, E \rangle$ of an MDP $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, s_0, \mathcal{S}_* \rangle$ is spanned over $V := \mathcal{S}$ by the edge set $E := \{(s,t) \mid \exists a \in \mathcal{A} : \mathcal{T}(s,a,t) > 0\}$. The successor states of a vertice-edge pair, or more precisely of a state-action pair (s,a) in the graph, are given by $succ(s,a) := \{t \mid \mathcal{T}(s,a,t) > 0\}$.

Definition 9: Cyles & (Bottom) Strongly Connected Components

A *cycle* is a path in the underlying graph G of an MDP, i.e., a sequence of connected states, starting and ending in the same state. A *strongly connected component (SCC)* in G is a subset of states $V' \subseteq V$ such that $\forall (s,t) \in V' \times V'$ a path from s to t exists. A *bottom strongly connected component (BSCC)* B is an SCC of maximal size from which only states in B are reachable.

When reasoning about the behavior of systems modeled as MDPs, often concrete executions, described, e.g., by paths in the model, are considered. These paths are often induced by an entity resolving the nondeterministic choices in the MDP states. Resolving the nondeterminism in an MDP results in a Markov chain. Depending on the context, this entity deciding which action to apply in nondeterministic states is called *action policy*, *scheduler*, or *adversary*. The policy can be deterministic or may use *randomization* (picking a distribution over the applicable actions), and it may use the past *history* when picking, but does not have to. Histories are represented as finite sequences of states (i.e., sequences over S), thus they are drawn from S^+. We use $last(w)$ to denote the last state in $w \in S^+$.

Definition 10: Policy/Scheduler

For a given MDP $\mathcal{M}$ as above a function $\pi : S \to \mathcal{A}$ satisfying $\pi(s) \in \mathcal{A}(s)$ for each state s is called a (deterministic) *memory-less policy* or *scheduler*. The function determines the next action to take for any given state.
A (deterministic) *memory-full policy* or *scheduler* is a function $\pi\colon S^+ \to \mathcal{A}$ such that $\forall w \in S^+\colon \pi(w) \in \mathcal{A}(last(w))$. Memory-full is sometimes also called *history-dependent*.
In contrast, a (memory-less) *randomized policy* is a function $\pi : S \to \mathcal{D}(\mathcal{A})$, assigning probability 0 to actions $a \notin \mathcal{A}(s)$, which determines for each action a probability that it is taken in a specific state. Memory-full randomized policies are defined analogously.

Throughout the thesis, we will consider deterministic policies and thus present the theory in the following only for deterministic memory-full and memory-less policies. In later chapters, we mainly consider memory-less policies and explicitly state when memory-full policies are meant.

If a memory-less action policy π were instead taking a state sequence as input, it would satisfy $\pi(w) = \pi(w')$ whenever $last(w) = last(w')$. Hence, memory-less action policies are special cases of memory-full action policies.

If a policy is only given by a partial function, i.e., if not every state of S is assigned to an action by π, it is called a *partial policy*.

We say that a policy is *applied* on an MDP or in a state if the results of the policy function are used to determine which action to take next in every state. By starting in the initial state of an MDP, taking the action given by the current policy and continuing iteratively, a *path induced by the policy* is built through the MDP. With $Paths(\pi)$ we denote the set of all paths which can be built by applying policy π starting in the initial state of the MDP.

For memory-less policies the *subgraph G_π induced by policy π* in the underlying graph G of an MDP is obtained by restricting the edge set of G to $\{(s,t) \mid \mathcal{T}(s,\pi(s),t) > 0\}$. For

memory-full policies, the edge set of G is restricted to $\{(s,t) \mid \mathcal{T}(s,\pi(w),t) > 0\}$ over all finite paths w which start in the initial state s_0 of the MDP and end in state s.

An MDP $\mathcal{M}$ together with a deterministic memory-less action policy $\pi\colon \mathcal{S} \to \mathcal{A}$ induces a finite Markov chain $\langle \mathcal{S}, \mathcal{T}', s_0, \mathcal{S}_* \rangle$, where $\mathcal{T}'(s,t) := \mathcal{T}(s,\pi(s),t)$ for any $s,t \in \mathcal{S}$.

An MDP $\mathcal{M}$ together with a discrete memory-full action policy $\pi : \mathcal{S}^+ \to \mathcal{A}$ induces a countable-state Markov chain $\langle \mathcal{S}^+, \mathcal{T}', s_0, \mathcal{S}^+_* \rangle$ over state histories. With $\mathcal{T}'(w, ws) := \mathcal{T}(last(w), \pi(w), s)$ for any $w \in \mathcal{S}^+$ and successor state $s \in \mathcal{S}$. $\mathcal{S}^+_*$ consists of all $w \in \mathcal{S}^+$ with $last(w) \in \mathcal{S}_*$.

Definition 11: Cylinder Set of a Path

A *cylinder set* C of a finite path τ contains all infinite paths ζ which have τ as a prefix [22]. Formally, the set is specified by $C(\tau) := \{\zeta \in Paths(\mathcal{M}) \mid \tau \in pref(\zeta)\}$.

After having introduced all of this theory, we are ready to define a probability space on MDPs.

Definition 12: Probability Space on MDPs

A *probability space on an MDP* $\mathcal{M}$ can be defined as follows [22, 136]:
The sample space Ω consists of all possible infinite paths $Paths(\mathcal{M})$ in $\mathcal{M}$:
$\Omega = Paths(\mathcal{M})$.

The events Σ are obtained by a cylinder set construction over the finite paths in $\mathcal{M}$. Σ is the smallest σ-algebra over all cylinder sets of the finite paths in $\mathcal{M}$, which start in the initial state s_0: $\Sigma = \sigma(\{C(\tau) \mid \tau \in pref(Paths(\mathcal{M}))\})$.

Each of the sets T in Σ is thus characterized by one or more finite prefixes. These *characterizing prefixes* T_{fin} of the sets T in Σ are given by those shortest prefixes, characterizing the cylinder sets of which T consists, which are not prefixes of each other.

The probability function $\mathcal{P}^\pi : \Sigma \to [0,1]$ is defined w.r.t. to a deterministic memory-less policy π as follows. The probability for a set of infinite paths $T \in \Sigma$ when following policy π is defined as the sum over the characterizing prefixes of T given in T_{fin} of the product of the transition probabilities along them w.r.t. π:

$$\mathcal{P}^\pi(T) = \sum_{\tau \in T_{fin}} \prod_{i=0}^{|\tau|-1} \mathcal{T}(\tau[i], \pi(\tau[i]), \tau[i+1]).$$

For a deterministic memory-full policy π the probability function $\mathcal{P}^\pi(T)$ is defined similarly by $\sum_{\tau \in T_{fin}} \prod_{i=0}^{|\tau|-1} \mathcal{T}(\tau[i], \pi(\tau[0,i]), \tau[i+1])$.

Intuitively, Σ consists of the empty set, the set of all infinite paths $Paths(\mathcal{M})$, all cylinder sets of finite paths, and all possible countable unions of these cylinder sets as well as their complements. From the definition of the events, we can conclude that all sets of infinite paths of an MDP are measurable subsets of $Paths(\mathcal{M})$.

From the definition of the probability function on MDPs, we can conclude that the *probability that a specific finite path* τ is taken when following policy π (for memory-less and memory-full π), is given by the probability that any infinite path of the cylinder set $C(\tau)$ is taken. This probability can be calculated by a product of the probabilities of the transitions connecting the states of τ when following policy π, i.e., it is calculated by $\mathcal{P}^\pi(C(\tau)) = \mathcal{P}^\pi(\tau) = \prod_{i=0}^{|\tau|-1} \mathcal{T}(\tau[i], \pi(\tau[i]), \tau[i+1])$ for memory-less policies and by $\mathcal{P}^\pi(C(\tau)) = \mathcal{P}^\pi(\tau) = \prod_{i=0}^{|\tau|-1} \mathcal{T}(\tau[i], \pi(\tau[0,i]), \tau[i+1])$ for memory-full policies.

Beside calculating the probabilities of paths induced by policies in MDPs, we can also sum up the rewards of the actions taken to build the paths. This sum is called the *accumulated reward* of the path w.r.t. the policy.

Definition 13: Accumulated Reward for a Path Induced by a Policy

The *accumulated reward* $\mathcal{R}^\pi_{acc}$ over an infinite path ζ induced by a memory-less policy π through an MDP $\mathcal{M}$ is defined by $\mathcal{R}^\pi_{acc} : Paths(\mathcal{M}) \to \mathbb{R}$ where

$$\mathcal{R}^\pi_{acc}(\zeta) := \sum_{i=0} \mathcal{R}(\zeta[i], \pi(\zeta[i]), \zeta[i+1]).$$

For the finite prefixes τ of such a path the reward summation constituting $\mathcal{R}^\pi_{acc\text{-}fin}(\tau)$ can be calculated and truncated accordingly. This means, the *accumulated reward collected when traversing a finite path* τ is the sum of the rewards on each of the transitions taken, i.e.,

$$\mathcal{R}^\pi_{acc\text{-}fin}(\tau) := \sum_{i=0}^{|\tau|-1} \mathcal{R}(\tau[i], \pi(\tau[i]), \tau[i+1]).$$

Analogously, for a memory-full policy π we define:

- $\mathcal{R}^\pi_{acc}(\zeta) := \sum_{i=0} \mathcal{R}(\zeta[i], \pi(\zeta[0,i]), \zeta[i+1])$, for an infinite path ζ
- $\mathcal{R}^\pi_{acc\text{-}fin}(\tau) := \sum_{i=0}^{|\tau|-1} \mathcal{R}(\tau[i], \pi(\tau[0,i]), \tau[i+1])$, for a finite path τ.

Using the definition of the probability for a path induced by a specific policy π and the accumulated reward induced by π, the expected accumulated reward induced by this policy π can be defined.

Definition 14: Expected Accumulated Reward Induced by a Policy

The *expected accumulated reward* induced by policy π is defined over the accumulated reward function $\mathcal{R}^{\pi}_{acc} : Paths(\mathcal{M}) \to \mathbb{R}$, which in this case directly constitutes the random variable assigning to each path the reward obtained when taking this path by following π.

For a deterministic memory-less or memory-full policy π the expected accumulated reward is defined by

$$E(\mathcal{R}^{\pi}_{acc}) \coloneqq \sum_{\zeta \in Paths(\pi)} \mathcal{P}^{\pi}(\zeta) \cdot \mathcal{R}^{\pi}_{acc}(\zeta)$$

over the infinite paths induced by π.

Example 2: Probabilities and (Expected) Accumulated Rewards in MDPs

We consider again the MDP from Example 1. On path τ_2 a reward of 1 is accumulated. Because the path can only be induced by a policy selecting action c in s_0, the accumulated reward on the path is only the reward of this action. Under this policy, the path has a probability of 0.6. For path τ_1 the accumulated reward depends on the actions taken, i.e., on the policy which induces it. In case a and then b were taken, the accumulated reward is $0 + 2 = 2$. The probability for this path under that policy taking a in s_0 and b in s_1 is 1. If we have a policy choosing c in s_0 and b in s_1, the accumulated reward sums up to $1 + 2 = 3$ on the induced path. This path has a probability of 0.4.

The expected accumulated reward for a policy choosing c in s_0 and b in s_1 is $(0.4 \cdot (1 + 2)) + (0.6 \cdot 1) = 1.8$.

2.2. Model Checking and Verification

Markov decision processes (MDPs) are the base model for probabilistic model checking. With the probability measure over MDPs, several interesting properties of the MDP models of, e.g., cyber-physical systems, can be defined. For example, it might be of interest what the maximal probability for a certain failure in a power plant is, or what the minimal probability to complete a certain task within a predefined number of steps is for a robot in a production line. In addition, the expected time (i.e., number of steps) to complete a certain task using the modeled system, or the expected costs (e.g., consumed fuel in a car) to reach a certain goal may be of relevance.

Formulating these properties is often done by means of propositional *Linear Temporal Logic (LTL)* [224], *Computation Tree Logic (CTL)* [63], or *Probabilistic Computation Tree Logic (PCTL)* [126]. We only consider non-nested formulas to express properties in the following and therefore do not introduce the full logics.

The basic elements property descriptions consist of are *atomic propositions (AP)*. To reason about the behavior of MDPs, it is possible to annotate states with *state labels* which constitute of the set of atomic propositions holding in these states, as introduced for transition systems in Definition 4. In practice, atomic propositions are often variable values, for instance of the form $x = 1$.

To speak about execution paths of MDPs, i.e., sequences of states, and their properties, we define the notion of *traces*.

Definition 15: Trace of a Path

A *trace* t of an infinite path ζ is the infinite sequence of sets of the atomic propositions holding in the states of the path. This means, for the path ζ its trace is given by $t = trace(\zeta) := \mathrm{AP}(s_0)\ \mathrm{AP}(s_1)\ \mathrm{AP}(s_2)\ldots$, where $\mathrm{AP}(s)$ denotes the atomic propositions holding in state s.

In this thesis, we concentrate on *reachability properties*, i.e., properties describing conditions which have to be fulfilled by the atomic propositions at some point in the trace, i.e., in some state, called a goal state, while before reaching that state other conditions have to be satisfied in the trace. To express such reachability properties, atomic propositions can be combined to a logical formula ϕ using the logical operators *and* ($\wedge$), *or* ($\vee$), and *not* ($\neg$). In addition, we make use of the unary LTL operator $\Diamond\phi$, called *eventually*, which means that at some point a state is reached in which the formula ϕ is fulfilled. $\Diamond_{[l,u]}\phi$ means eventually in $[l, u]$ steps or with accumulated reward in these bounds a state fulfilling ϕ is reached, respectively. Note that not only closed intervals, but also (half) open intervals and other comparison operators can be used to express bounds. The eventually operator is a

special case of the binary *until* operator $\mathcal{U}$. In the form $\phi_1 \; \mathcal{U} \; \phi_2$ it states that ϕ_1 has to be fulfilled as long as no state is reached in which ϕ_2 holds, where ϕ_1 and ϕ_2 are again logical formulas over atomic propositions in the form defined above. For the satisfaction of the formula it is required that at some point a state fulfilling ϕ_2 is reached. $\Diamond\phi$ is an abbreviation for $\mathit{True} \; \mathcal{U} \; \phi$. The until operator can also be used with step or reward bounds in the form $\phi_1 \; \mathcal{U}_{[l,u]} \; \phi_2$, which means that within the given bounds on the number of steps or on the accumulated reward a state fulfilling ϕ_2 has to be reached and before that, ϕ_1 has to hold. Again, other variants for the description of bounds can be used.

The binary operator $s \models \mathit{Prop}$ is used to express that in a state s given on the left the property Prop on the right is satisfied. In a similar way, the operator is used to state that a path fulfills a property.

Reachability Properties w.r.t. a Policy. We first focus on how to calculate the *reachability probability* w.r.t. a certain policy, which is the probability with which a reachability property is fulfilled when following a given policy. Later, we discuss *(expected) accumulated rewards for reachability properties* under a certain policy, i.e., the (expected) reward collected until reaching a goal state when applying a given policy.

For *reachability probabilities* w.r.t. a policy, the probability to take one of the infinite paths in the MDP having a finite prefix fulfilling the property and reaching a goal state using the policy has to be calculated.

Hence, let $\mathit{Fin\text{-}Reach} := \{\tau \in \mathcal{S}^+ \mid |\tau| = n+1 \wedge s_n \in \mathcal{S}_* \wedge \forall k < n : s_k \notin \mathcal{S}_*\}$ be the set of minimal finite prefixes of paths eventually reaching a goal state, i.e., fulfilling $\Diamond\mathcal{S}_*$. Note that with this construction the cylinder sets of all paths in this set are stochastically independent, i.e., $\forall \tau \neq \tau' \in \mathit{Fin\text{-}Reach} : C(\tau) \cap C(\tau') = \emptyset$. This enables the summation over the probabilities for these cylinder sets to obtain the probability of their union.

Step- or reward-bounded reachability probabilities can be calculated in a similar fashion by considering the paths which reach the goal in the given number of steps or by gaining a reward in the given bounds, respectively, when following policy π. In more detail, the set of minimal finite prefixes of paths eventually reaching a goal state while accumulating a reward in the required bounds or by taking a number of transitions within the bounds has to be considered, respectively.

For rewards in the bounds $[l,u]$, the set is defined as follows: $\mathit{Fin\text{-}Bounded\text{-}Reach}_{[l,u]} := \{\tau \in \mathcal{S}^+ \mid |\tau| = n+1 \wedge s_n \in \mathcal{S}_* \wedge \mathcal{R}^{\pi}_{\mathit{acc\text{-}fin}}(\tau) \in [l,u] \wedge \forall k < n : s_k \notin \mathcal{S}_*\}$. For step bounds the constraint on the accumulated reward above is replaced by $n \in [l,u]$, meaning that the number of transitions taken until reaching s_n in the path lies in the bounds. Note that the bounds do not have to be given in a closed interval. (Half) open intervals and bounds specified

through comparison operators are also possible. There are also properties only specifying a lower or an upper bound but not both. The paths in those sets are also stochastically independent.

Based on these sets it is possible to define and calculate values for the respective property types.

- The *probability of eventually reaching states* in $\mathcal{S}_*$ starting in s_0 when following the memory-less or memory-full policy π, is the probability for the set of all infinite paths having a prefix in $\mathit{Fin\text{-}Reach}$. We use the following notation for it

$$\mathcal{P}^\pi(s_0 \models \Diamond \mathcal{S}_*) \coloneqq \mathcal{P}^\pi(\{\zeta \mid \exists\, \tau \in \mathit{pref}(\zeta) \wedge \tau \in \mathit{Fin\text{-}Reach}\}).$$

  This probability can be calculated by the sum of the probabilities of the paths in $\mathit{Fin\text{-}Reach}$ when following π:

$$\mathcal{P}^\pi(s_0 \models \Diamond \mathcal{S}_*) = \sum_{\tau \in \mathit{Fin\text{-}Reach}} \mathcal{P}^\pi(C(\tau)).$$

- The *probability of eventually reaching a state* in $\mathcal{S}_*$ *while accumulating a reward* in $[l, u]$ or taking a number of steps in $[l, u]$, respectively, when following policy π, is the probability for the set of all infinite paths having a prefix in $\mathit{Fin\text{-}Bounded\text{-}Reach}_{[l,u]}$. We use the following notation:

$$\mathcal{P}^\pi(s_0 \models \Diamond_{[l,u]} \mathcal{S}_*) \coloneqq \mathcal{P}^\pi(\{\zeta \mid \exists\, \tau \in \mathit{pref}(\zeta) \wedge \tau \in \mathit{Fin\text{-}Bounded\text{-}Reach}_{[l,u]}\}).$$

  This probability can be calculated in the following way:

$$\mathcal{P}^\pi(s_0 \models \Diamond_{[l,u]} \mathcal{S}_*) = \sum_{\tau \in \mathit{Fin\text{-}Bounded\text{-}Reach}_{[l,u]}} \mathcal{P}^\pi(C(\tau)).$$

- The *accumulated reward obtained on an infinite path ζ until reaching a state* in $\mathcal{S}_*$ when following policy π, denoted by $\mathcal{R}^\pi_{acc}(\zeta \models \Diamond \mathcal{S}_*)$ to make the property explicit, is obtained in the following way:

$$\mathcal{R}^\pi_{acc}(\zeta \models \Diamond \mathcal{S}_*) = \begin{cases} \mathcal{R}^\pi_{\mathit{acc\text{-}fin}}(\tau), & \text{if } \tau \in \mathit{pref}(\zeta) \cap \mathit{Fin\text{-}Reach} \\ \infty, & \text{if } \forall \tau \in \mathit{pref}(\zeta) : \tau \notin \mathit{Fin\text{-}Reach} \end{cases}.$$

  This means that, because no reward is accumulated anymore in goal states, the reward w.r.t. policy π of all paths in the cylinder set of a finite path τ, which ends in a goal state, is the same as for τ, i.e., $\forall \tau \in \mathit{Fin\text{-}Reach} : \forall \zeta \in C(\tau) : \mathcal{R}^\pi_{\mathit{acc\text{-}fin}}(\tau) = \mathcal{R}^\pi_{acc}(\zeta \models \Diamond \mathcal{S}_*)$.

If the goal cannot be reached along an infinite path, the accumulated reward until reaching the goal is defined to be ∞ [22].

- The *expected accumulated reward* collected when trying to reach a state in $\mathcal{S}_*$ following policy π starting in state s, $E(\mathcal{R}^{\pi}_{acc})(s \models \Diamond \mathcal{S}_*)$, is then defined as the expected value of the random variable which assigns each infinite path starting in s the reward collected when following it until reaching a goal state. This random variable is directly given by the accumulated reward function $\mathcal{R}^{\pi}_{acc} : Paths(\mathcal{M}) \to \mathbb{R}$. This means:

$$E(\mathcal{R}^{\pi}_{acc})(s_0 \models \Diamond \mathcal{S}_*) = \sum_{\tau \in Fin\text{-}Reach} \mathcal{P}^{\pi}(C(\tau)) \cdot \mathcal{R}^{\pi}_{acc\text{-}fin}(\tau)$$

if $\mathcal{P}^{\pi}(s_0 \models \Diamond \mathcal{S}_*) = 1$ and ∞ otherwise [22].

Note that, taking memory-less policies into account for bounded properties, will often not lead to optimal results, just because they are missing the relevant information about the current value of the accumulated reward or the number of steps left to decide how to proceed optimally.

Later, $E(\mathcal{R}^{\pi}_{acc})(.)$ is abbreviated by $ER^{\pi}(.)$.

For until properties $\phi_1 \; \mathcal{U} \; \phi_2$ we use the notation $\mathcal{S}_U \; \mathcal{U} \; \mathcal{S}_*$, where $\mathcal{S}_U$ is the set of states in which ϕ_1 is satisfied. The properties defined above can be calculated on these formulas in the same manner by taking the restriction on the states visited before reaching a goal state into account. In this case the sets $Fin\text{-}Reach$ and $Fin\text{-}Bounded\text{-}Reach_{[l,u]}$ are refined as follows: $Fin\text{-}Reach := \{\tau \in \mathcal{S}^+ \mid |\tau| = n+1 \wedge s_n \in \mathcal{S}_* \wedge \forall k < n : s_k \in \mathcal{S}_U \wedge s_k \notin \mathcal{S}_*\}$ and $Fin\text{-}Bounded\text{-}Reach_{[l,u]} := \{\tau \in \mathcal{S}^+ \mid |\tau| = n+1 \wedge s_n \in \mathcal{S}_* \wedge \mathcal{R}^{\pi}_{acc\text{-}fin}(\tau) \in [l,u] \wedge \forall k < n : s_k \in \mathcal{S}_U \wedge s_k \notin \mathcal{S}_*\}$.

So far, we only inspected reachability probabilities and expected rewards w.r.t. to a certain policy π, i.e., we basically only dealt with purely probabilistic and not with nondeterministic parts of models. If nondeterminism occurs in a model, the extremal, i.e., minimal and maximal, values for reachability probabilities or expected accumulated rewards are often of special interest, i.e., minimization or maximization of these measures over all possible policies has to be performed. In the following, we give an overview of the measures on MDPs used throughout the thesis.

Measures of Interest on MDPs. Typical properties of interest in this context include (maximal and minimal) reachability probabilities with respect to a set of goal states as well as (maximal and minimal) expected rewards (or costs) which are accumulated until reaching a goal state. These properties can also be subject to bounds on the number of steps until reaching a goal or bounds enforcing a certain reward (or cost) amount to be accumulated on the way to the goal.

We denote by $\mathcal{P}^\pi$ the probability measure on paths which are induced by policy π and use ER^π to denote the expectation of the accumulated reward with respect to sets of paths. We define the extremal values $\mathcal{P}_{\max}(\Gamma) = \sup_\pi \mathcal{P}^\pi(\Gamma)$, and $\mathcal{P}_{\min}(\Gamma) = \inf_\pi \mathcal{P}^\pi(\Gamma)$ as well as $ER_{\max}(\Gamma) = \sup_\pi ER^\pi(\Gamma)$, and $ER_{\min}(\Gamma) = \inf_\pi ER^\pi(\Gamma)$, for subsets $\Gamma \subseteq Paths(\mathcal{M})$. $\sup$ and $\inf$ denote the *supremum* and *infimum* over all possible policies, i.e., the smallest upper bound and the largest lower bound, respectively.

For the sake of brevity, instead of giving the concrete set of paths, we write $\mathcal{P}(F)$ for the probability to fulfill specification F starting from the initial state of the considered MDP.

We consider the following types of properties where $opt \in \{\max, \min\}$ (echoing what is supported in the JANI model format [53, 161], discussed in Section 3.1):

- *MaxProb* and *MinProb*: $\mathcal{P}_{opt}(\mathcal{S}_U \ \mathcal{U} \ \mathcal{S}_*) = \mathcal{P}_{opt}(\{\zeta \in Paths(\mathcal{M}) \mid \exists s \in \mathcal{S}_* : s = \zeta[j] \wedge \forall k < j : \zeta[k] \notin \mathcal{S}_* \wedge \zeta[k] \in \mathcal{S}_U\})$ is the maximal or minimal probability of eventually reaching a goal state, and all states visited before being in $\mathcal{S}_U$. $\mathcal{P}_{opt}(\mathcal{S} \ \mathcal{U} \ \mathcal{S}_*)$ will be abbreviated as $\mathcal{P}_{opt}(\Diamond \mathcal{S}_*)$.

- Minimal/Maximal expected rewards: $ER_{opt}(\mathcal{S}_U \ \mathcal{U} \ \mathcal{S}_*) = ER_{opt}(\{\zeta \in Paths(\mathcal{M}) \mid \exists s \in \mathcal{S}_* : s = \zeta[j] \wedge \forall k < j : \zeta[k] \notin \mathcal{S}_* \wedge \zeta[k] \in \mathcal{S}_U\})$ is the maximal or minimal reward expectation of eventually reaching a goal state, and all states visited before being in $\mathcal{S}_U$. Note that reward ∞ is accumulated for policies not inducing a goal reachability of 1 (see above).

- Step-bounded properties: $\mathcal{P}_{opt}(\mathcal{S}_U \ \mathcal{U}_{[l,u]} \ \mathcal{S}_*)$ is the maximal or minimal probability of reaching a goal state in $[l, u]$ steps defined as $\mathcal{P}_{opt}(\Gamma_{[l,u]})$ where $\Gamma_{[l,u]}$ is the set of paths that reach a goal state in $[l, u]$ steps while only passing through $\mathcal{S}_U$ before. Step bounded expected reward properties are defined in a similar fashion.

- Reward-bounded properties: If instead a reward structure is defined which can be used for the bounds, $\mathcal{P}_{opt}(\mathcal{S}_U \ \mathcal{U}_{[l,u]} \ \mathcal{S}_*)$ is the extremal probability of reaching a goal state with accumulated reward in $[l, u]$ defined as $\mathcal{P}_{opt}(\Gamma_{[l,u]})$ where $\Gamma_{[l,u]}$ is the set of paths that have a prefix τ reaching a goal state with accumulated reward in $[l, u]$, and only passing through $\mathcal{S}_U$ before. Remember again that we defined goal states to be terminal, which forces to stay in the goal state forever without accumulating further reward (see Definition 6). Reward-bounded expected reward properties are defined in a similar way. We also allow open intervals for the bounds.

In many input languages of model checkers it is possible to describe two categories of properties, *quantitative* and *qualitative properties*. Quantitative properties are those which have been discussed so far, for which the model checker has to calculate the numerical result (up to an error bound). For qualitative properties it has to be checked if the comparison

statement given to the model checker as a constraint on the quantitative part of the property is $True$ or $False$ [19, 127]. An exemplary qualitative property is $\mathcal{P}_{max}(\Diamond Goal) < 0.8$. To validate it, the model checker can calculate the probability of the quantitative part $\mathcal{P}_{max}(\Diamond Goal)$ of the property and compare it to the required value 0.8 to decide if the property holds. For certain properties, e.g., properties containing a comparison to 0 or 1, and in certain solution techniques efficient approaches exist which do not require to calculate the probability and compare it only afterwards but instead return the result by purely structural reasoning.

Example 3: Calculating Properties of MDPs

Taking up the MDP from Example 1, we can for instance ask for the maximal and minimal goal reachability probability $\mathcal{P}_{\min}(\Diamond G)$ and $\mathcal{P}_{\max}(\Diamond G)$ when starting in the initial state s_0.

To solve these properties, we have to reason about the goal reachability probabilities of all possible policies and search for the ones giving the infimum and supremum, respectively.

The policy which takes action a in s_0 and action d in s_1 leads to state D which can never be left, and hence, G cannot be reached. This leads to the minimal goal reachability probability of 0 for the MDP.

The maximal goal reachability probability can be achieved with two different policies. It is possible to take action a or c in s_0, and b in s_1. In both cases the goal is reached with certainty, i.e., $\mathcal{P}_{\max}(\Diamond G) = 1$.

To make the calculation of expected accumulated rewards a little bit simpler to start with, we relax the example by removing the action d and the state D. In this case, the goal is always reached with certainty, i.e., $\mathcal{P}_{\max}(\Diamond G) = \mathcal{P}_{\min}(\Diamond G) = 1$.

The properties of interest are the minimal and maximal expected accumulated reward when reaching the goal state, $ER_{\max}(\Diamond G)$ and $ER_{\min}(\Diamond G)$. When choosing a policy which takes action a in s_0 and then action b in s_1, we obtain a reward of 2, which is the maximal expected accumulated reward, since there is no other policy gaining a higher reward. The second possible policy choosing c in s_0 and b in s_1 has an expected accumulated reward of $0.6 \cdot 1 + 0.4 \cdot (1 + 2) = 1.8$, which is smaller than 2. Because there is no other possible policy, 1.8 is the minimal expected accumulated reward.

In the original example, where action d and state D are part of the MDP, the question about the maximal and minimal expected accumulated reward is more involved. It is basically a special case, because there is a policy which reaches a dead-end with probability 1, i.e., the reachability condition $\Diamond G$ is never fulfilled with this policy. Since for this policy the goal reachability probability is smaller than 1, the expected reward

is defined to be ∞. Therefore, the maximal expected accumulated reward is ∞. The minimal expected accumulated reward is induced by another policy, the one described above, and turns out to be 1.8.

All of this can be verified using the MODEST model in Appendix A.1 with MCSTA [119, 122, 129], the exhaustive model checker of the MODEST TOOLSET.

After having specified properties of interest over MDPs in the last sections, we will concentrate on different algorithmic solutions to calculate values for them in the following.

2.2.1. Exhaustive Probabilistic Model Checking

Model checking of probabilistic models (such as MDPs) nowadays comes in two flavors. *Probabilistic model checking* (PMC) [15, 17, 19, 22, 69, 90, 168, 188, 260] is an algorithmic technique to determine the extremal (maximal or minimal) probability (or expectation of accumulated reward) with which an MDP satisfies a certain property when ranging over all imaginable action policies. For some types of properties (step-bounded reachability, expected number of steps to reach) it does not suffice to restrict to memory-less policies, while for others (inevitability, step-unbounded reachability) it does. At the core, solution techniques of PMC are numerical algorithms that require the full state space to be available [127, 216]. That is the reason why standard PMC is often also called *exhaustive* PMC to differentiate it from other techniques on probabilistic models, which we discuss later. A basic, iterative technique to approximate the solution of these problems is called *value iteration*. It is often also the base for more sophisticated or specialized approaches.

Value Iteration. *Value iteration (VI)* [226] falls into the category of dynamic programming approaches. *Dynamic programming* [31, 35, 155] is used to solve optimality problems by dividing the problem into smaller parts, which have to be solved first, and the intermediate results produced are then used to approach the optimal solution for the entire problem incrementally by combining them step-by-step.

Value iteration is a variant of dynamic programming where a value is assigned to each state by a value function $V : \mathcal{S} \to \mathbb{R}$ which specifies the current approximation of the value of this state w.r.t. the property which is to be calculated. The value function is placed in an iterative procedure updating the states' values depending on the values of their successor states. Usually, the value function is calculated greedily via the *Bellman function* [32] (spelled out below for the calculation of the maximal value, but similar for minimum), which in full generality takes the reward values of actions into account.

$$V_{i+1}(s) = \max_{a \in \mathcal{A}(s)} \sum_{s' \in \mathcal{S}} \mathcal{P}(s, a, s') \cdot (\mathcal{R}(s, a, s') + V_i(s')) \tag{2.1}$$

The value function V_0, with which the iterative process starts, can be initialized arbitrarily in general, but depending on the property of interest, often at least the terminal states are initialized with 0 or 1. For the property types defined above, one would use an initial value of 0 for all states when calculating expected reward properties. When calculating reachability probabilities, an initial value of 1 for goal states and 0 for all others would be chosen, where rewards would be set to 0.

Basic value iteration operates on the full reachable part of the state space of the model. The values are refined until convergence to the least fixpoint, i.e., until all values have converged, which means further updates using the Bellman function do not change the values anymore. In many situations this fixpoint $V^* = \lim_{n\to\infty} V_n$ corresponds to the optimal value function one is looking for, from which the optimal policy can be extracted. In practice, convergence and thus termination of the value iteration process is checked up to an error bound ε, i.e., the value propagation is terminated as soon as no value changes by more than ε anymore perceivably. Algorithm 1 shows a detailed pseudo code of the standard value iteration procedure.

Algorithm 1: Value Iteration

1: **proc** ValueIteration(ε: float)
2: given: reachable state space $\mathcal{S}$ of an MDP and value function $V : S \to \mathbb{R}$ initialized for all states arbitrarily, except for terminal states (0 or 1 depending on which property type to calculate).
3: **repeat**
4: $\Delta = 0$
5: **for each** $s \in \mathcal{S}$ **do**
6: $V_{old}(s) = V(s)$
7: $V(s) = \max_{a \in \mathcal{A}(s)} \sum_{s' \in \mathcal{S}} \mathcal{P}(s, a, s') \cdot (\mathcal{R}(s, a, s') + V_{old}(s'))$
8: $\Delta = \max(\Delta, |V_{old}(s) - V(s)|)$
9: **until** $\Delta < \varepsilon$

Because of the way the error bound ε is used to determine when a fixpoint has been reached, the standard version of value iteration may not be sound, which means that even if convergence with the error bound ε is reached, the calculated result may still be arbitrarily far away from the actually correct value [226]. Luckily, such phenomena only occur in very specific cases.

However, there exist techniques based on linear programming with which it is possible to compute exact solutions for the properties [77, 189]. In addition, there are techniques like *interval iteration* which try to tackle the imprecisions of standard value iteration by calculating an upper and a lower bound for the final result instead of only approaching the result from

below [23, 48, 117]. Inspired by this technique, a *sound version of value iteration* [228] has been proposed, which is based on the idea of interval iteration approximating the value from above and below. In contrast to interval iteration, it does not need starting values for upper and lower bounds to be performant, but splits the reachability probability into the sum of the probability of reaching the goal within a certain number of steps and the probability of reaching it only after this number of steps.

There is also a refinement of the standard value iteration procedure, called *asynchronous value iteration* [34], in which only one state's value is chosen to be updated in each step. This variant is often used in planning and reinforcement learning, in which only a small fraction of the states is deemed sufficient to answer the considered property and not all states in the state space are considered. It often converges faster and is less memory-consuming than standard value iteration. The order in which values are updated is made dependent on their current value estimates. We make use of a variant of asynchronous value iteration in our adaption of algorithms presented in Chapter 4.

To speed-up the value iteration process for large models, it is possible to pre-compute exact results for some of the states instead of propagating their values, which can also avoid numerical rounding errors. This is possible with the help of a graph-based analysis for states from which it is not possible to reach any goal state at all and for states from which the goal is reached with absolute certainty [90]. The values of such states are then set and never changed anymore during the VI procedure, which thereby improves the overall precision of the final result.

Policy Iteration. A related approach to value iteration is *policy iteration* [155], where first an arbitrary policy defined on the reachable state space of the system under evaluation is picked with an arbitrary initialization V_0 of the value function, except for terminal states, like indicated for the value iteration procedure above. Then, policy evaluation and policy improvement steps alternate in each iteration. First, the current policy π is evaluated by calculating the new value for each state in the policy using the Bellman function $V_{i+1}(s) = \sum_{s' \in S} \mathcal{P}(s, \pi(s), s') \cdot (\mathcal{R}(s, \pi(s), s') + V_i(s'))$. If the maximal change in the value function for states on the currently selected policy is smaller than the error bound, a near-optimal policy has been found and the procedure can be stopped. Otherwise, the policy has to be improved by manipulating it in such a way that always the action leading to the state with the currently highest successor value is selected. If the policy does not change anymore in this step, the procedure can be stopped, too.

Techniques for Large State Spaces. To tackle the *state space explosion problem* [66, 67] occurring when trying to build the whole state space of a very large model for model checking

techniques like the ones discussed so far, various approaches to reduce, abstract, or compactly store the state space have been invented. For those techniques it is often possible to store the state space in such a way that model checking can still be executed with clearly quantifiable and acceptable error bounds, a requirement extremely important for the verification of critical infrastructure and cyber-physical systems in general. One of these approaches to compactly store states to allow efficient operations on them are *(multi-terminal) binary decision diagrams* ((MT)BDDs) [2, 76, 93, 140, 142, 143, 177, 186, 187, 208].

A different idea is implemented in the MODEST TOOLSET with the disk-based exploration and analysis engine of MCSTA usable to model check MDPs by partitioning the state space and outsourcing parts of it onto disk [129].

Another approach to reduce the treated state space are *bisimulation techniques* [207] or *minimization algorithms* [22]. They often require the full state space upfront to be able to reduce it, which does not reduce the memory consumption but only the verification time. If the system under study consists of a network of interacting automata, the reduction techniques can often be used component-wise before building the full network of interacting automata, which reduces the considered state space [21]. In contrast, partial order reduction [18, 101, 219] and confluence reduction [42, 253] construct bisimilar but not necessarily minimal models on-the-fly.

In addition, there are abstraction techniques which work on a coarser representation of the model by merging multiple states into one. An example is *counterexample guided abstraction refinement (CEGAR)* [65, 144].

Other approaches try to reduce the visited state space size on-the-fly by building only the parts of the state space necessary to calculate the current property. Such an algorithm, using heuristic search techniques from planning, will be presented in Chapter 4.

All of these techniques have in common that they more or less work very well on some model structures but can only achieve a small reduction on others [67, 176, 230].

2.2.2. Statistical Model Checking

A different model checking technique, in general only applicable to deterministic models, conceived to cope with the state space explosion problem, is *statistical model checking (SMC)* [28, 44, 51, 141, 194, 197, 198, 240, 273, 271, 275]. With this approach, the model under study is evaluated by simulating a certain number of sample executions of the system under study and evaluating it by performing hypothesis testing on the results to get a statistical evidence for inferring whether a property holds or what the value of it is. In general, this technique is not directly applicable to models containing nondeterminism, like MDPs, because during the simulation it is necessary to somehow resolve the nondeterminism to

proceed with the sample execution. But still, there are approaches applying SMC to MDPs, which we discuss at the end of this section.

The sample executions performed during SMC are often also called *traces* through the model. In the SMC context, the term should not be confused with a trace induced by a path in a model, as introduced in Definition 15.

At its core, SMC harvests classical Monte Carlo simulation [112, 211, 233]. In a nutshell, n finite samples of model executions are generated and evaluated to determine the fraction of executions satisfying a property under study or resulting in a certain value. This yields an estimate q' of the actual value q of the property, together with a statistical statement on the potential error. A typical guarantee is that $\mathcal{P}(|q' - q| > \varepsilon) < \delta$. This means, the probability that the estimate q' differs from the actual value q by more than ε is smaller than δ. $1 - \delta$ is called the *confidence* that the result is ε-correct. To decrease ε and δ, n must be increased. Essentially, this means that sequential hypothesis testing techniques are applied [266].

In the model checking settings considered in this thesis, SMC is a popular alternative to exhaustive probabilistic model checking, because PMC, requiring the full state space, is limited by the state space explosion problem, but SMC is not, even if the underlying model has infinite size. The approach only requires constant memory independent of the size of the state space because only a single trace is treated at once. Furthermore, SMC can be extended to non-Markovian formalisms or complex continuous dynamics effectively [47, 74]. When facing rare events, however, the number of samples needed to achieve sufficient confidence may explode.

In its original basic form, SMC as a simulation-based approach is not meant for nondeterministic processes [9, 57]. For these kinds of models, SMC is based on the idea that fixing a particular policy turns the MDP into a Markov chain which is easy to evaluate on its own. Therefore, in the MDP setting (or more complicated settings), SMC analysis is always bound to a particular action policy turning an otherwise nondeterministic model into a stochastic process. Nevertheless, there are some approaches for SMC on MDPs [44, 51, 78, 139], and many SMC tools support nondeterministic models, e.g., Prism [189], UPPAAL SMC [75], PAC [13], and PLASMA [163]. Often, they use an implicitly defined uniform random action policy to resolve choices, assign probabilities to the choices in some other form, or use Q-learning techniques like in the case of PAC. The statistical model checker modes [51], which is part of the Modest Toolset [128], lets the user choose out of a small set of predefined policies (also comprising a uniform random action policy), or provides lightweight support for iterating over policies to statistically approximate an optimal policy [51, 199]. More details on the SMC engine of the Modest Toolset are given in Section 3.4 and 5.1.1. In addition, variants of so called sound SMC, which are able to treat nondeterminism in MDPs,

have been discussed [43, 133, 134]. In most of these cases, results obtained on MDPs by SMC are to be interpreted relative to the implicitly or explicitly defined action policy.

The full discussion of this issue is not of greater relevance for this thesis because our *Deep Statistical Model Checking* approach introducing a new technique based on SMC to assess the quality of trained decision-making agents, like neural networks, acting in MDP environments as a determinizer, exploits exactly the fact that the nondeterminism is resolved prior to applying SMC on the resulting Markov chain.

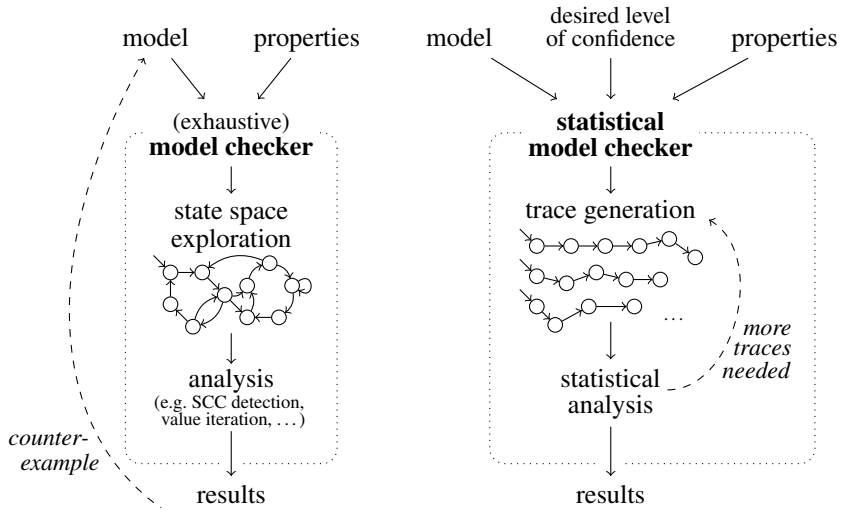


Figure 1.: Principle of exhaustive and statistical model checking [127].

Figure 1 visualizes the functionality principles of exhaustive model checking and statistical model checking. As discussed in detail in Section 2.2.1, an exhaustive model checker explores and analyzes the whole state space of a model to check a given property. This can be done with the help of different techniques, like standard value iteration or more involved symbolic model checking techniques. For qualitative model checking, it is then able to certainly state that the property is fulfilled or a counter-example can be provided which proves that the property is violated. For quantitative model checking, it is able to provide, e.g., a probability or an expected accumulated reward for a property. This result can be exact or lie in a predefined error range, depending on the concrete PMC algorithm.

In contrast, statistical model checking can only deliver results for qualitative and quantitative properties up to a predefined confidence level by statistically evaluating sample executions, i.e., traces, through the model. To reach the desired level of confidence in the statistical analysis the number of sample traces has to be increased until the result has the desired quality. If a nondeterministic model is given to the SMC solver, an entity providing a policy giving information on how to resolve the nondeterminism is required.

There are comparative studies between statistical and exhaustive model checking [169, 271] but it is a controversially discussed question if the approaches can really be compared because of their quite different handling of nondeterminism. But as already mentioned, this discussion is not of greater importance for the thesis.

2.3. Probabilistic Planning & Heuristic Search

The goal of planning approaches is to find an optimal path w.r.t. a certain specification through a system. For example, a robot driving around in a factory is supposed to find the shortest path to grab a specific item. Planning can be divided into the fields of *path planning*, like in the previous example, but there is also *motion planning*, *scheduling*, and *task planning* [236]. The environments where the agents executing the plans act in can be deterministic but it is also possible that they contain nondeterministic or probabilistic behavior. Hence, the planning environments can also be modeled as MDPs, although they are classically defined over a set of *facts*. Usually, planning tasks consist of a set of facts which do initially hold. When taking an action, facts are deleted or added until a goal is reached, which is again defined by a set of facts which have to be fulfilled [236]. Often specific functions are applied guiding the search for the plans faster to the goal by giving additional information or estimations. Those functions are called *heuristics*.

Although the objectives of planning and model checking are different, and therefore also the problem formulations and solution approaches differ, there are close connections between *probabilistic planning* [184, 203, 270] and probabilistic model checking, which can and should be exploited to get the best solution methods inspired by both worlds. So far, there were only a few works that started to bridge the gap between planning and model checking, e.g., [29, 48, 247, 251]. But with translations between the standard language for describing probabilistic planning domains, the *Probabilistic Planning Domain Definition Language (PPDDL)* [272], and JANI [53], an overarching format for probabilistic models, as well as with a detailed, extensive comparison of different planning and model checking approaches [175, 176], the author of this thesis layed the foundations for more cross-fertilization between both communities, and for interchanging and combining algorithmic approaches. One of these approaches to use planning techniques for model checking is presented in Chapter 4.

Most of the theory discussed so far is commonly used in both, probabilistic planning with heuristic search and model checking. But there are additional terms and concepts from the probabilistic planning context not discussed so far, which are introduced in the following.

Variants of value iteration methods are not only used in model checking but also in the probabilistic planning context. For some of those variants it is crucial for the functionality that the value function, especially its initialization, is *admissible*.

Definition 16: Admissibility

A value function V is called *admissible* if it provides an optimistic estimate of the correct final optimal value of V^* w.r.t. the property type under investigation. This means, if we try to minimize, the value function V is admissible if $V(s) \leq V^*(s)$ for all $s \in \mathcal{S}$. If we instead maximize, a value function with $V(s) \geq V^*(s)$ for all $s \in \mathcal{S}$ is admissible.

While in probabilistic model checking MDPs often reflect some sort of concurrency phenomena, they also have a longer tradition in the context of *sequential decision-making under uncertainty* [35, 160]. Depending on these differences in the modeling context, MDPs are usually considered decorated with rewards in model checking, as in Definition 6, but with *costs* in planning. The term reward is traditionally used if the goal is to maximize the earnings. In the dual context of costs, the spendings are usually to be minimized. All of this is done under the assumption that the decisions in the MDP are controllable. Instead, and in particular in a setting where the MDP results from concurrent interleavings, it can also be natural to ask for the maximal cost lurking or the minimal reward obtainable, since in this case the decisions need to be assumed uncontrollable and the *worst case scenario* is of interest.

As discussed above, model checking tools often rely on value iteration techniques, which are limited by memory for the explicit exploration of the whole state space. The support of complex temporal properties often requires to consider, and to actually reconstruct, the entire state space of the model from the compact input description. A considerable portion of work was therefore spent on developing compact representation methods that efficiently support operations required for the analysis of the given properties, e.g., (multi-terminal) binary decision diagrams (MT)BDDs [2, 76, 93, 140, 142, 143, 177, 186, 187, 208]. In contrast, probabilistic planners focus on plain reachability analysis variants and can therefore make use of different approaches, called *heuristic search* algorithms [27, 45, 46, 125], which use their current knowledge of the value function or other information about the state space in order to disregard parts of the MDP's state space that provably cannot be part of an optimal solution. A central idea underlying many of the algorithms invented in this context is the incorporation of additional information, automatically extracted from the compact model description. This information is exploited in order to take only a small part of the whole state space into consideration, which is deemed to be relevant to answer the desired objective.

Taking the basic value iteration for value maximization (see Algorithm 1) as an example, a heuristic search algorithm can for instance be based on the observation that for computing

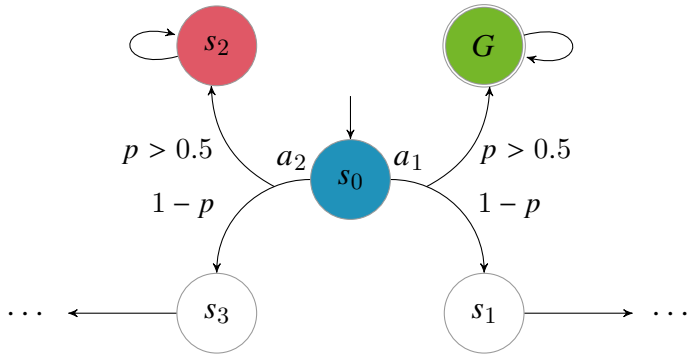
the maximal goal reachability probability for just the initial state, $V^*(s_0)$, it is not necessarily required to consider and to compute V^* for all states reachable from s_0. An example is given in the following.

Example 4: Pruning the State Space with Heuristic Search

As a simple example, we consider the snippet of an MDP below. State s_0 has two applicable actions: (1) a_1 leading to a goal state G with probability $p > 0.5$ and to another state s_1 with the remaining probability $1 - p$, and (2) a_2 leading to a terminal non-goal state s_2 with the same probability p and to some state s_3 with the remaining probability. We do not have more information about the part of the MDP reachable from s_1 and s_3.

Assuming we want to maximize the probability of reaching the goal, a_2 can never be the action selected in the maximization part of the Bellman Equation 2.1 for s_0, because the goal reachability probability is smaller than 0.5 and a_1 leads with a probability of more than 0.5 to the goal.

In particular, we do not need to know the value of s_3 to prove that a_2 cannot be chosen in s_0 by any optimal policy. At the same time, s_3 might be rooting a large subgraph of the state space. This entire subgraph could however be ignored, since its consideration is only required for the computation of $V^*(s_3)$.



To make already the search's initial value function estimates more accurate, and thus to improve the overall efficiency, one can additionally exploit *heuristics*, functions that provide per-state approximations of the optimal result. In general, a heuristic is a function which tries to estimate the optimal value of a state w.r.t. the problem under consideration as close as possible such that it is not necessary anymore to consider certain paths or areas of the state space in the expensive straight forward solving method. MDP heuristic search methods [27, 45, 46, 125, 180, 205, 246] try to shrink the visited state space by using the help of a heuristic function when interleaving the partial state space exploration with the value computation and propagation.

The use of a heuristic function is demonstrated in the next example.

Example 5: Application of a Heuristic Function

In a slightly modified version of the MDP of Example 4 a heuristic function could for instance provide estimates of the states' values. Let us assume that s_2 is not terminal, like s_1 and s_3. In this case, it is no longer possible to exclude a_2 from consideration right away, since now an optimal solution could potentially pass through s_2. On the other hand, assume that we are also given an admissible, i.e., optimistic, approximation of V^*, in the form of an efficient to compute heuristic function $h : \mathcal{S} \to \mathbb{R}$ such that $h(s) \geq V^*(s)$ for all states s.

In this case, h can be used to provide a necessary condition for the probabilistic transition a_2 to be potentially part of an optimal solution. This condition is $p \cdot h(s_2) + (1-p) \cdot h(s_3) > p$. This is the case, because h overapproximates the goal reachability probabilities for the states. If already this overapproximation, in this example for state s_2 and s_3, leads to a smaller or equal value for s_0 when taking a_2, than what is achieved for sure when taking a_1, the latter action has to be taken for optimality.

When calculating extremal values for properties, often optimal policies according to a certain measure have to be found by building paths through the state space optimizing from one state to the next. This procedure can be described as being *greedy*.

Definition 17: Greedy Policy & Greedy Graph

A *greedy policy* is always defined w.r.t. a value function V. For each state the greedy policy always picks the action leading to the successor state(s) (multiple states in case of probabilistic actions) with the best value(s) according to the value function. This action may not be unique, e.g., because two successors have the same value, which means that there can be multiple possible greedy policies.

A *greedy graph* G_V of graph G of an MDP (see Definition 8) with respect to the value function V is the superposition of all graphs G_π induced by any greedy policy π w.r.t. V. This means, it is the combined reachability graph of all greedy policies.

Definition 18: Traps

A *trap* [179, p. 171 ff.] in a graph is a BSCC (see Definition 9) not containing a goal state. In our approaches *traps are defined on the greedy graph* G_V induced on the graph G of an MDP by the value function V.

We distinguish between two types of traps. There are *permanent traps* of G_V which are also BSCCs of G and thereby traps of G, i.e., there is no non-greedy policy which would lead out of the trap of G_V when considering the full graph G instead.

In contrast, *transient traps* of G_V are SCCs of G, but not BSCCs, and therefore not traps of G, i.e., there is a policy in G which is non-optimal under the current value function V leading out of the trap of G_V.

The planning literature has identified a number of model classes with convenient properties on which quite a lot of algorithms are based and rely. Initially, some of these approaches try to cope with arbitrary rewards in $\mathbb{R}$, which in the end is often restricted to only some property types or narrowed to positive rewards. The supported range of rewards, property types, and model types is therefore quite fragmented across multiple approaches. Since in Chapter 4 we will extend one of these procedures to work on MDP models with positive and zero-valued rewards on all established property types, except long-run averages and nested properties, which is much more than the original supported properties, we briefly introduce notions needed to talk about characteristics of these originally considered model classes and properties in the following.

Definition 19: Almost-Sure & Proper Policies

π is an *almost-sure* policy for an MDP if the probability of reaching $\mathcal{S}_*$ it induces is 1 regardless of the initial state. If, on the other hand, that probability is only guaranteed to be positive, π is called a *proper* policy.

Definition 20: Stochastic Shortest Path (SSP) MDP

A *Stochastic Shortest Path (SSP)* MDP [35] is an MDP admitting (i) at least one almost-sure policy and (ii) inducing expected accumulated reward ∞ for each policy π which is not almost-sure.

The latter corresponds to G_π containing no reachable cycle on which (in the MDP) the accumulated reward does not increase. Assuming the former, the latter can trivially be enforced by restricting to models with reward function $\mathcal{R}$ confined to positive values (possibly except at goal states).

As an apparent relaxation, Bertsekas [36] later introduced *conditions* (i′) and (ii′) which replace the role of *almost-sure* policies by *proper* policies in (i), respectively (ii), but showed them to be (pairwise) equivalent.

In addition, the class of SSPs has been further refined in the planning community by defining *Generalized Stochastic Shortest Path* problems.

Definition 21: Generalized Stochastic Shortest Path (GSSP) MDP

In a *Generalized Stochastic Shortest Path (GSSP)* MDP [179] the first condition (i) of SSPs is kept while the second condition (ii′) from above is further relaxed by instead assuming that (ii″) for each policy π and state s the expected sum of negative rewards is bounded from below.

This relaxation in particular allows for zero-valued reward cycles, while it precludes cycles with alternations of positive and negative rewards that cancel out each other. The latter can trivially be enforced by restricting to models with reward function $\mathcal{R}$ confined to non-negative values, as we do in our definition of MDPs (see Definition 6). Our contribution in Chapter 4 relinquishes condition (i) and (i′) of SSP and GSSP, i.e., our algorithmic contributions do not rely on the existence of almost-sure or proper policies.

2.4. Neural Networks and Q-Learning

Since we present an approach to assess the quality of neural networks and automated decision-making agents in general later in this thesis in Chapter 5, a brief overview of the required neural network learning context is given in the following.

Neural networks (NN), in particular *deep neural networks (DNN)*, promise astounding advances across a manifold of computing applications in domains as diverse as image classification [182], natural language processing [146], and game playing [244]. NNs are the technical core of ever more *intelligent systems*, created to assist or replace humans in decision-making. They have recently been applied with dramatic successes to learning of action policies in large transition systems, from Atari games [209] over Go and Chess [244] to Rubik's cube [1]. From the examples we listed here and in the introduction (Chapter 1), it is clear that NNs play a key role in action decisions of many autonomous systems already, which will become even more in the future. In particular, this pertains to action decisions in environments which can be formalized as MDPs.

NNs consist of *neurons*, which are atomic computational units that typically apply a non-linear function, their *activation function*, to a weighted sum of their inputs [238]. For example, a neuron can use the activation function $f(x) = \max(0, x)$, which is called the *rectified linear unit (ReLu)* activation function. We consider relatively simple *fully-connected feed-forward NNs* as depicted in Figure 2, a classical architecture where neurons are arranged in a sequence of *layers*, each consisting of a specific number of neurons. The first layer is called *input layer* and the last layer is the *output layer*. All layers in between are called *hidden layers*. Inputs are provided to the neurons $i_1, \ldots, i_n$ of the input layer, and the computation

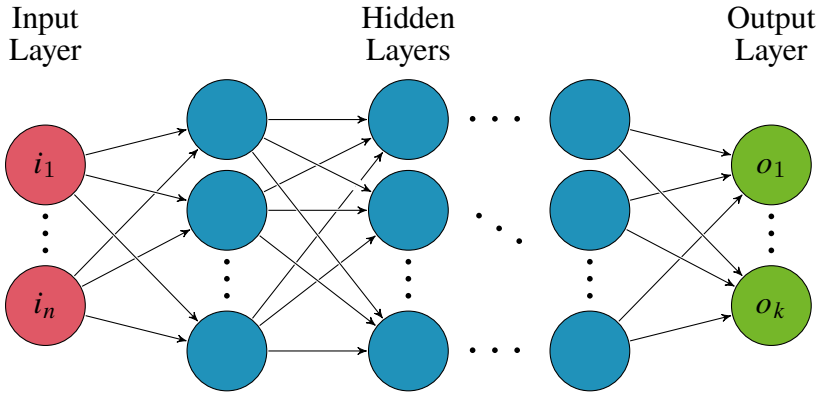


Figure 2.: Structure of a feed-forward neural network.

results of the neurons in each layer are propagated as inputs to all neurons of the next layer in the sequence until reaching the output layer, consisting of neurons $o_1, \ldots o_k$. This means, the values are propagated forward through the layers (called feed-forward) and all neurons of a layer are connected via input/output to all neurons of the layer in front and behind (called fully-connected). For a given set of possible inputs $\mathcal{I}$ and (final layer) outputs $\mathcal{O}$, a neural network can be considered as an efficient-to-query total function $nn\colon \mathcal{I} \to \mathcal{O}$. In every layer, every neuron receives as inputs the *weighted* outputs of all neurons in the previous layer, possibly together with an additional value, called a *bias*, to manipulate the function in a certain way. During the learning process of a neural network, adequate values for these weights and biases have to be found such that the outputs fulfill a certain quality requirement w.r.t. the task the NN should be trained for.

Feed-forward NNs are comparatively simple, yet they are in widespread use [97], and are in principle able to approximate any function to any desired degree of accuracy [152]. Deep neural networks consist of many layers. In tasks such as image recognition, successful NN architectures have become quite sophisticated, involving, e.g., convolutional layers taking multi-dimensional inputs and max-pooling layers discarding superfluous information [182].

NNs are often trained for the usage as agents in environments where they should make decision in such a way that a predefined goal is reached while taking care of some optimality criterion, e.g., operating as fast as possible or by consuming as few resources as possible. For this purpose, the NN is provided with a state description of the environment to identify the next action to take to reach the predefined goal. When propagating a description of the current state of the environment as input trough the NN, the values of the neurons in the output layer are classified, i.e., interpreted, in such a way that a decision of the agent in the context of the environment can be concluded. These decisions are called *classifier outputs*.

NNs can be trained in a multitude of ways. In our setting, decision-making agents are trained by iterative execution and refinement steps. One such execution and refinement step is called *training episode*. This means the decision-making agent is taught via trial and error. The current quality of the NN decisions w.r.t. the predefined goal is measured by the discounted sum of rewards received for achieving certain (partial) goals while interacting in the training environment. The final discounted sum of rewards at the end of each training episode is called *training or learning return*. Formally, the training return is calculated by:

$$G_t = \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k, \tag{2.2}$$

which is the accumulated discounted reward from time t on, where R_i is the random variable giving the reward obtained in step i for certain task specific achievements, $\gamma \in [0, 1]$ is a discount factor, and T is the final time step [249]. To define a policy of good quality, each learning episode executes the current version of the NN from some state of the environment on, and updates the NN weights and biases using gradient descent optimization [235] w.r.t. to the training return achieved. The goal during training is always to achieve a certain task by getting the highest return.

We use *deep Q-learning* [209], a successful and nowadays widespread form of reinforcement learning [91, 166, 236, 259]. When taking decisions in the environment, the NN approximates per possible action the expected discounted accumulated return to fulfill the required task in the environment, the so-called *Q-value*, that will be achieved when deciding for an action and following the decisions afterwards. An arg-max function on the Q-values is used to determine the next action to take. Deep Q-learning has been shown to learn high-quality NN action policies in a variety of challenging decision-making problems [111, 209, 243, 244, 263].

The area of neural network learning which is closest to our model checking approaches, and which is based on environments which can be modeled as Markov decision processes, is *model-based reinforcement learning* [167, 210, 249]. Tasks to be fulfilled by a trained decision-making agent in model-based learning are often very similar to the model checking properties, e.g., in *safe reinforcement learning* [95], where policies are learned which even during the learning process always respect safety or performance requirements. Since the model checking properties and learning tasks are so close in model-based learning, the reward function of the MDP model of the environment can directly be used as the reward function for the learning process.

We instead make use of a *model-free reinforcement learning* approach [55], which in contrast does not directly make use of the full MDP description, e.g., the transition probabilities, the reward function, the exact state space information, and the action structure

of a formal model of the environment, but uses an approach to train an agent with only parts of these information, modified versions thereof, or by using additional external information. With this technique an agent is trained to make decisions in an environment modeled often quite similar to an MDP but optimized for the learning process, e.g., w.r.t. the reward function used.

Reward structures in MDPs specify numerical rewards to be accumulated when transitioning along state sequences, i.e., $\mathcal{R} : \mathcal{S} \times A \times \mathcal{S} \to \mathbb{R}$ (see Definition 6 where we assume $\mathbb{R}_{\geq 0}$). In the model checking problems for which agents are trained in this thesis, we are instead interested in the probability of property satisfaction independent of the reward function, i.e., the problem specification of learning, which requires a reward function to measure the quality of training, and the model checking problem do not match exactly. Hence, rewards independent from the MDP used for model checking appear during the model-free reinforcement learning approach as part of the NN training aiming at optimizing the training return.

3. Context of Benchmarks, Models, and Tool Implementations

Markov models in general, and especially Markov decision processes, are suitable to model the behavior of many real world cyber-physical systems and technical processes. Thus, as discussed above, not only the model checking community uses this model family, but, e.g., also the planning community often considers MDPs to model and solve their problems. This shows that in different contexts Markov models are of interest. Various MDP benchmarks have been created, which are, e.g., collected in the Prism benchmark suite [190], the Quantitative Verification Benchmark Set [132], and the benchmark set of the International Probabilistic Planning Competition [272]. In their original versions, they are often modeled in context specific languages and are thus not even accessible to the whole community they stem from, not to mention other communities potentially also interested in these model types.

To improve on that, and to foster cross-fertilization between communities, we use JANI models [53] throughout this thesis which are freely available. Since this thesis is centered around model checking approaches for systems whose functionality can be modeled as MDPs, a large MDP benchmark set comprising examples with different foci from various areas and communities is needed to evaluate the resulting tools in multiple contexts. The benchmarks used throughout the thesis and their origins are described in this chapter. Most of them are part of the Quantitative Verification Benchmark Set (QVBS) [132], whose origins, content, and functionality is discussed in the following. These models also form the basis of QComp [54, 132], the quantitative verification competition, which we briefly present in the following. In Chapter 4, we evaluate one of our tool contributions, Modysh, in this competition's setting. In addition, we introduce the Racetrack benchmark in its basic form, and discuss the variants of it used in many of the authors's publications, especially also in the works on Deep Statistical Model Checking in Chapter 5.

All our tool contributions evaluated on these benchmarks are implemented in one common infrastructure, the Modest Toolset. We give a summary of the functionalities available in the toolset and of its architecture. Thereby, we focus especially on the tools we used for our purposes and on the tools we extended.

The chapter does not belong to the deep core of the thesis but nevertheless contains genuine contributions of value for the communities using MDP models.
In summary, the contributions relate to:

- The development of the Quantitative Verification Benchmark Set.
- The first and second edition of the QComp competition.
- The development of multiple variants of the Racetrack benchmark and infrastructure for Racetrack benchmarking, including parsing maps, JANI model generation, and model export.

Information about the QVBS and the QComp competition can be found on their website[1]. The artifact providing the technical infrastructure, the tooling, and the results of the second edition of QComp in 2020 is archived on Zenodo[2]. All variants of and tool infrastructure centered around Racetrack can be found online[3]. Among these materials is also a generator for JANI Racetrack benchmarks with different parameters. Detailed information on the Modest Toolset is available on its website[4].

Organization and Origins of the Chapter. Section 3.1 introduces the JANI model format in which all of the models used in the thesis are specified. Section 3.2 introduces the Quantitative Verification Benchmark Set and the QComp competition based on it. In Section 3.3 we present the Racetrack benchmark and its variants. The functionality and architecture of the Modest Toolset is discussed in Section 3.4.

The JANI model format is clearly not a contribution of the author but necessary to be aware of before talking about concrete models used throughout the thesis. Similarly, the information provided on the Modest Toolset is necessary to understand the context of the tool contributions of the thesis. The Modest Toolset is maintained and has mainly been implemented by Arnd Hartmanns with the help of several co-authors, like the author of this thesis who implemented all tool contributions presented in later chapters in the environment of the toolset.

Together with Arnd Hartmanns and Tim Quatmann, the author was part of the core team developing the QVBS, which has been presented in a TACAS 2019 tool demonstration paper [132]. The benchmark collection as well as the QComp competition based on it was, beside other influencing factors, inspired by previous works of the author in the planning

[1] https://qcomp.org/
[2] http://doi.org/10.5281/zenodo.3965312
[3] https://racetrack.perspicuous-computing.science/
[4] https://www.modestchecker.net/

community [175, 176]. The author submitted 50 of the currently 78 benchmarks to the QVBS and thus considerably contributed to it. In addition, the author together with Arnd Hartmanns and Tim Quatmann designed and organized the first QComp for the TOOLympics 2019 at TACAS [121], and was responsible for the technical setup, tool execution, and result evaluation of the second edition at ISoLA 2020 [54, 131].

The Racetrack model has been part of many works of the author, and has been refined and extended with new features in several of them. The initial JANI model of Racetrack has been built for the first works on DSMC by the author with the help of Marcel Steinmetz [106], which has later been discretized in a finer manner by the author for the scalability study of DSMC in a journal article [109]. An extended version of Racetrack with new features, like fuel consumption and different engine types, has been implemented by the author of the thesis for the evaluation of suitability notions of components in probabilistic systems published at ISoLA 2020 [20]. A summary of all Racetrack variants used as a laboratory for research on perspicuous automated decisions and the tools developed in this context is published in a TAILOR 2020 paper [16], which was co-authored by the author of this thesis, and which is accompanied by a website[5] on the works centered around Racetrack coordinated by the author.

3.1. Probabilistic Models in JANI

Most model checkers have their own, tool-specific input language, like the Prism language [190] of the Prism model checker [189], or Modest [123] of the Modest Toolset [128]. This makes it notoriously hard to exchange models, e.g., for the purpose of comparing performance of different tools with similar functionalities or just to exchange interesting use cases. The *JANI-model format* [53] has been invented to solve this issue. It is an overarching format conceived to foster verification tool interoperation and comparability. JANI is targeted at establishing a common input format for probabilistic model checkers. Apart from others, it is directly supported as an input format by the state-of-the-art model checkers the Modest Toolset, Storm [77], and ePMC [92, 124]. Translations from and to the Prism language exist, too [77, 124]. In full generality, JANI models are networks of *stochastic hybrid automata (SHA)*. But the core formalism are MDPs. Most of the JANI models belong to one of the following types: *Markov automata (MA)*, *timed automata (TA)*, *probabilistic timed automata (PTA)*, *continuous-time MDPs (CTMDPs)* as well as *discrete-*, and *continuous-time Markov chains (DTMCs/CTMCs)*. JANI allows users to model a rich variety of distributed and concurrent systems in the form of quantitative automata networks with variable decorations, clocks, and probabilities. JANI, as a JSON based format, is designed with a particular focus on

[5] https://racetrack.perspicuous-computing.science/

extensibility and machine readability. Extensibility simplifies the integration of new features, not part of the JANI core already. Machine readability makes it easier to add JANI support to existing tools. In addition, it is a human-readable, but not necessarily a human-writable format. To write a model by hand, it is possible to select from other more concise modeling languages, like the MODEST or the PRISM language, and later translate that to JANI. With Momba [178] the whole tool-ecosystem of Python functionality is now also accessible to build and work with JANI models. The author of this thesis has partially been involved in the developments around Momba.

In addition, language features of JANI are of considerable interest to the AI community. For example, the thesis' author was part of a team showing that JANI is much more general and capable than the *Probabilistic Planning Domain Definition Language (PPDDL)* [272] even for describing probabilistic planning tasks [148], which shows that the language of the model checking community has benefits when used on the planning side. With the compilations between PPDDL and JANI [175, 176] the author together with others paved the way for future connections and research in this direction.

A JANI model consists of three main components: 1) a list of *global variables*, 2) an *automata network*, i.e., a set of individual automaton specifications of interacting automata with variables, and 3) the *property* to be checked. The format supports the specification of variables of many different types, including discrete, bounded integers and reals, continuous variables, and clock variables. This enables the representation of many different kinds of probabilistic systems. Expressions over these variables can be composed of all standard arithmetic operations as well as conjunction and disjunction. An automaton is specified through a set of *local variables*, and a set of *locations* connected by directed *edges*, which can be labeled with *edge labels*. Each edge defines a single source location, a *guard*, i.e., a condition that must be satisfied to apply the edge, and either a single *destination* or a discrete probability distribution over multiple destination locations. Beside the target location, each destination additionally carries a list of *assignments* to global and local variables that apply atomically whenever going to the respective destination.

To give a formal semantics to the overall automata network, JANI requires the definition of the *system composition*. The *transitions* of the network are then obtained by synchronizing the automata, i.e., in every transition, potentially multiple automata participate with one edge, respectively. In the simplest case, in each step of the overall system, exactly one applicable automaton edge is executed nondeterministically. However, JANI also supports the specification of *synchronization vectors*, allowing the parallel or synchronized execution of edges of multiple automata based on action labels. JANI allows the definition of one or multiple initial states by putting constraints on the global and local variables.

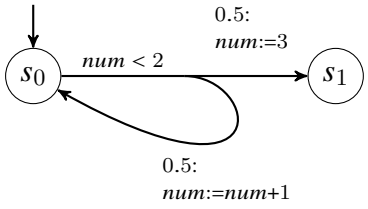
Properties to be checked are temporal formulas based on computation tree logic (CTL) [63]. The property description in CTL-style may follow different schemes, depending on the exact type of the quantitative measure to consider. More details on the JANI syntax can be found in its specification[6] [161].

An exemplary snippet of a JANI model can be inspected in the following.

Example 6: A JANI Automaton

```
"variables": [
  { "name": "num",
    "type": { "kind": "bounded", "base": "int",
              "lower-bound": 0, "upper-bound": 3} } ],
"restrict-initial": { "exp": "=", "left": "num", "right": 0 },
"automata": [
  { "name": "aut",
    "locations": [ {"name": "s0"}, {"name": "s1"} ],
    "initial-locations": ["s0"],
    "edges": [ {
      "location": "s0",
      "guard": { "exp": { "op": "<", "left": "num", "right": "2" }},
      "destinations": [
        { "probability": { "exp": 0.5 },
          "location": "s0",
          "assignments": [
              { "ref": "num",
                "value": { "exp": { "op": "+", "left": "num", "right": 1 } }

              } ] },
        { "probability": { "exp": 0.5 },
          "location": "s1",
          "assignments": [ { "ref": "num", "value": 3 } ] } ]
    } ]
  } ]
```



[6] https://jani-spec.org/

The JANI snippet above encodes an automaton which describes the behavior depicted by the graph below the JANI example. The automaton is called `aut` and contains a single global bounded integer variable `num` initialized with its lower bound 0 and bounded by 3 from above. The automaton has two locations s_0 and s_1. The initial location s_0 has one outgoing probabilistic edge with two possible destinations. To be able to take this edge, the current location needs to be s_0 and the variable `num` must be smaller than 2, as defined in the guard. With a probability of 0.5 the edge leads back to s_0 and increases the value of `num` by 1. The other part of the probabilistic edge leads to the destination location s_1 with probability 0.5 and `num` is set to 3

3.2. The Quantitative Verification Benchmark Set and the QComp Competition

As already hinted at, a large spectrum of JANI case studies exists. To collect them together with informative metadata and to make them easily accessible for everyone, many of them are part of the *Quantitative Verification Benchmark Set (QVBS)*[7] [132].

On the one hand, the QVBS was partially inspired by other communities also using MDP benchmarks. One of them was the planning community which established the benchmarks of the *International Planning Competition (IPC)* [258] as the standard benchmark set the community is working on. The competition also contains a probabilistic track, the *International Probabilistic Planning Competition (IPPC)* [272], centered around probabilistic models, mainly MDPs, written in the *Probabilistic Planning Domain Definition Language (PPDDL)* [272]. Probabilistic verification and planning have been connected by the author of this thesis [175, 176] by providing translations between PPDDL and JANI, which makes the planning benchmarks also available and attractive for the model checking community, and which especially enabled the author to contribute JANI versions of planning benchmarks to the QVBS.

In addition, MDPs have gained interest in the AI and learning community [106, 110] over the last years, i.e., MDP benchmarks have been designed there too.

On the other hand, in the model checking community only a few sample benchmarks delivered with the tools in their specific language, and the Prism benchmark suite [190] with models in the Prism language, existed at the time of the development of the QVBS. Most of the Prism benchmark suite models were designed for the use in Prism, and therefore they work the best with the BDD engine of Prism [132]. But there are much more model checking techniques specialized for the use of other model types, which again highlights that a general overarching benchmark set comprising models from all areas is required.

[7] https://qcomp.org/benchmarks/

JANI models were basically not collected so far, only the Modest Toolset contained some sample models, and a few of the Prism models had been translated to JANI and were available in a small JANI model repository[8].

Therefore, the idea behind the QVBS was to foster cooperations and new developments in all those communities using formal quantitative models, by providing a collection of benchmarks usable in planning, model checking, and learning contexts, in JANI as a common format.

These major factors built the starting point of the QVBS. The QVBS includes established probabilistic model checking, fault tree, Petri nets, and planning benchmarks originally designed in a variety of languages together with properties to be analyzed on these models. They cover industrial case studies, Petri nets, probabilistic programs, queuing systems, dynamic fault trees, planning models in uncertain environments as well as biological systems. In addition, the QVBS does not only make the models accessible in JANI together with the original version and information on the conversion procedure. It also archives detailed metadata, like tool performance measurements, the state space sizes as well as descriptions and references for the model. Currently, the QVBS contains 78 DTMC, CTMC, MDP, MA, and PTA benchmarks of which 50 have been submitted by the author of this thesis.

With the QVBS, the model checking community owns a common set of realistic and challenging benchmarks on which algorithms and tools can be easily compared, like it is already standard in the planning [258], SMT [26], and software verification [37] community.

These tool competitions gave the inspiration to launch *QComp*[9], the *Comparison of Tools for the Analysis of Quantitative Formal Models*, a quantitative verification competition based on a curated subset of the QVBS comprising DTMC, CTMC, MDP, MA, and PTA models. Participating tools do not have to support all model or property types. In addition, it is not required to work on the JANI version of the benchmarks, but one of the languages contained in the QVBS suffices, e.g., the format the original version of the benchmark is specified in. The competition compares the performance, versatility, and usability of the tools. It is a friendly competition in the sense that there is no global ranking in the end but data comparing several facets of the tool is processed in tables, plots, and diagrams. Performance, versatility, and usability is evaluated in text form such that the strengths and trade-offs between tools can be highlighted. The competition consists of multiple tracks with different types of correctness guarantees on results. For this thesis only one of these tracks is of relevance, which is the track where *often ε-correct results* have to be produced. In this track, tools participate which do not

[8] https://github.com/ahartmanns/jani-models
[9] https://qcomp.org/

guarantee any bound on the difference between the correct value and the tool's result at all but often deliver results in a predefined bound ε, which is set to $+/-10^{-3}$ in the competition. An example for such an approach is standard value iteration. In all of the tracks, the tools can participate with *default parameters*, which have to be the same for all benchmark instances, and in a second round *specific parameters* tweaked per benchmark instance are allowed. We only concentrate on the default track version in this thesis.

A variety of probabilistic model checkers, like EPMC [92, 124], MCSTA [119, 122] of the MODEST TOOLSET [128], PET [48], PRISM [189], STORM [77], and also our MODYSH tool presented in Chapter 4 are being developed in the community nowadays, and are supported by orchestrated initiatives like the QComp verification competition and the QVBS. The author of this thesis was involved in the development process of the first QComp in 2019 [121] and managed the technical setup, the tool executions, and the result evaluations for QComp 2020 [54].

3.3. The Racetrack Benchmark

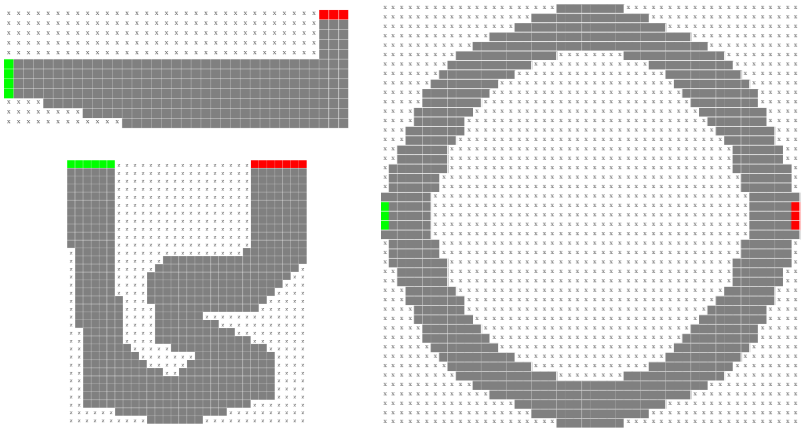


Figure 3.: Maps of Racetrack benchmarks: Barto-small (left top), Barto-big (left bottom), Ring (right).

The Racetrack benchmark with all its different versions and extensions is currently not part of the QVBS, but has been used in different versions in multiple works of the author [16, 20, 103, 105, 106, 109, 107, 110, 178], especially in the works around DSMC presented in Chapter 5.

Originally, Racetrack is a pen and paper game [96] in which a track consists of a two-dimensional grid drawn on a squared sheet of paper, where each cell of the grid is either a starting position, a goal position, a free road cell, or a wall. Exemplary track shapes [27]

are depicted in Figure 3, where starting positions are marked in green, goal positions are colored red, and walls are indicated in gray. Throughout the thesis we use *track* and *map* as synonyms for the description of a Racetrack environment with a specific shape induced by the placement of walls, start as well as goal cells. A vehicle starts with a certain velocity, usually 0, from a position on the start line and tries to reach a goal cell as fast as possible without crashing into a wall. During the race, the player, i.e., the driver, can choose out of nine possible acceleration actions in unit steps to modify the current velocity vector by one unit in each round after every move. The actions are the acceleration directions up, down, left, right, the four diagonals, and keeping the current velocity, which is mathematically encoded as accelerations in x- and y-dimension in $\{-1, 0, 1\}^2$. The difficulty is, that actions may fail, i.e., it is possible that the desired action is not applied and the car continues driving with the previous velocity vector. This behavior models slippery road conditions by introducing noise with a certain probability. This means, the vehicle does not necessarily reach the goal with certainty, even if played optimally.

This simple game lends itself naturally as a benchmark for sequential decision-making in risky scenarios. When modeling the game, it is most natural to construct an MDP. Racetrack was first adopted as a benchmark for MDP algorithms in the AI community [27, 45, 204, 220, 221].

Racetrack in JANI. We encoded the Racetrack benchmark in a JANI Markov decision process. The details of the Racetrack encoding in JANI we make use of in the following chapters are relegated to Appendix A.4. We give a brief summary of the core functionality in the following.

The track itself is represented as a (constant) two-dimensional array whose size equals that of the grid. The JANI files of different Racetrack maps differ only in this array. Vehicle movements and collision checks are represented by separate automata that synchronize using shared actions.

The vehicle automaton keeps track of the current vehicle state via four bounded integer variables, which are the x- and y-position and the directional velocity in x- and y-direction. In addition, two Boolean variables are used indicating whether the vehicle has crashed or reached a goal. The initial automaton location has edges for each of the 9 different acceleration vectors. Each of them updates the velocity accordingly, and sends the current source and next target coordinates to the collision check automaton. It then awaits that automaton to respond with one of three answers: “valid”, “crash”, or “goal”. For the latter two, the automaton moves to a terminal location. For “valid”, the vehicle automaton sets the target coordinates as its new source coordinates, and transitions back to its initial automaton location.

The collision check automaton checks whether the vehicle's next target coordinates lie within the grid. If so, it iterates over the cells on the discretized trajectory from the current source to the next target, and looks up for each such cell whether it represents a wall or goal cell. Such a result is sent to the vehicle automaton as soon as available. If the entire trajectory is found free of such events, the vehicle automaton's request is answered with "valid", and the automaton location is reset, waiting for the next trajectory to check.

Throughout the thesis we consider the single-agent version of the game and use (variations of) the three traditional Racetrack maps illustrated in Figure 3, originally introduced by Barto et al. [27].

Racetrack as Autonomous Driving. The benchmark is a variable and configurable, but admittedly very gross, abstraction of the *autonomous driving challenge*. This challenge consists of a vision of multiple vehicles which autonomously navigate smoothly through city traffic. They avoid accidents by always following the traffic rules, by detecting objects and obstacles on the road, and by driving carefully while watching the weather and road conditions. In addition, they should optimally drive economically by reducing the fuel consumption but reaching the destination as fast as possible. This autonomous driving vision is the most prominent and discussed application of future intelligent systems.

In the most basic form as presented above, Racetrack at first sight only abstractly resembles the autonomous driving challenge with some drastic restrictions relative to the grand vision. Only a single car is considered in the environment, where only walls and the track boundaries are obstacles, and no other moving traffic is around. In addition, the car has a full view on the scene, i.e., not only a certain area in the surrounding is accessible by sensors. Furthermore, no traffic rules or fuel consumption limitations have to be considered, and the weather or road conditions, e.g., modeled through noise, are constant along the track. In comparison to real-life traffic, the Racetrack environment is discretized in a coarse manner, which simplifies the problem considerably.

But a clear benefit of the Racetrack benchmark is, that it is formal and precise. It connects the autonomous driving challenge to the modeling world of MDPs and provides a common formal ground for basic studies in this environment. Racetrack provides laboratory conditions for a systematic, structured, and extensible analysis of, e.g., machine-learnt entities that are supposed to act in that environment. In addition, it can easily be extended, scaled, or varied in such a way that it is usable to investigate machine perception based on collected sensor data, behavior prediction, risk assessment, or trajectory and resource planning. Therefore, it is of interest for many research fields, e.g., model checking, machine learning, planning, and of course cyber-physical systems in general.

There are endeavors, like image segmentation, semantic segmentation, and instance segmentation done with the help of NNs, especially in the context of autonomous driving [89, 241, 255], which try to make the real-world challenge more abstract by concentrating on the relevant data needed to act in the environment. These approaches are used to reduce objects and their properties in an image of the scene to the parts and information relevant for making decisions by describing the scene more abstractly, such that the data can be processed more efficiently. With the current version of Racetrack, we are already at this abstraction level, which is required by many tools and otherwise would have to be achieved by such segmentation methods first.

If required, the level of abstraction of real-world autonomous driving can easily be adapted in Racetrack, w.r.t. continuous time, space constraints, linearization of trajectories, or abstraction from features, like fuel consumption, road surface conditions, speed and acceleration limits, other traffic participants and traffic regulations, moving obstacles, different probabilistic perturbances, and the change from map perspective to ego-perspective of an autonomous vehicle, mediated by vision and other sensor systems, and so on [16].

Finer discretizations of the tracks have, e.g., been considered in the scalability study for DSMC [109] described in Section 5.3.

A suitability analysis in Racetrack environments considered as probabilistic systems consisting of selections of different types of components has enlarged the set of Racetrack variants [20] further. This variant allows the combination of (i) different car engine types, determining the possible maximal and minimal acceleration, the maximal and minimal speed as well as the fuel consumption with (ii) various undergrounds, like ice, sand, and tarmac, and with (iii) different tank sizes, to figure out which car configuration is the most suitable to drive on a certain track most efficiently, w.r.t. travel time, crash risk, and fuel consumption.

To conclude, Racetrack provides a common formal ground and laboratory environment for research in autonomous driving, is easily scalable and extensible, and there is a growing tool support for research on this benchmark [16, 59, 103, 105, 106, 110]. Recently, Racetrack has been used in many works on probabilistic verification [20, 59, 103, 105, 106, 110, 111, 178]. A summary of all these works centered around Racetrack is also available [16], and a website[10] presenting all variants of the benchmark, example tracks, the MDP model variants, and the works using them, has been coordinated by the author of this thesis.

[10] https://racetrack.perspicuous-computing.science/

3.4. The Modest Toolset

The Modest Toolset[11] [128] is a modular framework currently consisting of an integrated collection of six tools centered around modeling and the analysis of hybrid, real-time, distributed, and stochastic systems. It enables the study of probabilities, rewards, real-time behavior, and continuous dynamics of quantitative models by providing verification mechanisms for non-functional properties, like quality, reliability, and performance measures.

The modeling contexts supported by the toolset are based on the stochastic hybrid automata formalism [123], which encompasses networks of stochastic timed automata, (probabilistic) timed automata, Markov decision processes, labeled transition systems, discrete-time Markov chains, continuous-time Markov chains, interactive Markov chains, and Markov automata. Quantitative and qualitative properties on these model types can be evaluated with the tools provided by the Modest Toolset.

All tools of the toolset support models expressed in the JANI format and in the Modest language [123], which is a high-level compositional modeling language for stochastic hybrid systems.

To analyze models of these formalisms, the Modest Toolset provides the tools in the following list. We lay a focus on those tools used in the remainder of the thesis and those relevant for the implementation of our contributions by giving a more detailed description of them.

- mcsta [119, 122] is an exhaustive explicit-state probabilistic model checker based on standard value iteration usable for stochastic timed automata, probabilistic timed automata, and MDPs. It features a disk-based model exploration and analysis engine for very large models which would not fit into memory without the usage of secondary storage space [129].

- moconv can be used to convert models between JANI and Modest in both directions.

- modes [51] is the statistical model checker supporting stochastic hybrid automata, stochastic timed automata, probabilistic timed automata, and MDPs. modes is capable of solving reachability probability and expected reward properties on those models.

  modes thus far offered the explicit options `Uniform` and `Strict` to resolve the nondeterminism in the input model uniformly at random or to stop if nondeterminism is detected during simulation, respectively.

  In modes multiple statistical methods are available, including confidence intervals, Okamoto bound [215], and SPRT [266], to statistically evaluate the results of the

[11] https://www.modestchecker.net/

sample runs. As simulation is easily and efficiently parallelizable, MODES can exploit multi-core architectures during simulation.

To calculate the probability of rare events in nondeterministic models, MODES combines fully automated importance splitting [50, 52, 195] with smart lightweight scheduler sampling [199] to statistically approximate optimal schedulers. With lightweight scheduler sampling it is possible to use efficiently constant memory in the number of states, even for reachability probabilities and undiscounted expected reward properties. The runtime of the approach only depends on the probability that near optimal policies are sampled.

- MOSTA can visualize stochastic hybrid automata semantics of the input model.
- PROHVER [123] is a safety model checker for stochastic hybrid automata.

With the implementation of the probabilistic model checking approach based on dynamic heuristic search and planning techniques presented in Chapter 4, we were able to contribute MODYSH as a sixth tool.

In addition, we extended the statistical model checker MODES with the Deep Statistical Model Checking functionality presented in Chapter 5 by adding the options `NN` and `Oracle` to resolve nondeterminism in MDPs by querying a neural network or an arbitrary decision-making agent connected via a socket communication.

The MODEST TOOLSET is available for Windows, Linux, and Mac OS and it is mainly implemented in C#, except for some special extensions.

With MCSTA and MODES, the MODEST TOOLSET took part in QComp 2019 and 2020 showing convincing performance results demonstrating that both tools belong to the state-of-the-art model checkers.

4. MODYSH: Adapting Dynamic Heuristic Search for Model Checking

As introduced in Section 2.3, heuristic search methods can be used to compute optimal values for reachability probabilities and expected accumulated rewards based on only a small fraction of the states which are sufficient to answer the considered properties. MODYSH is a probabilistic model checker which harvests and extends such non-exhaustive exploration methods originally developed in the planning context. It implements a variant of *asynchronous value iteration* (see Section 2.2.1). Its core functionality is based on enhancements of the heuristic search methods *Labeled Real-Time Dynamic Programming (LRTDP)* [45] and *Find-Revise-Eliminate-Traps (FRET)* [180] with several modifications and extensions to make them work for MDPs with positive and zero-valued rewards on all established property types, except long-run averages and nested properties. MODYSH is thus capable of handling efficiently maximal and minimal reachability probabilities, expected accumulated reward properties as well as bounded versions thereof. The algorithmic elements and their integration are described in detail in the following. The implementation of the new MODYSH tool is integrated in the infrastructure of the state-of-the-art model checking tool the MODEST TOOLSET, and extends the property types supported by this toolset. We discuss the algorithmic particularities in detail, and give correctness and optimality proofs. We furthermore evaluate the competitiveness of MODYSH in comparison to state-of-the-art model checkers in a comprehensive case study rooted in the QVBS introduced in Section 3.2. This study demonstrates that MODYSH is especially attractive to be used on very large benchmark instances which are not solvable by any other tool.

Our contributions with MODYSH comprise the following points:

- We extend, modify, and adapt the well known dynamic heuristic search algorithms FRET and LRTDP in such a way that they are applicable to MDP models with positive and zero-valued rewards to solve minimal/maximal reachability probability, and expected reward properties as well as bounded versions thereof. In addition, we give correctness and optimality proofs for the modified algorithms.

- We present the implementation of these adapted algorithms in MODYSH, a new model checking engine of the MODEST TOOLSET.
- In a comprehensive benchmarking study based on the QVBS in the style of the *default often ε-correct track* of QComp, we compare MODYSH to the other state-of-the-art model checkers, and demonstrate that it is more efficient, time and memory wise, on very large models, many of which are not solvable by any other of the considered tools.

The MODYSH tool is shipped as part of the MODEST TOOLSET and can be downloaded on the MODEST TOOLSET's website[1]. It can be considered as an alternative to MCSTA [119, 122, 129], the exhaustive explicit-state probabilistic model checker based on traditional value iteration in the MODEST TOOLSET. Integrating MODYSH into the MODEST TOOLSET opens it for property types not supported thus far.

An artifact enabling the reproduction of all empirical results reported in this chapter is available online[2] [174].

Organization and Origins of the Chapter. Section 4.1 discusses in detail how LRTDP and FRET can be extended and modified such that they are applicable to MDP structures with positive and zero-valued rewards on the above mentioned set of properties. Section 4.2 presents a large empirical evaluation on QVBS benchmarks demonstrating that model checking with MODYSH is competitive, outperforming state-of-the-art model checkers especially on very large state spaces with a parallel structure. In Section 4.3 we review the related literature in the planning and model checking area. We conclude in Section 4.4 with a short discussion of our achievements and future work.

The entire Chapter 4 is based on a QEST 2021 publication of the author together with Holger Hermanns [173]. In this paper the tool MODYSH has been introduced and the modifications to the existing algorithms have been discussed in detail. In addition, the large case study on the QVBS has been conducted, which compares other model checkers and planners to MODYSH. In this chapter, we extend the content of the paper by giving several additional examples on how to apply the algorithms by demonstrating how the procedures operate on exemplary MDPs in addition to the theoretical explanations. Futhermore, we provide more detailed results of the evaluation than in the paper.

A clear delineation between this chapter and the author's Master's thesis [171] could be of interest. To give a short summary, no content of or work done for the Master's thesis has been used in this chapter. The only commonality of the two works is that they treat

[1] https://www.modestchecker.net/
[2] http://doi.org/10.5281/zenodo.4922360

FRET-LRTDP in the context of the MODEST TOOLSET. But the Master's thesis only contains a pure re-implementation of the original algorithms, whereas this chapter is about modifications and extensions of them, which resulted in a completely new implementation. The evaluation parts of the Master's thesis and this chapter have an entirely different focus and scope, and are based on different tools and benchmarks.

In more detail, the Master's thesis contains a translation from JANI into PPDDL, but not the way back, which in turn is covered in a JAIR article [176]. In addition, it contains the exact re-implementation of FRET-LRTDP as it is defined in the pseudo code of the papers introducing these algorithms [45, 180]. Specifically, the implementation does not contain *any* of the modifications presented in this chapter and is, like the original works, only applicable to GSSP problems. The performance comparisons done in the Master's thesis have a completely different target and scope w.r.t. the tools und benchmarks used.

The work presented in this chapter is a manifold enhancement of the part of the Master's thesis about FRET-LRTDP, not only w.r.t. adaptions to the algorithms to make it applicable to more MDP and property types, but also w.r.t. the data structures and implementation details as emphasized by this quote from the Master's thesis:

In the thesis it has been noted that the implementation is not as efficient as it could be *"[. . .] because the compiled network used in the [MODEST TOOLSET] is not meant to be changed. Following an action in the inverse direction from the target state to the start state is also not intended. Clearly, a preprocessing step could be done to compile the MODEST TOOLSET's network representation into a graph structure providing all functionality for our purposes. But this would lead to additional memory consumption and longer runtimes [. . .] Compiling the network in advance into an adequate data structure would destroy the idea of LRTDP which is avoiding to inspect the whole state space and unfolding it in advance"* [171, p. 28].

This issue has been solved with the implementation of efficient data structures especially tailored to explore the state space in MODYSH.

4.1. Theoretical Contributions

The research field using probabilistic planning and heuristic search for computing optimal values for reachability and expected reward properties is quite active as we will see in more detail in Section 4.3, but the approaches are fragmented w.r.t. to assumptions on the model's problem classes and supported property types. This is also the case for LRTDP and FRET, on which the functionality of MODYSH is based.

The original version of LRTDP [45] is designed for SSP models fulfilling the conditions (i′) and (ii′) of Definition 20, i.e., there has to be at least one proper policy from every state and all non-proper policies must accumulate infinite reward. To easily fulfill the second requirement,

the algorithm acts in an environment with strictly positive rewards, where the goal is to minimize the expected accumulated rewards. The original work on FRET [179, 180] extends the problem space to GSSP models (see Definition 21) and operates under the assumptions (i) and (ii″), stating that there is at least one policy reaching the goal with certainty, and that the expected sum of negative rewards is bounded from below. FRET allows arbitrary, i.e., positive and negative, rewards. In the original work, it is used for cost minimization over MDPs with bounded negative cost accumulation. The algorithm calculates the optimal policy over all policies reaching the goal with probability 1, a restriction not needed when dealing with SSP problems. All these restrictions are necessary to guarantee convergence to the optimal value in the original version of the algorithms. The authors of FRET showed that other MDP problems which do not directly fall into the category of GSSPs can be reduced to it by model transformations, such that FRET is applicable on them. This is for instance the case for *MaxProb* properties.

The adaptions of algorithms we propose do not rely on any of those assumptions regarding the model characteristics and property types, and no model transformations are needed, which also makes the correctness and optimality proofs easier. Especially, we do not rely on restrictions regarding goal reachability, allow positive and zero-valued rewards, and the reward accumulation does not have to be bounded in the model. We do restrict to positive and zero-valued reward structures which is assumed often in the probabilistic model checking community [22, 127, 164, 226], because otherwise expected rewards may be unbounded or not even well defined (for more details see [226, Chapter 7]). Our implementation is applicable to such MDPs having positive or zero-valued rewards on all established property types (except for long-run averages and nested properties), i.e., maximal and minimal reachability probabilities, expected rewards, and bounded versions thereof.

In the discussion of the algorithms in the following, we make use of two notions not in focus thus far, to talk about states from which a goal is not reachable, and to indicate that a policy is defined for all states reachable with this policy. We assume an MDP $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, s_0, \mathcal{S}_* \rangle$ is given (see Definition 6).

Definition 22: Sink States

States from which $\mathcal{S}_*$ cannot be reached with positive probability are called *sink states* and collected in $\mathcal{S}_\perp$.

Definition 23: Closed Policy

A partial memory-less policy π is called *closed for a state* $s \in \mathcal{S}$, if $\pi(t)$ is defined for every state $t \notin \mathcal{S}_\perp \cup \mathcal{S}_*$ that is eventually reachable with positive probability from s by following π. A partial memory-full policy π is *closed for a path* $\tau \in \mathcal{S}^+$ if it is defined for every path $\tau w \in \mathcal{S}^+$ for which $last(w) \notin \mathcal{S}_\perp \cup \mathcal{S}_*$.

A policy is *closed* if it is closed for the initial state s_0 of the MDP it is used on.

Algorithm Overview. We introduce our algorithmic contributions one-by-one and go in detail through all parts of the algorithms. Thereby, we describe the original functionality of LRTDP and FRET in combination with our adaptions. The pseudo code can be found in Algorithm 2 – 7. All modifications, adaptions, and extensions made to the original versions are marked in blue. If existing, the original version of modified lines is stated in comments of the form ▷. . .. To give an intuition of how the whole procedure works for each of the property types, examples are given highlighting the subtleties of the specific cases.

For the modified version of the whole procedure, the base algorithm called *General Labeled Real-Time Dynamic Programming (GLRTDP)* given in Algorithm 2, describing LRTDP and our generalized version of it, uses two flags dependent on the property class to be evaluated. `max-rew` is set to *True* if a maximal expected reward property is evaluated, otherwise its value is *False*, analogously for `min-rew`. We do not use explicit flags for indicating maximal or minimal reachability probabilities because there are no code fragments specific to these property types (except for line 2 in Algorithm 4 which is to be understood dependent on the choice of minimum or maximum in general). In addition, we assume that the initial and current value function, V_0 and V_i, are always globally accessible in all algorithms.

Before presenting all parts of the procedures in close detail, we give a brief summary and intuition of how the original algorithms work which build the basis of our contribution.

LRTDP is a heuristic search dynamic programming optimization of standard value iteration operating asynchronously. It is the base algorithm of the whole procedure and expects as inputs the state s of the MDP for which to evaluate the property, and the desired result precision ε. Note that, like for standard value iteration (see Section 2.2 and Algorithm 1) there are a few specific cases where the algorithm terminates but the result has an error larger than ε, because ε-consistency is only checked locally. Speaking in terms of the QComp categories to classify algorithms w.r.t. their correctness guarantees, FRET-LRTDP and also our modified version, thus, is *often-ε-correct* (c.f. Section 3.2).

Algorithm 2: General Labeled Real-Time Dynamic Programming (GLRTDP)

```
1:  proc GLRTDP(s: state; ε: float)
2:      max-rew, min-rew = True, if max., resp. min. reward property is calculated.
3:      value function V globally accessible.
4:      while ¬Solved(s) do
5:          GLRTDP-TRIAL(s, ε)
6:      return V

7:  proc GLRTDP-TRIAL(s: state, ε: float)
8:      visited = Empty-Stack
9:
10:     while ¬Solved(s) do
11:         visited.Push(s)
12:         v_old = V(s)
13:         UPDATE(s)
14:         v_new = V(s)
15:         if Is-ε-cons(v_old, v_new) then break
                                          ▷ original condition IS-GOAL(s)
16:         a = GREEDY-ACTION(s)
17:         if a ≠ NULL then
18:             s = PICK-NEXT(a, s)
19:             if max-rew && visited.Contains(s) then
20:                 if ELIM-CYCLE-MAX-REW() then
21:                                   ▷ returns True if a cycle is eliminated, description in text
22:                     V(init-node) = ∞
23:                     Solved(init-node) = True
24:                     return
25:             else
26:                 if min-rew && visited.Contains(s) then
27:                     if ELIM-CYCLE-MIN-REW() then
28:                                   ▷ returns True if a cycle is eliminated, description in text
29:                         s = MERGED-NODE(s)
30:                                   ▷ returns the merged node replacing the previous node s
31:         else
32:             break
33:
34:         while visited ≠ Empty-Stack do
35:             s = visited.Pop()
36:             if ¬CHECK-SOLVED(s, ε) then
37:                 break
```

Algorithm 3: Check-solved Procedure used in GLRTDP

```
1:  proc CHECK-SOLVED(s: state; ε: float)
2:      rv = True
3:      open = Empty-Stack
4:      closed = Empty-Stack
5:
6:      if ¬Solved(s) then open.Push(s)
7:
8:      while open ≠ Empty-Stack do
9:          s = open.Pop()
10:         closed.Push(s)
11:
12:         if Dead-end(s) || Goal(s) then continue
13:
14:         a = GREEDY-ACTION(s)
15:         if max-rew || min-rew then
16:             check-∞-loop = False
17:             for each s' s.t. P(s, a, s') > 0 do
18:                 if closed.Contains(s') then
19:                     check-∞-loop = True
20:             if max-rew && check-∞-loop then
21:                 if ELIM-CYCLE-MAX-REW() then
22:                     V(init-node) = ∞
23:                     Solved(init-node) = True
24:                     return True
25:             else
26:                 if min-rew && check-∞-loop then
27:                     if ELIM-CYCLE-MIN-REW() then
28:                         return False
29:
30:         v_old = V(s)
31:         UPDATE(s)
32:         v_new = V(s)
33:         if not Is-ε-CONS(v_old, v_new) then
34:             rv = False
35:             continue
36:         for each s' s.t. P(s, a, s') > 0 do
37:             if ¬Solved(s') && ¬In(s', open ∪ closed) then
38:                 open.Push(s')
39:
40:     if rv then
41:         for each s ∈ closed do
42:             Solved(s) = True
43:     else
44:         for s ∈ closed do
45:             UPDATE(s)
46:     return rv
```

Algorithm 4: Subroutines of GLRTDP and FRET

```
1:  proc GREEDY-ACTION(s: state)
2:      return argMinMax_{a∈A(s)} QVALUE(a, s)
3:                                   ▷ Min/Max depending on the property to calculate

4:  proc QVALUE(a: action, s: state)
5:      return
            ∑_{s'} P(s, a, s') · (R(s, a, s') + V(s'))

6:  proc UPDATE(s: state)
7:      a = GREEDY-ACTION(s)
8:      V(s) = QVALUE(a, s)

9:  proc PICK-NEXT(a: action, s: state)
10:     pick s' randomly from all successors with P(s, a, s') > 0
                                     ▷ originally with probability P(s, a, s')
11:     return s'

12: proc IS-ε-CONS(v_old, v_new: double)
13:     if abs(v_old − v_new) ≤ ε || v_old = ∞ && v_new = ∞ then
14:         return True
15:     return False
```

To find an optimal policy, up to a predefined accuracy ε, starting in a specific initial state, LRTDP attempts to avoid the need for exploring the entire state space and delivers the requested values for the initial state only, rather than for all states like it would be the case in standard value iteration. It constantly keeps updating a *current best solution* of the state value estimates on single exploration paths, i.e., it works on a partial value function providing the current state value estimates. The procedure operates asynchronously, which means that in each round only a single state is selected for an update. These updates are performed during repeatedly sampling *trials*, i.e., executions starting in the initial state, selecting the next state greedily, and ending once a terminal state is reached. While doing so, the optimal policy is constructed incrementally by extending a partial policy step by step in a greedy fashion until it is closed. The FRET procedure is wrapped around calls to LRTDP to guarantee convergence of LRTDP to the optimal value in MDP structures containing cycles of specific types. It eliminates cycles in the MDP to guide the exploration of LRTDP to the correct solution.

In the following, we discuss the functionality of the modified and extended algorithms. In fact, the original algorithmic contributions have been made without a specific focus on reachability probabilities, which as long as zero-valued rewards are supported, can actually be cast into reward accumulations. We here make an explicit distinction between these cases

for the purpose of better explainability and for the purpose of more direct and hence faster implementation in MODYSH.

4.1.1. Reachability Probability Properties

For reachability probability properties, `max-rew` and `min-rew` are set to *False*. We first concentrate on calculating *MinProb*, i.e., $\mathcal{P}_{\min}(\mathcal{S}_U \mathcal{U} \mathcal{S}_*)$. We detail our modifications to the original version of the algorithm in order to enable that condition (i′) and thus (i) of Definition 20 of SSPs and Definition 21 of GSSPs can be dropped. Afterwards, we turn to *MaxProb* and show how GLRTDP in combination with a slightly modified version of FRET can be used to solve this kind of property on MDPs with positive and zero-valued rewards, too. Kolobov et al. [180] already provided a reduction to show that FRET in combination with LRTDP is applicable to general *MaxProb* properties, even though condition (i′) of GSSPs is violated at first sight. We will give an alternative proof, based on the proof for *MinProb*, demonstrating that our implementation is also valid for MDP types with positive and zero-valued rewards as defined above, not only for problems having at least one proper policy.

We denote by $V^\pi : \mathcal{S} \to [0, 1]$ the value function indicating the goal reachability probabilities induced by policy π. Intuitively, goal states in $\mathcal{S}_*$ have goal reachability probability value 1 while sink states and other states enforced to be avoided have probability value 0. Using a reward function defined as $\mathcal{R}(s, a, s') = 1$ if $s \notin \mathcal{S}_* \wedge s' \in \mathcal{S}_*$ and 0 otherwise, and then applying the Bellman equation given in Equation 2.1 for the minimum case of synchronous value iteration will iteratively fill the partial policy bottom up for this case. We can omit this reward function in the back propagation formula by initializing goal states with 1 while sink states and other states to be avoided get a value of 0 directly. This procedure leads to the following definition of V^π which constitutes the least fixpoint of Equation 4.1.

$$V^\pi(s) = \begin{cases} 1 & \text{if } s \in \mathcal{S}_*, \\ 0 & \text{if } s \in \mathcal{S}_\perp \cup \overline{\mathcal{S}_U} \setminus \mathcal{S}_*, \\ \sum_{s' \in \mathcal{S}} \mathcal{P}(s, \pi(s), s') \cdot V^\pi(s') & \text{otherwise.} \end{cases} \tag{4.1}$$

Minimum Reach Probability. For *MinProb* properties $\mathcal{P}_{\min}(\mathcal{S}_U \mathcal{U} \mathcal{S}_*)$ (see Section 2.2) the objective is to find the minimal probability to reach a state in $\mathcal{S}_*$ if initialized in s_0 while avoiding the complement of $\mathcal{S}_U$. We are ultimately interested in the minimal value over all possible policies on the MDP:

$$V^*(s_0) = \min_\pi V^\pi(s_0). \tag{4.2}$$

For the calculation of reachability probabilities, it is sufficient to consider partial policies under the condition that they are closed for state s_0 because for the values of properties of this type, it is irrelevant what happens in unreachable parts of the state space. This means, we can refine Equation 4.2 in the following way:

$$V^*(s_0) = \min_{\pi:\pi \text{ closed for } s_0} V^\pi(s_0). \tag{4.3}$$

An admissible initialization for the case of the calculation of the minimum is a valuation of 0, except for goal states which get a value of 1.

This amounts to replacing the third line of (4.1) by

$$\min_{a \in \mathcal{A}(s)} \sum_{s' \in \mathcal{S}} \mathcal{P}(s, a, s') \cdot V^\pi(s') \quad \text{otherwise.} \tag{4.4}$$

which echoes the greedy nature of the computation.

However, giving up synchronicity in favor of a heuristic approach is the key to efficiency. For *MinProb* the procedure is completely implemented in GLRTDP, a generalization of LRTDP [45, Algorithm 4]. The pseudocode is shown in Algorithm 2. The subroutines used in this procedure can be found in Algorithm 3 and 4. As discussed, the algorithm iteratively selects only a single state for a Bellman update in each round. It constantly updates a *current best solution*, i.e., a partial function providing the current state value estimates, and repeatedly runs *trials* (Algorithm 2, line 5), i.e., sample executions of the MDP, starting from the initial state, and ending once a state is reached for which an update does not change the value by more than ε, i.e., ε-consistency is reached locally in this update step. This termination check happens in line 15. Lines 19-30 are not relevant for this case. To determine which successor state to follow after state s in the trial construction, GLRTDP considers an action $a \in \mathcal{A}(s)$ *greedy* with respect to the current value function (line 16), i.e., one that minimizes Equation 4.4 for s (cf. Algorithm 4, line 2 and 4, without reward accumulation) [45, Algorithm 2], and then selects a successor state (line 18 in Algorithm 2). Picking the next state randomly, i.e., with a uniform distribution, from the set of successors of the greedy action (cf. Algorithm 4, line 9) instead of taking the probabilities into account is an optimization which leads to better performance as already noted in the probabilistic extension of FAST DOWNWARD [248], because also less likely successors are explored frequently.

To summarize, during these trial executions always a policy greedy w.r.t. the current value function is followed and the values of visited states are updated such that the values in the relevant parts of the state space are constantly improved towards the optimal value.

The entire exploration procedure is systematic, which means that it does not starve relevant states if the heuristic function used for initialization is admissible, i.e., it will not allow a state, which has not converged so far, to stay in the greedy graph forever without its value

being revised. Therefore, it is guaranteed to converge to an optimal solution w.r.t. ε, the desired convergence accuracy.

After each trial exploration, those states are labeled as *solved* whose values and those of their descendants have reached ε-consistency (cf. Algorithm 3) [45, Algorithm 3]. Trials are terminated immediately at solved states, fostering convergence. GLRTDP terminates the value update procedure as soon as the initial state is solved (cf. Algorithm 2 line 4, 10, 36).

This is possible because a value remains ε-consistent if its descendants' and its own value do not change by more than ε anymore (Algorithm 3). This is because $V(s)$ can only change by more than ε if the greedy graph starting in s changes or the value of a descendant changes by more than ε. The graph can only change if the value of a state within the graph changes. Updating states outside the greedy graph will never make them part of it, because by the monotonicity property, updates according to the Bellman function can only make the states less attractive [35]. Thus, a state's value can only change by more than ε if a descendant changes by more than ε but then it cannot have been marked as solved before.

This algorithm converges faster than classical value iteration because not all states need to be converged (or even updated) before terminating. The termination criterion is similar to ε-convergence in standard value iteration.

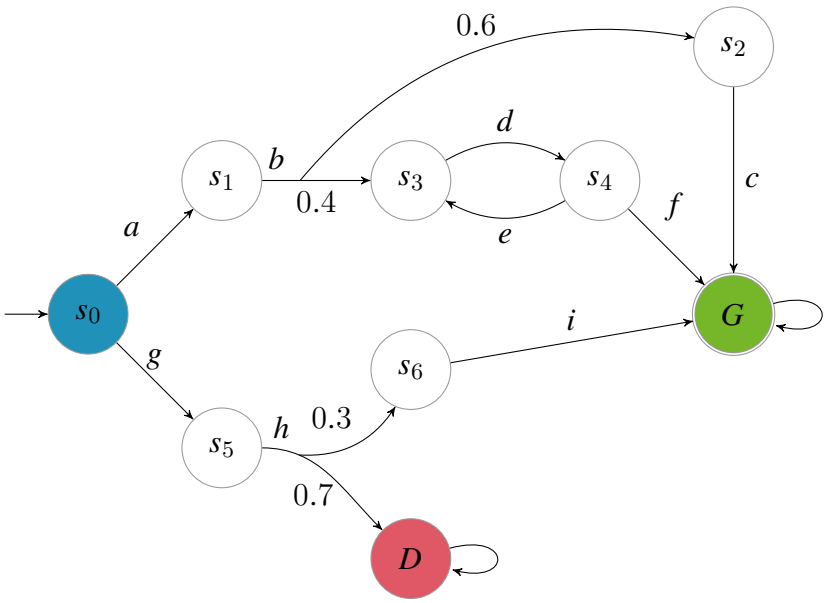
However, if there occurs a cycle in a policy, it needs to be handled during the construction of trials in GLRTDP to guarantee convergence to an optimal value function. In the *MinProb* case permanent as well as transient traps (see Definition 18) have to be treated as dead-ends because in the worst case it is possible to always nondeterministically take an edge leading back to a state in the cycle instead of leaving the loop, i.e., $\mathcal{P}_{\min}$ of eventually reaching the goal is 0 in such traps. This is done indirectly by the termination criteria in line 15 of Algorithm 2 and the check before adding a new state in line 37 of Algorithm 3, respectively. Because of the initialization with 0, values of trap states will lead to a cut of this trial immediately, because these states never change their value in an update and stay ε-consistent, i.e., the cycle is not explored further and the algorithm concentrates on other branches.

To sum up, when calculating *MinProb* over an MDP, GLRTDP presented in Algorithm 2 with an admissible initialization for this case and CHECK-SOLVED() as in Algorithm 3 can be used. We will explain in the following why this combination is correct and converges to the optimal fixpoint. A formal proof can be found in Appendix A.2.

All greedy policies inspected by GLRTDP at some point end in a goal state or a dead-end state. This could be a real dead-end, i.e., a sink state with only a self-loop, or a trap. Because of the initialization, their value is already 0. In addition, we tag these states, do not explore them further and propagate their value back through the graph. Cycling forever is not possible

because eventually all such cycles in greedy policies are eliminated by setting the values of the states in the cycle to 0 and not exploring them further. Having this, we can state that at some point no more states have to be explored in GLRTDP because all relevant traps are eliminated and a goal or a sink has been found, since the models we treat are finite MDPs. In this case, the current greedy policy is fully explored. Then GLRTDP runs until the state values of the current greedy policy are converged up to ε. Even if the greedy policy is not the same in every iteration, at some point it will stay within a set of states which are part of finitely many policies. The values of these states converged close enough to the optimal ones such that the algorithm concentrates on these policies. The value function used in GLRTDP is initialized admissibly and therefore can only monotonically increase and approach the optimal result (fixpoint) from below. When this point is reached, the whole procedure terminates. This fixpoint has to be optimal because the Bellman equation only has one fixpoint [36].

Example 7: Calculating *MinProb* with GLRTDP



To see how all of the theory discussed so far works in practice, we demonstrate the calculation of *MinProb* by using GLRTDP. Consider the MDP depicted above. This example is larger than the MDPs we saw before to show all the facets of the algorithm after having it introduced in detail above.

If we want to calculate the minimal probability to reach the goal, we initialize the states admissibly with 0 for all non-goal states, and with 1 for goal states. This means, if we start the first trial built by GLRTDP, s_0 is initialized with 0 as well as its successors s_1 and s_5. Afterwards, it has to be decided greedily which of them to explore first. Since both successors currently have a value of 0 the decision is taken randomly with a

uniform distribution. Let us assume s_1 is taken first, which leads to an initialization of s_2 and s_3 with 0. Since b is a probabilistic edge, the successors of both states have to be explored further in this case to get an updated value for s_1. In this case, it is decided randomly which one comes first. Let us assume it is s_2. Its only successor is G which is initialized with 1 and not explored further because it is a goal state. The value is back propagated to s_2 during updating this state's value. Then s_3 is explored further, which leads to an initialization of s_4 with 0. The successors of s_4, i.e., G and s_3, are already initialized. Updating s_4 takes greedily the best value of the two successor states, which is 0 from s_3 in case of *MinProb*. The current trial at this point is described by: starting in s_0, applying action a, reaching s_1, applying action b, in case of reaching s_2, applying c, and in case of reaching s_3, applying d, reaching s_4, applying e, which leads back to s_3. This means, in the end this trial contains a loop between s_3 and s_4. But as explained in the description of the algorithm above, the loop in this case is caught by the termination criterion in the check before adding a new state in line 37 in Algorithm 3, i.e., the exploration of this part of the graph ends here and the value of s_1 can be updated with $0.4 \cdot 0 + 0.6 \cdot 1 = 0.6$. Being back at s_0 now in the update procedure called in line 45 of Algorithm 3, the best value of s_1, which is 0.6, and s_5, which has still its initial value 0 from the beginning, is taken greedily. Because we are calculating *MinProb*, the value of s_0 stays 0.

Now, the CHECK-SOLVED() procedure is done and returns False, because several states have not been ε-consistent during the value updates before.

Therefore, GLRTDP starts another trial in s_0. As above, the values of the successor states s_1 and s_5 are taken into account to decide how to proceed with the trial. This time it is clear which action to take because g leads to a state with value 0, whereas s_1 has currently an admissibly estimated minimal goal probability of 0.6. To update s_5, the values of s_6 and D are needed, which both get initialized with 0. Because D is a dead-end, only s_6 has to be explored and updated further, which back propagates the value 1 from G. Then s_5 is updated to $0.3 \cdot 1 + 0.7 \cdot 0 = 0.3$. Afterwards s_0 again gets the best value out of the values of s_1 and s_5, i.e., out of 0.6 and 0.3, which is 0.3.

Once again, in this round not all states have been ε-consistent, which leads to a third trial run of GLRTDP in which finally all states can be marked as *solved*, and the procedure terminates with the correct result of 0.3 for the minimal goal reachability probability.

Maximum Reach Probability. For *MaxProb* properties $\mathcal{P}_{\max}(\mathcal{S}_U \; \mathcal{U} \; \mathcal{S}_*)$ the objective is to find the maximal probability to reach a state in $\mathcal{S}_*$ while avoiding the complement of $\mathcal{S}_U$. An admissible initialization is 1, except for states from which only dead-end states can be reached, which get a value of 0. $\mathcal{P}_{\max}$ can be calculated by changing the initialization and replacing the occurrences of $\min$ by $\max$ in Equations 4.2, 4.3 and 4.4.

Algorithm 5: Find, Revise, Eliminate Traps (FRET)

```
1:  M is the graph of the MDP
2:  proc FRET(M, s : state)
3:      V_i = GLRTDP(s, ε)
                                  ▷ originally FIND-AND-REVISE(M, V_0)
4:      (V_{i+1}, elim-trap) = ELIMINATE-TRAPS(M, V_i)
5:      while elim-trap do
6:          V_i = V_{i+1}
7:          V_{i+1} = GLRTDP(s, ε)
                                  ▷ originally FIND-AND-REVISE(M, V_i)
8:          (V_{i+1}, elim-trap) = ELIMINATE-TRAPS(M, V_{i+1})
```

Algorithm 6: Eliminate-Traps (for *MaxProb*)

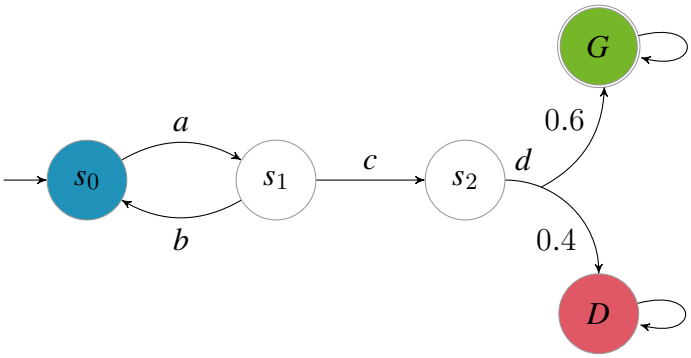
```
1:  proc ELIMINATE-TRAPS(M, V)
2:      elim-trap = False
3:      V_next = V
4:      G_V = {S_V, A_V} ← V's greedy graph
5:      SCC = Tarjan(G_V)
6:      CSet = ∅
7:
8:      for each SComp C = {S_C, A_C} ∈ SCC do
9:          if ∄(s_i, s_j) ∈ A_G : (s_i ∈ S_C, s_j ∉ S_C)
                && (∄g ∈ G : g ∈ S_C) then
10:             CSet = CSet ∪ {C}
11:
12:     for each C = {S_C, A_C} ∈ CSet do
13:         if ∄a ∈ A, s ∈ S_C, s' ∉ S_C : T(s, a, s') > 0 then
14:             for each s ∈ S_C do
15:                 V_next(s) = 0
16:             MergeSCC(C)
17:             elim-trap = True
18:         else
19:             A_e = {a ∈ A | ∃s ∈ S_C; s' ∉ S_C : T(s, a, s') > 0}
20:             m = max_{s∈S_C, a∈A_e} QVALUE(s, a)
21:             for each s ∈ S_C do
22:                 V_next(s) = m
23:             MergeSCC(C)
24:             elim-trap = True
25:     return (V_next, elim-trap)
```

In MODYSH, we use a combination of GLRTDP (Algorithm 2, `max-rew=min-rew=`*False*) and a modified version of FRET (Algorithm 5 and 6), adapted from the originals [180, Algorithm 1] to calculate *MaxProb*. As already shown in the original work on FRET, the combination is needed to guarantee convergence of GLRTDP for *MaxProb* [179, 180]. In FRET, iterations of GLRTDP followed by a call to ELIMINATE-TRAPS() to eliminate zero-valued reward cycles are performed. In the original version of FRET, any Find-and-Revise algorithm is foreseen, we fix that to GLRTDP (Algorithm 5, line 3 and 7) in our implementation. The call to ELIMINATE-TRAPS() (line 4 and 8) is needed if facing zero-valued reward cycles, because these may induce convergence of GLRTDP-trials to a non-optimal value by always choosing an action that loops on the cycle, and thus the goal is never reached (line 16 and 18 in Algorithm 2). The trap elimination procedure evaluates and accordingly changes the value function computed in the last iteration of GLRTDP and the graph it is working on, thus guaranteeing progress in its next call (Algorithm 5, line 4 and 8). This is achieved by finding and eliminating *traps* (cf. Algorithm 6). States which are part of a trap are merged into a single new state replacing all trap states. In contrast to *MinProb*, where traps are handled directly during the trial construction, permanent and transient traps have to be handled differently here. When calling ELIMINATE-TRAPS(), all SCCs in the current greedy policy are collected using Tarjan's Algorithm [250] (Algorithm 6, line 5) and it has to be checked if these SCCs are traps (line 8). First, permanent traps (line 13) are dead-ends from which the goal can never be reached. Therefore, all states' values in this SCC can be set to 0 (line 15) and the states of the SCC can be merged into one. If the SCC is a transient trap (line 19), it has to be left to reach the goal eventually. From all states in the SCC it is possible to take the exit with the highest probability value to reach the goal (line 20). Therefore, we merge these states and set the resulting state to this value (line 21-23). In the next GLRTDP trial this will change the greedy policy, i.e., the cycle is eliminated from the greedy graph. The algorithm terminates if the policy of the last GLRTDP run does not contain a trap anymore.

While the original version of FRET [180] considers in each trap elimination step *all* actions that are optimal according to the current value function, our implementation uses an optimization on the input of Tarjan's algorithm (line 5), called FRET-π [248], considering in the subgraph of the state space inspected during trap elimination only those transitions which are given by the current greedy policy.

To demonstrate how GLRTDP and FRET are combined to calculate *MaxProb* and to highlight why FRET is needed, we discuss the full execution of the algorithms in the following example.

Example 8: Why is FRET needed?



If we execute GLRTDP on the MDP depicted above to calculate $\mathcal{P}_{\max}(\Diamond G)$, we start exploring s_0 after initializing it with 1. There is only one successor state, s_1, which is initialized with 1. Exploring the successors of s_1 leads (i) to s_2 which is again initialized with 1, and (ii) back to s_0 which has already been initialized. GLRTDP now decides greedily which successor to explore further. But since both successor states have the same value, the decision is taken randomly, i.e., each of them is considered equiprobable. We assume that the algorithm takes action b back to s_0. Now the trial is done because there are no successor states of states in the trial, consisting of the subgraph induced by s_0 and s_1, to be explored further, and the two states are ε-consistent because an update of the current value function does not change their values anymore. s_2 is not part of the trial. It has been initialized just to update the value of s_1, and to allow the algorithm to decide how to proceed the trial in s_1. Bellman updates do not change the values of s_0 and s_1 anymore and GLRTDP is done. Currently, the maximal goal probability of s_0 calculated so far is 1, which is not the correct final result.

At this point FRET is called and detects the BSCC $\{s_0, s_1\}$ in the greedy graph induced by the current value function. This BSCC in the greedy graph forms a transient trap (see Definition 18) because there is an edge leading out of it in the full graph, i.e., it is only an SCC in the full graph of the MDP. The trap has to be eliminated, such that it is not misleading GLRTDP in the next iteration anymore.

Then GLRTDP starts anew, walks down the path of s_0, s_1, s_2 and initializes the goal G with 1 and the dead-end D with 0. A Bellman update of s_2 assigns a value of $0.6 \cdot 1 + 0.4 \cdot 0 = 0.6$. This value is propagated back to s_0 and is the correct maximal probability to reach the goal.

To sum up, when calculating *MaxProb* over an MDP, we call FRET, given in Algorithm 5, with GLRTDP, shown in Algorithm 2, with an admissible initialization for this case. The trap elimination procedure in FRET is instantiated with Algorithm 6. In the following, we give an intuition about why the presented combination of GLRTDP and FRET solves

MaxProb properties on MDP structures having positive and zero-valued rewards correctly by converging to the optimal fixpoint, not only on problems having at least one almost-sure policy and without transforming the MDP into a GSSP structure. A proof relying on such a transformation has been given in [180] but our approach is completely independent of such assumptions and therefore more direct and easier. A formal proof of the correctness of this approach for MDPs with positive and zero-valued rewards in the style of the proof for *MinProb* can be found in Appendix A.3.

All greedy policies inspected by GLRTDP at some point end in a goal state or a dead-end state. This could be a real dead-end, i.e., a sink state with only a self-loop or a permanent trap which has been transformed to a dead-end by the cycle elimination of FRET. If it is a permanent trap identified by FRET, the values of all states in it are set to 0. Otherwise, when the sink state is discovered for the first time, its value is also directly set to 0. This means, we tag these states, do not explore them further, and propagate their value back through the graph. Cycling forever is not possible because FRET eventually eliminates all such cycles in greedy policies. With this, we can state that at some point no more states are left to explore in the current GLRTDP trial because all relevant traps are eliminated or a goal or a sink has been found. Then GLRTDP runs until the state values of the current greedy policies are converged up to ε. Even if the greedy policy is not the same in every iteration, at some point it will stay within a set of greedy states which are part of finitely many greedy policies. The values of these states converged close enough to the optimal ones such that the algorithm concentrates on these policies. The value function used in GLRTDP is initialized admissibly, and therefore can only monotonically decrease and approach the optimal fixpoint from above. When this point is reached (up to ε), the entire procedure (GLRTDP + FRET) terminates. This fixpoint must be the optimal one because the Bellman equation only admits a single fixpoint [36].

4.1.2. Expected Accumulated Reward Properties

Expected reward properties $ER_{opt}(\mathcal{S}_U \; \mathcal{U} \; \mathcal{S}_*)$ ask for the minimal or maximal (referred to by opt) expected accumulated reward when reaching a goal state. For the reachability probability properties considered thus far, we have been able to ignore the reward function of the MDP, which is equivalent to assuming it to be 0 except for actions leading to goal states. The calculation of ER_{opt} proceeds very much in the same way as the algorithms discussed before. Iteratively, a variation of the Bellman function updates as presented in Equation 2.1 is performed, where contrary to the $\mathcal{P}_{opt}$-case, shown in Equation 4.1, rewards are taken into account. The conceptual variation is that goal states initially get a value of 0 and states $s \in \mathcal{S}_\perp \cup \overline{\mathcal{S}}_U \setminus \mathcal{S}_*$ a value of ∞.

The value function indicating the expected reward to reach a goal state induced by policy π is then given analogously to Equation 4.1 by Equation 4.5.

$$V^{\pi}(s) = \begin{cases} 0 & \text{if } s \in \mathcal{S}_* \\ \infty & \text{if } s \in \mathcal{S}_\perp \cup \overline{\mathcal{S}}_U \setminus \mathcal{S}_* \\ \sum_{s' \in \mathcal{S}} \mathcal{P}(s, \pi(s), s') \cdot (\mathcal{R}(s, \pi(s), s') + V^{\pi}(s')) & \text{otherwise} \end{cases} \tag{4.5}$$

Reward maximization and minimization is then calculated similar to the maximal and minimal reachability probabilities by maximizing and minimizing over all applicable actions in the third line of the equation.

Correctness and optimality proofs for these property types are very similar to the proofs for *MaxProb* and *MinProb* spelled out in the appendices and are therefore omitted.

Reward Maximization. For ER_{max} `max-rew` is set to *True* in GLRTDP. A trivial admissible initialization is 0 for goal states and ∞ for all others. But this is not practically feasible everywhere. Dead-ends can directly get a value of ∞, according to the definition of expected rewards, because if the goal is not reachable with certainty, the result is ∞. In the implementation, we use the constant $Positive\text{-}Infinity$ for this case. Initializing non-goal states with ∞, which is equivalent to assuming the largest possible overapproximation, is not practical. We therefore approach an admissible initialization for non-dead-end states from below by starting with a smaller $current\text{-}max$, obtained by *exponential search* [33]. This means, we execute full GLRTDP runs, as long as one of the final state values after termination is larger than the last $current\text{-}max$, because if this happens, the initialization has not been admissible. In each iteration the new $current\text{-}max$ is set to the largest state value of the previous iteration increased by 1 and multiplied by 2, which leads to the fastest solution we found in our experiments. This is shown in Algorithm 7 (contributed by us but not colored entirely in blue).

Algorithm 7: Exponential Search for ER_{max} Calculation

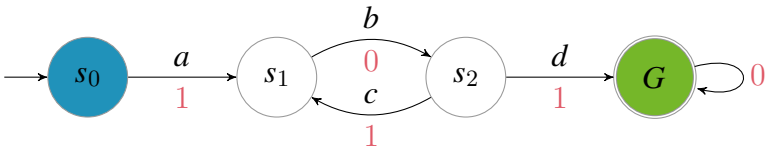
```
1:  M is the graph of the MDP
2:  proc MaximalExpectedReward(M, s : state, ε : float)
3:      too-low := True
4:      current-max := 4
5:      while too-low do
6:          too-low = False
7:          V = GLRTDP(s, ε)
8:          if V(s) == Positive-Infinity then
9:              return Positive-Infinity
10:         for each v in V do
11:             if v > current-max then
```

```
12:              too-low = True
13:              current-max = ⌈v + 1⌉ · 2
14:              break
15:      if ¬too-low then
16:          return V(s)
```

Cycles again require a special treatment. Before adding the next state to the current trial (line 18, Algorithm 2 and line 38, Algorithm 3), it has to be checked in a cycle detection procedure if this state closes an SCC in the current greedy graph. This happens in ELIM-CYCLE-MAX-REW() in line 20 of Algorithm 2 and in line 21 et seq. of Algorithm 3, independent of the reward accumulated in the SCC. If such an SCC is found, the maximal expected reward for this property can directly be set to ∞ because in the extreme case always this loop could be taken to accumulate arbitrary reward before reaching the goal. The following example shows such a situation.

Example 9: Calculating $ER_{max}(\Diamond Goal)$ with GLRTDP

To calculate the maximal expected reward accumulated on the way to the goal in the MDP depicted below, GLRTDP starts initializing s_0 with $current\text{-}max$ (in our implementation set to 2) and continues the trial with s_1 and s_2 which both get initialized with $current\text{-}max$. In the process of checking the successors of s_2, G is initialized with 0. Then greedily the next action is selected in s_2, which is c, because in a value update, it leads to a value of $current\text{-}max + 1$ for s_2, whereas taking action d gives a value of $0 + 1$. In line 18 of CHECK-SOLVED(), the cycle between s_1 and s_2 in the current trial is detected. The cycle elimination procedure called in line 21 detects that it is possible to loop between s_1 and s_2 forever without reaching the goal. That is why the value of the initial state s_0 is immediately set to ∞, the state is tagged as solved, and the procedure terminates with the correct result ∞ for $ER_{max}(\Diamond Goal)$.



Reward Minimization. For $ER_{\min}$ `min-rew` is set to *True* and the value function is initialized admissibly with ∞ for dead-ends and with 0 for all other states. Similar to the $ER_{\max}$ case, when adding the next state to the current trial, it has to be checked this time if it closes a zero-valued reward SCC which has to be eliminated because it has to be left eventually to reach the goal with minimal reward. This check if a zero-valued reward SCC is present in the trial, is performed in ELIM-CYCLE-MIN-REW() in line 27 of Algorithm 2, and also in line 27 of Algorithm 3.

This behavior of the algorithm is demonstrated in the following example.

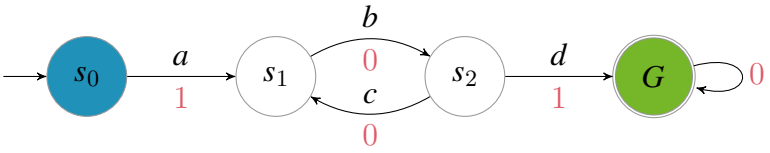
Example 10: Calculating $ER_{\min}(\Diamond Goal)$ with GLRTDP

To calculate the minimal expected reward to reach the goal in the quite simple MDP depicted below (which is not exactly the same as in Example 9 because of the reward structure), GLRTDP starts initializing s_0 with 0 and continues exploring the chain of successor s_1 and s_2, which also get 0 as initial value.

When exploring s_2 further, it is detected in line 18 of Algorithm 3 that action c, which would be chosen when greedily selecting the next state, because the reward for this action is 0, closes a cycle. This cycle has to be eliminated to enforce progress towards a potential goal state and still accumulate as few reward as possible. The trap elimination algorithm is called when reaching line 27 of the same procedure. It merges the states s_1 and s_2 because no reward is accumulated in between them.

Afterwards, a new trial of GLRTDP is started on the modified MDP, which does not contain the cycle anymore. It starts similar to the first trial and can proceed with exploring the states linearly until reaching G. Then, in the update procedure of CHECK-SOLVED(), the reward values are back-propagated and accumulated such that s_0 has a value of 2 in the end of this trial.

Because the states have not been ε-consistent in this trial, a third trial is performed in which all states can finally be tagged as solved and the algorithm terminates with the correct result $ER_{\min}(\Diamond Goal) = 2$.



4.1.3. Bounded Reachability Properties

Reachability probability and expected accumulated reward properties can be extended by step or reward bounds. $\mathcal{P}_{opt}(\mathcal{S}_U\ \mathcal{U}_{[l,u]}\ \mathcal{S}_*)$ is the extremal probability of reaching a goal state in $[l, u]$ steps or with accumulated reward in $[l, u]$. Expected reward properties with bounds are expressed analogously by $ER_{opt}(\mathcal{S}_U\ \mathcal{U}_{[l,u]}\ \mathcal{S}_*)$. Notably, and in contrast to the other properties considered thus far, for such bounded properties, memory-less policies can be outperformed by policies that are aware of the history regarding their past evolution, namely with respect to the number of steps left or the reward left for accumulation until exceeding the bound. So, we here work with memory-full policies (see Definition 10). For the purpose of bounded reachability properties, it has been shown that it is sufficient that

a policy can remember how many steps have already been made or how much reward has been accumulated [119, 120, 214, 257]. We make use of that as explained in the following and hence only need to change the procedures discussed above for unbounded reachability probabilities and unbounded expected accumulated rewards over reachability properties slightly. The changes that are needed for $\mathcal{P}_{opt}$ and ER_{opt} are similar.

Let us first look at step-bounded properties. For those, updating all state values synchronously in standard value iteration makes it possible to iterate only t times for properties with upper bound t [120]. Then, a step-dependent policy can be extracted. In heuristic search algorithms like FRET-LRTDP this is not possible because only the current greedy path is updated. In this case, a straightforward remedy is to encode a step counter into each state and consider all states for which the bounds regarding these counters are exceeded as dead-ends. Formally, one works in a derived MDP where states are enriched with counters and where states differing in the counter value are different and thus also the policy decision might differ for them (implying history awareness with respect to the original MDP). States which fulfill the reachability property *and* whose bound-counter lies in the target interval are considered as goal states. In our implementation we use the same variants of GLRTDP and trap elimination procedures like for the unbounded cases above and only add the bound to the procedure and the step counter to the states.

For reward-bounded properties the basic strategy is the same, except that the additional counters are now replaced by real-valued variables for tracking the reward accumulated so far. If the reward of the current policy exceeds the bound, the current state is considered as a dead-end. In either case (step or reward bounds), the derived MDP can be constructed in such a way that it is guaranteed to be finite-state (which is one of our early assumptions).

Since the overall procedures stay the same when adding bounds, the correctness and optimality proofs follow the respective same strategy.

MODYSH is the only tool of the MODEST TOOLSET which fully supports all variants of bounds w.r.t. step or reward bounds and interval types. All other tools do not treat step bounds at all and only support inclusive upper bounds.

4.2. Benchmarking and Scalability Study

After having studied the functionality of MODYSH, it is of great interest to investigate its performance and compare it to other state-of-the-art model checkers, which mostly implement completely different strategies. This will be done on a large set of diverse benchmarks with different structures coming from multiple domains. The diversity in the origin and structure of benchmarks is required to identify if approaches developed in one community, like MODYSH with strategies inspired by planning and heuristic search, also work well on benchmarks from another community whose structure potentially is different.

Using the benchmarks of the QVBS [132] (see Section 3.2) for our scalability study is obvious, since it is made for exactly this purpose and contains benchmarks of different structures from diverse domains, including planning problems as well as a lot of classical model checking benchmarks.

The setup of the QComp competition has been developed to compare different types of model checkers on QVBS benchmarks and all state-of-the-art model checkers have participated in the last QComp editions [54, 121]. This means, evaluating MODYSH in comparison to other model checkers in the same study setup is a natural way to go and enables the comparison of performances and plots to earlier editions.

For evaluation purposes, we used the setup from QComp 2020 *default often ε-correct track*, which used a precision of $\varepsilon = 10^{-3}$ and a timeout of 30 min, on an Intel Core i7-4790 CPU @ 3.60GHz with 32 GB RAM.

The benchmark set of QComp comprises, apart from other model types, 36 MDP instances from the QVBS. In addition, we added 58 *additional benchmark* instances from the QVBS to our case study to enlarge the number of MDP benchmarks, and thereby also the number of minimum reach and bounded properties. Furthermore, we wanted to test the tools on both smaller benchmarks, because many tools time out on the difficult QComp instances, as well as on considerably larger instances than the QComp benchmarks, with the intention to demonstrate the capabilities and benefits of MODYSH when only inspecting a fraction of the state space. Therefore, we scaled the models for the *israeli-jalfon* [159], *philosophers-mdp* [200], *pnueli-zuck* [225], *rabin* [229], and *wlan* [191] benchmarks up by parallelizing up to 100 automata for all of them except for wlan, for which 10 parallel processes are already enough such that only MODYSH is able to solve it. The QVBS contains only smaller instances of these benchmarks, i.e., we had to write our own scripts to scale them up further. For israeli-jalfon, the largest instance results in a state space size of $(2^{100}) - 1$, i.e., $1.268 \cdot 10^{30}$. For 100 dining philosophers the state space grows into the order of 10^{99} states, and for 100 parallel processes in pnueli-zuck and rabin it is in the order of 10^{100} and 10^{105} states, respectively. 10 parallel senders in wlan result in a size of around $7 \cdot 10^8$ states. We extracted the state space sizes with STORM.

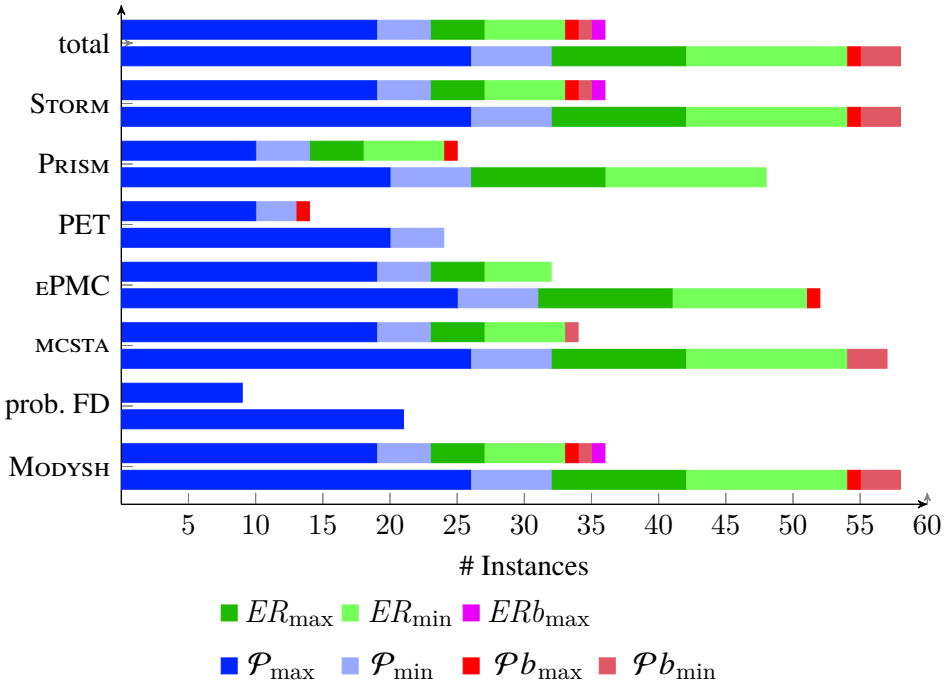


Figure 4.: Number of benchmark instances supported by tools per property type. (Upper bars: QComp, lower bars: additional benchmarks).

We compare the performance of MODYSH to the implementation of the basic FRET-π-LRTDP version in the planning tool Probabilistic FAST DOWNWARD [248]. In addition, the state-of-the-art model checkers EPMC [92, 124], MCSTA [119, 122, 129] of the MODEST TOOLSET [128], PET [48], PRISM [189], and STORM [77] take part in the evaluation. We contacted the authors of all these tools and asked for the newest version, i.e., improvements in other tools are also taken into account. With this setup, our evaluation is basically an update of the results form the often-ε-correct track of QCom 2020 [54]. The number of benchmark instances supported by each tool per property type, w.r.t. the tool's general functionality, are listed in Figure 4.

In the quantile plots in Figure 5, a point (x, y) indicates that the runtime of the xth fastest instance of the tool was y seconds. This allows comparing the overall performance of the tools. The benchmark instances are ordered independently for each tool depending on its runtime. The count of correctly solved benchmarks c (no timeout or error) and of supported instances s is given in the label as c/s.

The quantile plot at the top of Figure 5 shows that MODYSH is among the best three tools for a large number of instances of the QComp benchmarks. In addition, the strength of MODYSH is impressively demonstrated by the results on the additional benchmark set in the lower part of Figure 5. It clearly outperforms the other tools on the extremely large scaled benchmarks listed above because only a small fraction of the state space needs to be visited. MODYSH is able to solve seven benchmarks in less than $30s$ for which all other tools time out

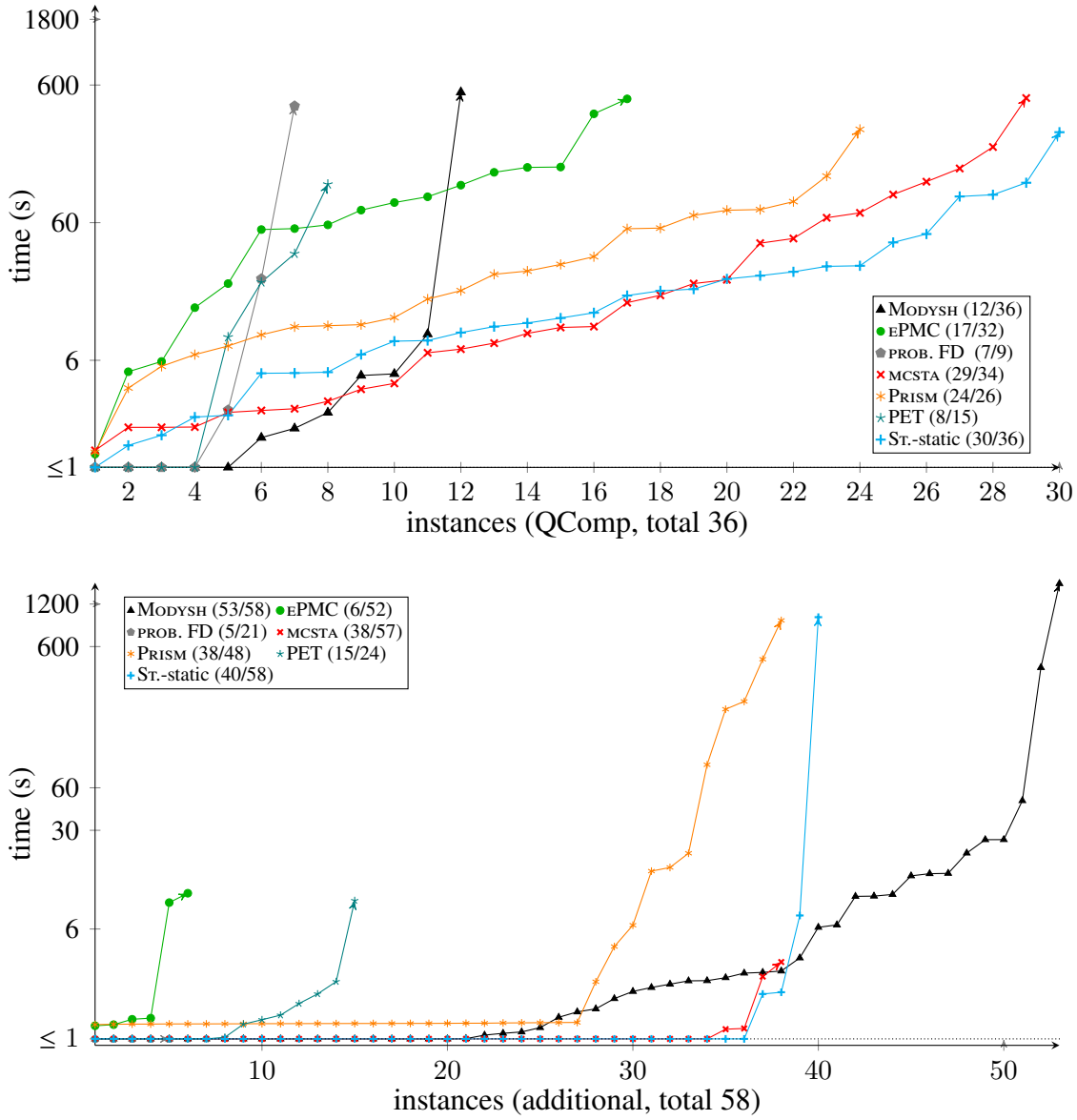


Figure 5.: Quantile plots for default tool versions in often ε-correct track.

or do not have enough memory. For five other models only one other tool is able to solve them. Details can be found in the interactive result tables for the QComp benchmarks[3] and for the additional QVBS benchmarks[4]. For the largest instances of philosophers, pnueli-zuck, rabin, and wlan only a few thousand states have to be visited in MODYSH, and only $1.7 \cdot 10^3$ for israeli-jalfon. That is the reason why MODYSH is often the only tool which is able to solve them. All these benchmarks have in common that they consist of the parallelization of automata of symmetric structure.

[3] https://depend.cs.uni-saarland.de/~klauck/results-qcomp-benchmarks/table_often-epsilon-correct.html
[4] https://depend.cs.uni-saarland.de/~klauck/results-additional-benchmarks/table_often-epsilon-correct.html

The results on both benchmark sets show that MODYSH is clearly able to compete with state-of-the-art model checkers and on certain MDP structures it is even able to quickly solve instances which no other tool is able to handle.

More detailed results can be inspected in Figure 6, 7, and 8, showing scatter plots comparing individual benchmark instances between two tools or a tool and the best of all other tools. A point (x, y) indicates a runtime of x seconds for the tool on the x-axis and a runtime of y seconds for the tool on the y-axis. This means, if the point lies above the diagonal line, the tool on the x-axis was the fastest. If the point lies above the dotted line, it was more than ten times faster. "TO", "ERR", and "INC" mean timeout, error, e.g., out of memory, and incorrect result, respectively. "n/a" means that the tool is not able to handle the benchmark instance. The number w of benchmark instances on which the tool on the x-axis outperformed the tool(s) on the y-axis and the number of in general supported instances s is given in parenthesis in the label in the form w/s. Note that, timeouts are not counted in w in case the other tool does not support the benchmark at all.

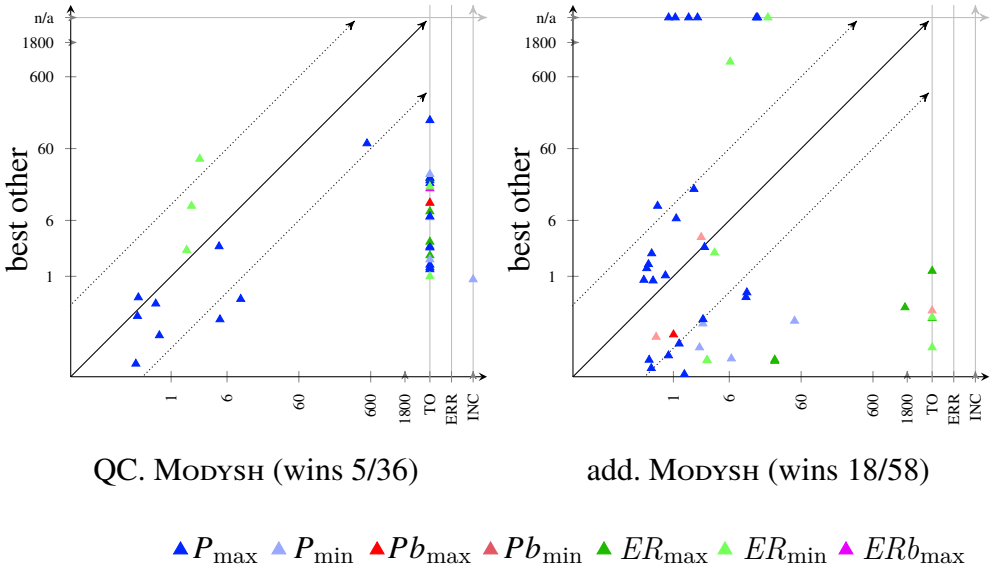


Figure 6.: Scatter plots MODYSH vs. best of all other tools (left: QComp, right: additional benchmarks).

Figure 6 shows the performance of MODYSH compared to the best result achieved by any of the other tools, i.e., the other tools are basically treated as one tool and always the best performance is taken. We see that MODYSH is able to compete with the other tools, especially on the additional benchmark set for which the results are depicted on the right.

This gets even more clear when having a look at the more detailed scatter plots in Figure 7, and especially in Figure 8 comparing MODYSH against each of the other tools separately on the QComp and the additional benchmark sets. MODYSH solves way more instances and

property types than probabilistic FAST DOWNWARD (first row, right in both figures), which is based on the same algorithms. It also supports more properties than MCSTA (first row, left in both figures), i.e., it improves the range of the MODEST TOOLSET and shows better performances on many instances, especially where MCSTA or various other tools (second and third row in both figures) are not able to deliver results at all.

This demonstrates the potential of the methods implemented in MODYSH. First, it improves the model checking performance of the MODEST TOOLSET in comparison to MCSTA on the same code base. Second, integrating these techniques specifically in STORM looks promising. If MODYSH was dominated by a competitor, e.g., on the QComp benchmarks (Figure 6 left and 7 bottom left), it was often outperformed by STORM. From QComp 2020 it is already known that STORM's code base is highly efficient and the performance is currently out of reach for other model checkers on most of the benchmarks. Implementing our approach in STORM would boost its performance even more, especially on the very large instances of the additional benchmark set.

Interactive result tables which enable a direct runtime comparison across tools and benchmark instances are available online for the QComp benchmarks[5] and for the additional QVBS benchmarks[6]. Furthermore, an artifact enabling the reproduction of all empirical results reported in this chapter is available online[7] [174].

[5] https://depend.cs.uni-saarland.de/~klauck/results-qcomp-benchmarks/table_often-epsilon-correct.html
[6] https://depend.cs.uni-saarland.de/~klauck/results-additional-benchmarks/table_often-epsilon-correct.html
[7] http://doi.org/10.5281/zenodo.4922360

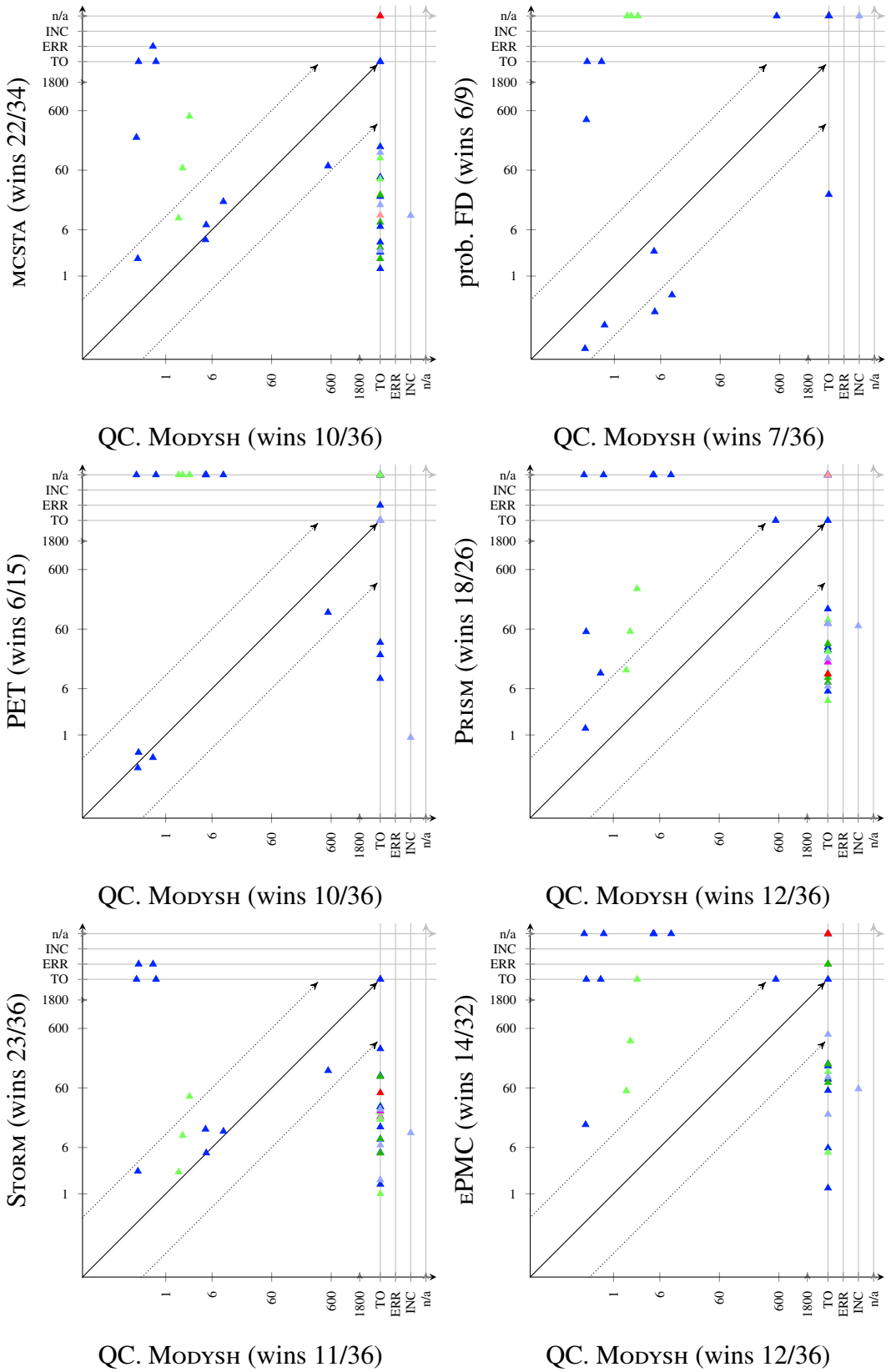


Figure 7.: Scatter plots for single tool comparison on QComp benchmarks. Legend as in Figure 6.

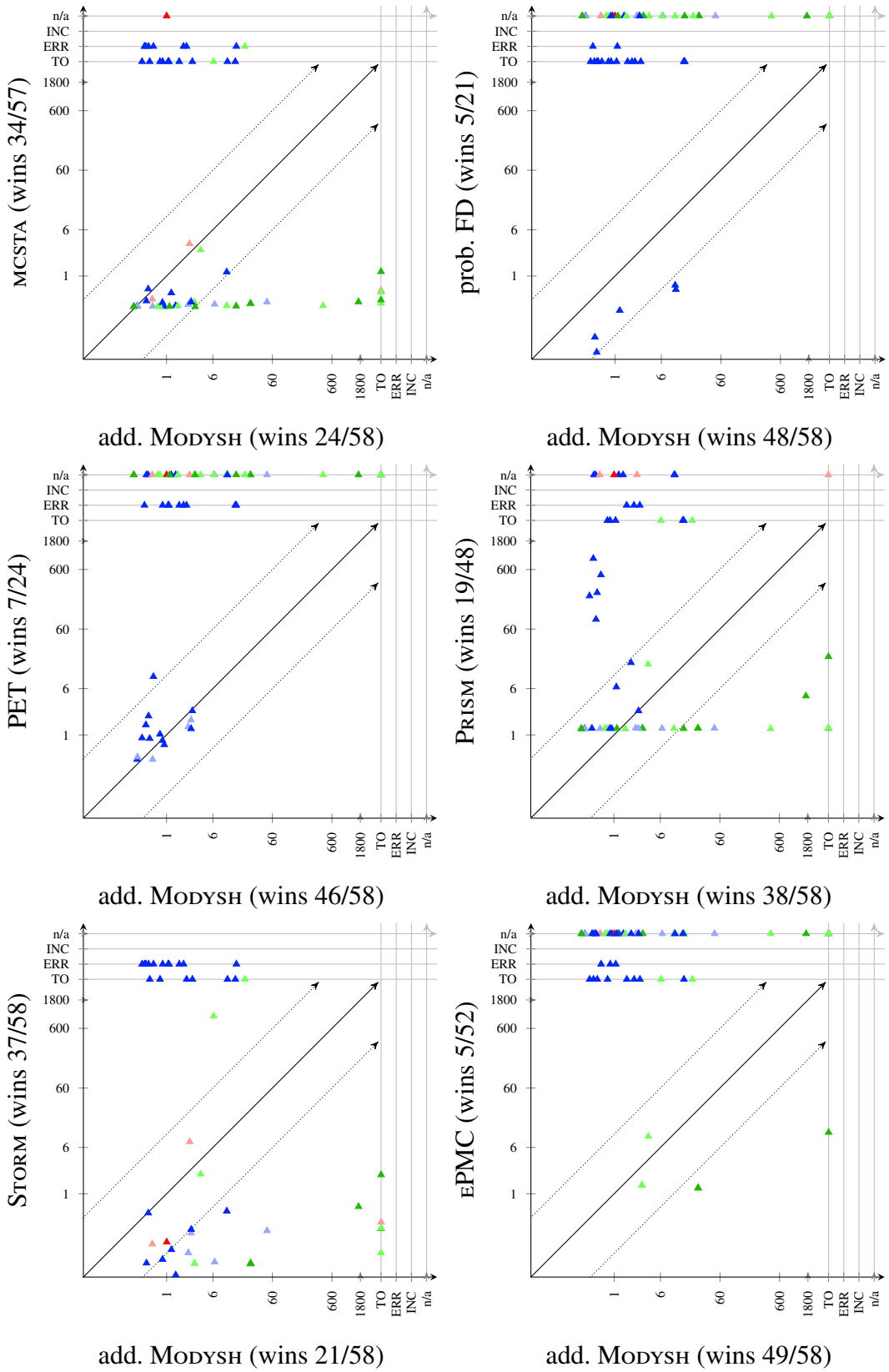


Figure 8.: Scatter plots for single tool comparison on additional benchmarks. Legend as in Figure 6.

4.3. Related Work

There have already been several endeavors trying to connect or combine planning and verification, mostly in two different categories: (i) by compiling input languages and hence, models, from one community into the standard of the other area, and (ii) by utilizing algorithmic approaches developed by the other community for own purposes. We start with a discussion of related work in the first area and afterwards summarize the related work regarding algorithms in both communities.

Compilations. It has been shown that system models represented as Kripke structures [181], e.g., in the SMV modeling language [206], can be translated into planning problems in PDDL. LTL safety, liveness, and fairness properties can then be checked using planning techniques for automated verification, especially heuristic search methods [6]. There exist further similar approaches solving translated LTL properties with classical planning [24, 218]. As has been shown by a compilation of the verification of safety properties in protocols modeled in the Promela language [150] into classical planning, simple safety properties, where there exists a finite counterexample, can easily be checked by heuristic search planners [83]. In contrast to the compilations between JANI and PPDDL done by the author of this thesis [175, 176], that compilation is not structure-preserving as it requires the introduction of several artificial state variables and actions for each individual Promela automaton transition.

This hints at what has already been noted in other works of the author, namely, that *"the more promising approach, though, is to instead port probabilistic heuristic search algorithms to model checking tools, to obtain a native realization tackling all the syntactic elements that cannot be easily compiled"* [176]. With the implementation of MODYSH, we realized this. In this sense, our research motivates a large area of cross-fertilization, pertaining to the exchange of algorithms. The integration of probabilistic heuristic search into probabilistic model checkers also provides access to the wealth of abstraction techniques developed by the latter community, e.g., BDDs (see Section 2.2.1).

Algorithms. There are quite a lot of works in the area of sharing algorithmic approaches between the communities which we discuss in the following together with works on algorithms relevant for our contribution or close to it.

As already described in Section 4.1, our algorithms in MODYSH are generalizations of well-known approaches used in the planning community for the purpose of cost-optimal planning. Of course, ideas behind heuristic search and planning have already been used in model checking. We highlight the parallels of works in both fields but also the differences to our approach.

Probabilistic Planning and Heuristic Search. As discussed, the original LRTDP work [45] is tailored to SSPs, with strictly positive action rewards (except at goal states). Kolobov et al. [179, 180] instead discussed GSSP problems for the usage of FRET on rewards in $\mathbb{R}$ but with lower bounds on negative rewards.[8] They showed that several MDP problems, including *MaxProb*, can be reduced to the GSSP problem class [179, 180], and that the respective properties can be solved using FRET with LRTDP. In MODYSH, we do not need to assume any of these problem classes, but restrict to positive and zero-valued reward structures.

A variant of the original FRET-LRTDP algorithms is available in the probabilistic version of FAST DOWNWARD [138] which is one of the classical progression planning systems based on heuristic search. It has been extended by Steinmetz et al. [248] for goal probability analysis, in more detail, for computing the maximal probability to reach a goal, but not for other problem classes not originally supported.

A quite different approach using heuristic search to solve reachability properties has been introduced by Trevizan et al. [256]. They introduced I-Dual, a heuristic search method capable of directly dealing with *MaxProb* MDPs, and thus making the FRET outer-loop obsolete. I-Dual's key to the support for more general MDPs is the use of linear programming instead of following the asynchronous value iteration scheme, which underlies most of the SSP heuristic search algorithms.

Probabilistic Model Checking. MODYSH is not the first work exploring probabilistic planning and heuristic search approaches for probabilistic model checking. For instance, heuristic search dynamic programming methods have been applied to MDPs, but for generating probabilistic counterexamples [7].

In addition, it has already been demonstrated that directed search algorithms can be exploited in explicit state model checking [84].

Another strand of works has been pursued in planning for MDP heuristic search [27, 45, 125] with a class of algorithms particularly tailored for reachability analysis. In this context, Brázdil et al. [48] applied and extended the heuristic search algorithm Bounded Real-Time Dynamic Programming (BRTDP) [205] for probabilistic model checking, showing promising results in their preliminary experiments.

Closer to our work, Křetínský et al. developed heuristics for initializing policies in policy iteration such that the computation time to solve long-run average reward properties on MDPs is reduced [185] with specific treatments of SSCs and maximal end components similar to

[8] Note that the LRTDP paper [45] presents the case where rewards are positive and should be minimized, whereas in the FRET paper [180] they are assumed to be mostly negative and should be maximized. By inverting the sign of rewards and switching minimization and maximization, the problem specifications can be aligned. That is the reason why our GSSP definition is not literally the same as in the FRET paper. In addition, note that the term proper is used with the meaning of almost-sureness in the paper on FRET.

the approach of MODYSH. Handling of end components has also been discussed in several other works [61, 73].

The PAC tool [13] uses asynchronous bounded value iteration techniques, with reinforcement learning, interleaved with guided simulation phases for permanent and transient trap elimination to perform reachability analyses with upper and lower bounds on MDPs and stochastic games with a focus on unknown probability functions.

Partial exploration of the state space for time-bounded reachability analysis of CTMDPs similar to our approach of building up trials through the state space and updating the state values on them has also been done [12]. To do so, a heuristic is needed on how to explore the system during trial construction. For that, a combination of BRTDP and Monte Carlo Tree Search has been devised with *MaxProb* objectives similar to ours [11]. In contrast to LRTDP, BRTDP works on upper and lower bounds for the property of interest, which means, one has to find a suitable initialization for the upper bound to boost the performance of the algorithm. For initializing the lower bound for *MaxProb*, we make use of the simple admissible dead-end heuristic during initialization described above, and we do not need an upper bound for LRTDP. Another difference is that LRTDP gives ε-convergence guarantees w.r.t. the Bellman equation on all states reachable with the current greedy policy, while BRTDP only requires a small residual on states reachable with significant probability. In addition, choosing the next action during exploration in BRTDP is biased towards states with larger differences between the lower and upper bound values. This difference is also taken into account for breaking exploration paths. In LRTDP, this is done based on the labeling procedure, which gives the algorithm its name. Such a labeling process is not performed in BRTDP. Furthermore, both algorithms, FRET-LRTDP and BRTDP, are originally only applicable to SSPs. We lifted the SSP condition for the FRET-LRTDP case in this chapter. The BRTDP methods [48] are made applicable to MDP structures with positive and zero-valued rewards by additionally triggering an end-component elimination after a certain number of steps to be safe. We do not introduce additional elimination steps periodically. Technical differences aside, this approach has only been applied to solve *MaxProb* properties.

Machine learning techniques have been exploited to verify reachability properties on MDPs using (1) BRTDP, and (2) delayed Q-learning for MDPs with limited information [48]. The techniques are also applicable to positive and zero-valued reward MDPs due to special treatments of end components and are implemented in PET (aka. PRISM-TUM), which is part of our evaluation in Section 4.2. In parts, the approach is close to ours for simple reachability properties, but restricted to that, and uses BRTDP instead of FRET-LRTDP. The paper explicitly mentions that so far no attempts have been made to adapt FRET-LRTDP methods in the context of probabilistic verification. With MODYSH we completely fill this gap.

As became clear in our empirical evaluation, heuristic search can be especially attractive for handling excessively large models. An entirely different approach to attack such problems is the use of external storage to slowly but exhaustively model check problem sizes that otherwise do not fit in memory [129].

4.4. Discussion

We introduced a heuristic approach to probabilistic model checking. It supports all established property types, except long-run averages and nested properties, on MDP structures with positive and zero-valued rewards based on LRTDP combined with FRET. To the best of our knowledge it has not been shown so far that an algorithm based on the core of FRET-LRTDP with some substantial changes is applicable across all these problems. We gave a correctness and optimality proof that the modified version solves all established property types except long-run averages and nested properties of MDP problems with positive and zero-valued rewards, i.e., a much broader spectrum of problems than the original works.

Originally, implementations of FRET-LRTDP as per Kolobov et al. have often only been used for *MaxProb* analysis in the planning community, e.g., in the probabilistic version of FAST DOWNWARD. Our new approach is implemented in MODYSH, which is part of the MODEST TOOLSET. With this tool we are now able to use enhanced probabilistic planning and heuristic search methods for model checking without translating model checking benchmarks to planning languages. We reported on a large empirical evaluation that has demonstrated the competitiveness of MODYSH relative to other state-of-the-art model checking tools. On very large state spaces our tool outperforms its competitors, demonstrating that planning techniques can indeed be used to enhance the performance and capabilities of model checkers. We expect further performance optimizations when exploring the trade-offs between memory usage and runtime even further or when implementing other heuristics known to work well in the planning community.

With our work we showed that the model checking community can benefit from collaborating and exchanging ideas with the planning community. We layed a cornerstone for further collaborations of the model checking and planning community, e.g., to treat also other automata types, or to investigate if an efficient sound version of our modified algorithms could be implemented.

5. Deep Statistical Model Checking

Neural networks (NNs) are taking over ever more decisions thus far taken by humans, even though verifiable system-level guarantees are far out of reach. Neither is the verification technology available, nor is it even understood what a formal, meaningful, extensible, and scalable testbed might look like for such a technology. The new approach, called *Deep Statistical Model Checking (DSMC)*, presented in this chapter, is an attempt to improve on both of the above aspects. As a testbed we use a family of formal models which are variants of the Racetrack benchmark. Due to the possibility to model random noise in the decision actuation, each model instance induces a Markov decision process as verification object. The NN in this context has the duty to actuate (near-optimal) decisions. From the verification perspective, the externally learnt NN serves as a determinizer of the MDP, the result being a Markov chain which as such is amenable to statistical model checking. The combination of an MDP and an NN encoding the action policy is the central attack point of what we call DSMC. While being a straightforward extension of statistical model checking, it enables to gain deep insights into questions like "How high is the NN-induced safety risk?", "How good is the NN compared to the optimal policy?" (obtained by exhaustive model checking of the MDP), or "Does further training improve the NN?". This means, the approach is usable for quality assurance done by end users or engineers, in system approval or certification, and for assessment of the infrastructure and algorithms used during training by learning experts.

We report on an implementation of DSMC inside the Modest Toolset in combination with externally learnt decision-making agents, e.g., NNs, demonstrating the potential of DSMC on various instances of the Racetrack model family, and illustrating its scalability in a detailed study as a function of instance size as well as other factors, like the degree of training. The DSMC implementation is part of MoGym, which is the first tool enabling the training of decision-making agents on formal models in combination with its quality assessment. We present MoGym with a focus on the DSMC functionality. In addition, we have a look at TraceVis, an interactive visualization tool for DSMC analysis on Racetrack. It fosters the analysis of DSMC results with the help of visualization, and forms a first step in combining model checking and visualization for the analysis of NN behavior.

In summary, our contributions around DSMC are as follows:

- We present Deep Statistical Model Checking, which statistically evaluates the connection of a decision-making agent and an MDP formalizing the problem context.
- We establish tool infrastructure for DSMC in MODES to connect to NNs and general decision-making agents, resolving the nondeterminism in MDP environments. Furthermore, we introduce MOGYM as a framework for training such agents and directly verifying and assessing them in one tool.
- We illustrate the use and feasibility of DSMC for quality assurance and learning pipeline assessment in Racetrack case studies.
- We demonstrate the scalability and performance of DSMC, along multiple dimensions, e.g., model size and number of training episodes as well as NN quality, in a huge, exhaustive study on scaled Racetrack benchmarks.
- We show how the analysis data we can generate and extract during DSMC evaluations can be visualized in the TRACEVIS tool implemented by our co-authors to get even more detailed insights into the quality of the NN. To do so, the DSMC implementation provides the option to extract the additional data needed for visualization in TRACEVIS.

The benchmarks for the case studies with their JANI models and all infrastructure, including our DSMC implementation in MODES, are archived and publicly available on Zenodo[1] [172]. This archive also includes the tool infrastructure of MOGYM. The benchmarks for the scalability study as well as the study framework in which the experiments have been performed, are also available. In addition, we provide all data produced for demonstration purposes of TRACEVIS and the tool itself in this archive.

Organization and Origins of the Chapter. We start with the discussion of the theoretical contribution of the whole Deep Statistical Model Checking approach in Section 5.1, which introduces NNs as MDP action oracles and describes the procedure of DSMC in detail. In addition, the tool infrastructure combined under the umbrella of MOGYM is presented briefly with a clear focus on the DSMC part of the tool. Subsequently, the presentation of the work on DSMC is split in three major parts.

First, Section 5.2 shows how DSMC evaluations can be applied in the Racetrack domain. A comprehensive case study on quality assurance in system approval is carried out and it is shown how the learning pipeline can be analysed and revised with the help of DSMC evaluations. Section 5.2.1 illustrates the use of DSMC for quality assurance by human

[1] http://doi.org/10.5281/zenodo.6362696

analysts, like end users and engineers, in system approval. Section 5.2.2 demonstrates the use of DSMC as a tool for engineers designing the NN learning pipeline. Section 5.2.3 evaluates the computational effort incurred by DSMC compared to a conventional SMC setting where the MDP policy is coded in the model itself.

Second, the scalability of the DSMC analysis is investigated and examined further in an extensive benchmarking study in Section 5.3.

Third, Section 5.4 briefly shows how the processed data collected during the DSMC analysis can be visualized in the TraceVis tool to gain even more insights into the decision-making agent's quality and deficiencies.

The related work centered around NN quality analysis and verification as well as visualization of NN behavior is summarized in Section 5.5. We conclude with a summary of the main contributions of our work and a discussion of future directions in Section 5.6.

The theoretical work and the introduction of the concept of DSMC (Section 5.1 and 5.2) is part of the FORTE 2020 publication [106] of Timo P. Gros, Holger Hermanns, Jörg Hoffmann, Marcel Steinmetz, and the author of this thesis. The author implemented the DSMC approach in the Modest Toolset, mainly in modes, and did all the benchmarking, scalability, and case study evaluations with the help of the heat maps designed for this purpose. The following sections show more evaluations of data and more details than provided in the paper. Timo P. Gros was responsible for the training of the used NNs.

The DSMC functionality and its integration in the MoGym [107] framework is presented in more detail in this thesis than in the paper. The other parts of MoGym, i.e., Momba Gym and the DSMC API of Momba, have been implemented by Maximilian A. Köhl. Timo P. Gros was responsible for the training and learning part.

The large scalability and performance evaluation in Section 5.3 is part of a submitted journal paper centered around DSMC [109]. This evaluation has been conducted by the author of this thesis and is shown in more detail than in the journal in the following.

We will also briefly summarize the follow-up works on DSMC co-authored by the author of this thesis. The TraceVis tool [103] considered in Section 5.4 has been implemented by David Groß and Stefan Gumhold. The author of this thesis assisted in the design and construction process by giving advice on which data could be visualized and which features would be helpful for a domain engineer and for quality analysis. In addition, the huge data set containing traces and meta-information about the DSMC analysis used for visualization in the tool has been generated and processed by the author of this thesis. To collect the data, the option to export TraceVis compatible meta-data from the DSMC analysis in modes has been used, which was implemented by the author. Furthermore, features, including new properties, were added to the Racetrack model by the author. A second part of the tool focussing on the

learning side by visualizing internal behavior of the NN and the learning process, especially the Q-values used during training [113], has been constructed with the participation of the author.

In addition, DSMC has been integrated into a feedback-loop to improve NN quality directly during the training process [105, 110]. In this paper, whose content is only summarized in the related work in Section 5.5, the author of the thesis supported their co-authors with information about the DSMC tool usage in the Modest Toolset and helped during the writing process.

5.1. Theoretical Contributions

The core theoretical contribution of DSMC is the functionality of *connecting MDPs and action oracles* which decide how to act in the environment defined by the MDP *to assess the oracles' quality with statistical model checking*. In essence, the role of action oracles is very close to that of an *action policy* (see Definition 10): Oracles decide in each *situation* which *option* to pick next. If we consider the *situations* (inputs $\mathcal{I}$) as the states $\mathcal{S}$ of a given MDP, and the *options* (outputs $\mathcal{O}$) as actions $\mathcal{A}$, then the *action oracle*, which could for instance be a neural network or any other trained decision-making agent, is a function $\sigma : \mathcal{S} \to \mathcal{A}$. Indeed, this is what the reinforcement learning process in Q-learning and other approaches delivers naturally (see Section 2.4).

Observe that an action oracle can be cast into an action policy except for a subtle problem. Action policies only pick actions from $\mathcal{A}(s)$, thus applicable at the current state s, while action oracles may not (cf. Definition 10). A better fitting definition would constrain oracles to always return an applicable action.

Yet it is not clear how to guarantee this for NNs. It would be necessary to add a safety constraint to the network such that it does not return an output ranking an inapplicable action the highest. But it is an open question how this can be implemented. Even for linear multi-classification, the hard constraints required to guarantee action applicability lead to non-convex optimization problems [58, 100, 192, 193]. An easy fix would use the highest-ranked applicable action instead of the NN classifier output itself. For our purposes however, where we want to analyze the quality of the NN oracle, it makes sense to explicitly distinguish inapplicable actions as a form of low quality.

If an oracle returns an inapplicable action, then no valid behavior is prescribed and in that sense the system can be considered *stalled*.

Definition 24: Action Oracle Stalling

Let $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{T}, s_0 \rangle$ be an MDP, and $\sigma \colon \mathcal{S} \to \mathcal{A}$ be an action oracle. We say that $s \in \mathcal{S}$ is *stalled* under σ if $\sigma(s) \notin \mathcal{A}(s)$.

To accommodate for stalling, we augment the MDP upfront with a fresh action † available at every state. This action is chosen upon stalling, leading to a fresh state ‡ with only that action to continue. So $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{T}, s_0 \rangle$ is transformed into $\mathcal{M}^{\ddagger} = \langle \mathcal{S} \cup \{\ddagger\}, \mathcal{A} \cup \{\dagger\}, \mathcal{T}', s_0 \rangle$, where for each state s, $\mathcal{T}'(s, \dagger) = \delta_{\ddagger}$ and otherwise $\mathcal{T}'(s, a) = \mathcal{T}(s, a)$ wherever the latter is defined.

Definition 25: Oracle Induced Markov Chain

Let $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{T}, s_0 \rangle$ be an MDP, and let σ be an action oracle for $\mathcal{M}$. Then the Markov chain C^{π} induced by σ is the one induced in $\mathcal{M}^{\ddagger}$ by the memory-less action policy π defined by $\pi(s) = \dagger$ whenever s is ‡ or $\sigma(s) \notin \mathcal{A}(s)$ (i.e., stalled), and otherwise by $\pi(s) = \sigma(s)$.

In words, the oracle induced policy fixes the probability distribution over transitions in each state to that of the chosen action. If that action is inapplicable, then the chain transitions to the fresh state ‡ which represents stalled situations.

Overall, C^{π} is a Markov chain induced by the policy π that uses σ as an oracle to determinize the MDP $\mathcal{M}$ whenever possible, and stalls otherwise. With σ, for instance implemented by a neural network, we can use statistical model checking on C^{π} to analyze the NN behavior in the context of $\mathcal{M}$. This analysis has the potential to deliver deep insights into the effectiveness of the NN applied, allowing for comparisons with other policies and also with optimal policies, the latter obtained from exhaustive model checking.

From a practical perspective, an important remark is that in the definitions above and in our implementation of DSMC described below, the inputs to the NN action oracles are assumed to be the MDP states $\mathcal{S}$ or often only relevant parts of it. This captures the scenario where the NN takes the role of a classical system controller, whose inputs are system state attributes, such as program variables. Often the controller only has a partial view on the environment and the system, e.g., an autonomous car can only observe a certain area of its surrounding.

More generally, the connection from the MDP model to the NN input may require an intermediate function f mapping $\mathcal{S}$ to the input domain of the NN. This is in particular the case for NNs processing image sequences, like in vision systems in autonomous driving.

This advanced form of connection remains a topic for future work. It lacks the crisp nature of the problem considered here.

But there are also developments in the autonomous driving domain, which go exactly into the opposite direction and thereby are quite close to our approach. With image segmentation, semantic segmentation, and instance segmentation done by NNs, especially for autonomous driving, there is already research on filtering and abstracting the image sequences of the scene which are usually processed by the NNs [89, 241, 255]. Objects and their properties in an image have to be reduced to the parts and information relevant for making decisions in the scene, i.e., the information have to be described more abstractly. In such scenarios, the decision-making entity is then not working on the image anymore but on a selected set of information relevant for making decisions, like it is the case in DSMC.

5.1.1. DSMC Core-Implementation in MODES

Deep Statistical Model Checking takes as input a trained decision-making agent and an MDP modeling the same environment the agent operates in. Usually, the decision-making agent is an NN. Since conceptually, there is no difference for the DSMC procedure between the various forms of decision-making agents, we will continue to talk about NNs in the following, but want to highlight that all of the work can be done with arbitrary decision-making agents or general oracles, and especially that the implementation of DSMC supports them. This means that we also often do not explicitly distinguish between these terms.

The NN is assumed to be trained externally, e.g., by deep reinforcement learning with the help of *PyTorch* [227], a leading Python library for NN learning, prior to the analysis in which it is combined with the MDP. To experiment with this concept, we have developed a DSMC implementation inside the MODEST TOOLSET [128] (see Section 3.4), as an add-on to the statistical model checker MODES [51]. For the comparison of the policy induced by the decision-making agent to the optimal policy we also make use of the explicit-state model checker MCSTA [119, 122]. We implemented two novel options in MODES to resolve nondeterminism during simulation of sample traces. The first option is called `NN` and is usable to evaluate a neural network. The second option is called `Oracle` and works with an arbitrary external procedure connected via a socket communication.

With the new options in place, every time the next action has to be chosen in a nondeterministic state in the MDP while simulating a trace in SMC, MODES provides the current model state to the action oracle, which then returns the chosen action to MODES. In this way, the SMC procedure can, e.g., connect to an external NN serving as an action oracle from MODES's perspective.

At the implementation level, connecting to standard NN tools is non-trivial due to the programming languages used. The MODEST TOOLSET is implemented in C#, whereas standard

NN tools are bound to languages like Python or Java. Our observation to overcome this issue was that a seamless integration is not actually required. Standard NN tools are primarily required for NN *training*, which is computationally intensive and requires highly optimized code. In contrast, implementing our NN oracle requires only NN *evaluation*, i.e., calling the NN on a given input, which is easy – it merely requires to propagate the input values through the network. We thus implemented the NN evaluation directly in the Modest Toolset's code base, as part of our extension. This variant is used for the experiments shown in Section 5.2.

The main variant of the DSMC implementation presented in the following sections uses *TorchSharp* [115], a .NET library that provides C# access to the library that powers PyTorch. In this variant, the NN access is simplified by exporting a JSON file describing the NN structure, and containing the NN weights and biases learned using standard NN tools. Our extension of modes reads that file, and uses it to reconstruct the same NN with TorchSharp, for the usage in the DSMC evaluation. When the oracle is called, modes connects via TorchSharp to the NN. The performance of DSMC stays the same for both variants of NN querying, but the connection to TorchSharp makes the tool handling easier for the user. The concrete implementation details and command line options to call modes for DSMC are described in the next section together with MoGym, the tool for training decision-making agents on formal models and directly perform DSMC on them.

5.1.2. DSMC Integration in MoGym

MoGym [107] is an integrated toolbox enabling the training, analysis, and verification of decision-making agents based on formal models in one common tool infrastructure. Given a formal model of a decision-making problem in JANI format and a reach-avoid objective, MoGym enables (a) training a decision-making agent for the reach-avoid objective using reinforcement learning (RL) techniques directly on the formal model, and (b) rigorously assessing the quality of such decision-making agents using DSMC. With these two parts, MoGym bridges the gap between formal methods and reinforcement learning and thereby helps to solve the issue discussed in Section 1.3, that so far, environments of decision-making agents during training are typically specified implicitly in the form of simulation code making it hard to check consistency properties on them. MoGym gives the opportunity to define a formal, mathematically precise and unambiguous description of the *training environment* to get a principled understanding of the power of RL algorithms and of the properties of a specific learned agent with the help of DSMC.
As depicted in Figure 9, MoGym consists of the following parts:

- *Momba Gym* is implemented on top of Momba [178], a Python toolbox for dealing with quantitative models from construction to analysis centered around JANI. With

Momba Gym, a training environment for reinforcement learning techniques can be built from a formal model together with a reach-avoid objective given by a JANI file using the explicit state space exploration engine of Momba. Momba Gym implements and extends the OpenAI Gym API [49], which is the widely used standard interface for connecting environments to different (deep) reinforcement learning algorithms [82, 116, 147, 217, 239]. The OpenAI Gym API enables the comparison of reinforcement learning algorithms and fosters the development of new techniques, thereby connecting to existing work in the reinforcement learning community.

- The *DSMC API* is also implemented on top of Momba. This is a Python API to access the DSMC functionality of the Modest Toolset.
- *DSMC* is implemented in mcsta of the Modest Toolset. With the DSMC functionality, it is possible to statistically model check the probability with which formal properties, i.e., reach-avoid objectives, are fulfilled when resolving nondeterminism during statistical evaluation in the formal model under investigation by querying the decision-making agent.

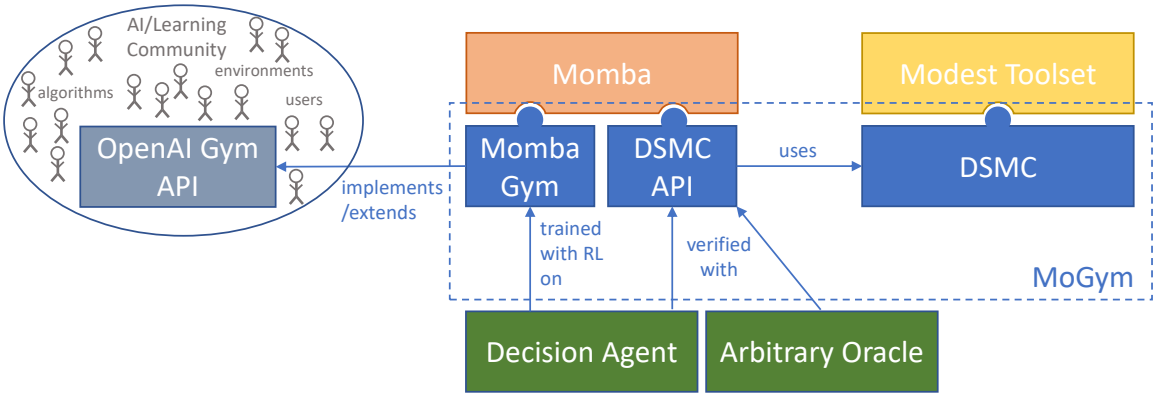


Figure 9.: The architecture of MoGym and its components [107].

The diagram in Figure 9 shows how the different parts of MoGym are connected to each other. First, a decision-making agent can be trained on a formal model and a reach-avoid property, specified in a JANI model, against the OpenAI Gym API by using Momba Gym with arbitrary reinforcement learning techniques implemented by the user of MoGym. Afterwards, the trained decision-making agent can be verified w.r.t. the property of interest by invoking the DSMC API, which makes use of the DSMC extension of the statistical model checker modes. Alternatively, the training step can be skipped, and an arbitrary external oracle can be checked by modes.

With this infrastructure, MoGym opens the JANI benchmark set for reinforcement learning. Employing a formal, precise, well-specified, and unambiguous model of the environment

instead of an informally programmed one, bears the promise of enabling rigorous assessment of decision-making agent properties via formal methods. The integration of the OpenAI Gym API in MoGym together with the DSMC functionality contributes an efficient feedback mechanism for improving in particular reinforcement learning algorithms.

Implementation Details. We assume that a decision-making agent controls a single automaton in the automaton network of the formal model describing the environment, i.e., the agent resolves the nondeterminism of this automaton. To train a network for this automaton, the OpenAI Gym API and thereby Momba Gym requires the definition of an *action space* and an *observation space*. In response to receiving observations from the observation space, the agent takes a decision from the action space. During the DSMC evaluation of the trained decision-making agent, exactly these two spaces have to be used, too. The inputs to the trained decision-making agent are part of this observation space and the outputs fed back to the SMC simulation engine are part of this action space. This means, the DSMC functionality in modes has to communicate with the decision-making agent via the same action and observation space as used by the Momba Gym API during training. To enable the usage of general JANI MDP models, the action space and observation space have to be extracted from the model. Depending on the model, there are multiple ways to do so. Momba Gym, and also DSMC in modes, implement multiple strategies for this extraction. For the action space, edges of the controlled automaton can be selected by index or by label. For the observation space, either (i) only global variables, (ii) global and local variables of the controlled automaton, or (iii) all variables can be made observable.[2] Other strategies can easily be added. The default setting is to make global variables the only observable ones.

Whenever the agent selects a decision in response to an observation, the decision is first mapped to an edge of the controlled automaton, and then to a transition of the network (for details on the structure of JANI models, see Section 3.1). If there are other entities in addition to the controlled automaton (e.g., adversaries modeled in another automaton of the automata network) nondeterministically influencing the environment of the decision-making agent, the user receives a warning and the remaining nondeterminism is resolved uniformly.[3] This should be taken into account when inspecting the results. Technically, this approach would extend to a multi-agent setting where different agents resolve the nondeterminism in different parts of the model.

After taking the respective transition, Momba's state space exploration engine during learning and modes during the DSMC analysis continue simulating a trace through the model until a state is reached where the agent can make a decision again.

[2]For more details about those strategies see https://momba.dev/gym/.
[3]That is, each of the remaining nondeterministic options is considered equiprobable.

Usage of MoGym and modes for DSMC. The DSMC API of Momba implements two functions, one for verifying arbitrary decision-making agents accessible via a Python function, and one for verifying PyTorch neural networks. Both functions rely on the DSMC functionality in the statistical model checker modes, which accepts both forms of decision entities, and both functions return the value of the reach-avoid property calculated by the model checker.

An arbitrary Python function mapping observations to decisions can be checked with MoGym by executing:

```
gym.checker.check_oracle(oracle, model, automaton)
```

Here, `oracle` is the Python function implementing the decision-making agent. Note that this is not limited to trained decision-making agents in any way. Any arbitrary Python function with an appropriate signature can be used. `automaton` is the automaton the decision-making agent controls within the model specified. All properties specified in the JANI model are checked. The optional arguments `actions` and `observations` can be used to specify the strategy for the extraction of the action and observation space (i.e., by index or by label, and which variables to observe, see above).

While `check_oracle` involves executing Python code, a more efficient approach is available when the decision-making agent is a PyTorch neural network. In this case, the neural network can be directly verified with `check_nn`:

```
gym.checker.check_nn(nn, model, automaton, property_name)
```

To this end, e.g., in case the NN consists of a sequence of layers, the structure of the NN and its characteristics have to be communicated. The function `check_nn` extracts these layers from the provided neural network `nn` and exports them in a JSON-based format. The neural network is then loaded by modes with TorchSharp and used for model checking without calling back into the Python runtime. With the help of TorchSharp, modes supports networks with arbitrary dimensions and activation functions. Convolutional networks are in principle also supported, though we did not experiment with those.

Alternatively to the DSMC API provided by Momba, it is also possible to invoke modes on the command line to check an NN in MoGym's JSON format or to connect to an arbitrary decision-making agent via a socket connection. This could then be any program taking the information of the observation space as input and sending an action decision back.

The DSMC evaluation of an NN oracle with modes can be called directly via:

```
modest.exe modes path/to/model.jani -R NN -NN path/to/NN/JSON -A
 controlled-automaton
```

The general oracle can be called over a socket communication on a given port by executing:

```
modest.exe modes path/to/model.jani -R Oracle -SOC host:port -A
 controlled-automaton
```

`controlled-automaton` is the name of the automaton controlled by the NN in the JANI model. If the automaton is not named explicitly in the model, MODES assigns a default name $autN$ to it where N is the number of the automaton in the JANI file. MODES starts counting at 0 from top to bottom in the file. The new option `--max-run-length-as-end` treats reaching the maximum length of a simulation run as if a dead-end state would have been reached, i.e., such a situation is treated in the simulation as if the goal were not reachable. This flag can be used in situations where it is necessary to avoid that the simulation is aborted if the maximum run length is reached and not all properties are decided. However, it should be checked how such a phenomenon has to be interpreted in the specific context of the model under investigation and how the SMC results are effected by this modification in the respective case.

`--observations-local-global` is the option to make global and local variables of the controlled automaton observable. To see all variables of all automata and the global variables, `--observations-omniscient` has to be set.

5.2. Application: DSMC Evaluations on Racetrack

After having discussed the technical details of the DSMC functionality and its implementation, we are now ready to apply DSMC in practice. As previously outlined, we consider Racetrack as a formal, simple, and discrete, yet highly extensible approximation of real-world phenomena that involve randomness and decision-making. In this section we demonstrate all facets of the usage of DSMC on the Racetrack benchmark. But first, we briefly summarize how the NNs we assess later with DSMC are obtained and what structure they have.

Learning Neural Networks for Racetrack. For the sake of realistic empirical studies, we have drawn on established NN learning techniques to obtain NN decision-making agents for the Racetrack case studies. Here, we briefly summarize the main design decisions. Notably, DSMC is entirely independent of the concrete learning process, depth, and shape of the NN employed.

NNs for Racetrack are learnt for a specific map (cf. Figure 3), with the inputs being 15 integer values, encoding the two-dimensional position, the two-dimensional velocity, the distance to the nearest wall in 8 directions, the x and y differences to the goal coordinates, and the *Manhattan* goal distance, which is the absolute x- and y-difference summed up. Actions to accelerate in the 9 possible directions are encoded as classification outputs, i.e.,

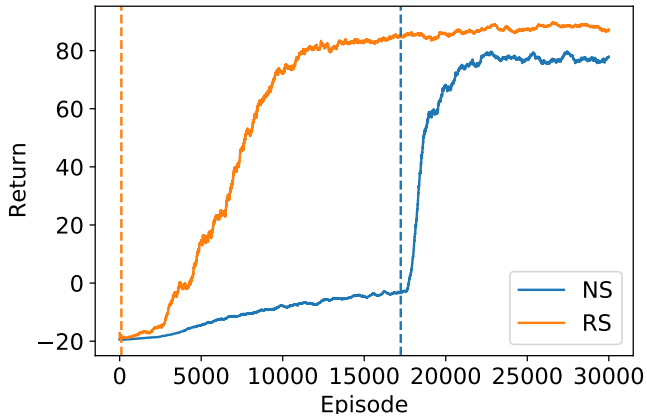


Figure 10.: Training plots of a policy trained in the normal start setting and a policy trained in the random start setting on the Ring map. The curves depict the sliding mean of the last 500 observed returns during training. The dashed lines indicate the first time a cell on the goal line was reached, i.e., the first time a positive return was observed in one episode.

the output layer consists of 9 neurons, where each neuron is semantically associated to one of the possible directions.

A crucial design decision is the learning objective, i.e., the rewards used in deep Q-learning. We set the reward for reaching a cell on the goal line to 100, and for crashing into a wall to values within $[-50, -20]$, depending on the map shape and the task to solve. We used a discount factor of 0.99 (see Equation 2.2) to encourage short trajectories to the goal. This arrangement was chosen because, empirically, it resulted in an effective learning process [102]. Such a procedure to choose the learning parameters is common in the learning community. With higher negative rewards for crashing, the policies learn to prefer not to move or to move in circles. Similarly, smaller negative rewards make the learnt policies prefer to crash quickly. Using a discount factor yields better learning performance, but does not match the overall Racetrack setup. This exemplifies that the choice of objectives for learning is governed by learning performance. Both hyper-parameters and numeric parameters, such as rewards, typically require fine-tuning orthogonal to, or at least below the level of abstraction of, the qualities of interest in the application, but this is out of scope of this thesis.

We experimented with a range of NN architectures and hyperparameter settings, the objective being to keep the NNs simple while still able to learn useful oracles in our Racetrack benchmarks. The NNs we settled on have the above described input and output layers, and two hidden layers each of size 64. All neurons use the ReLU activation function $f(x) = \max(0, x)$.

NNs are learnt in two variants: First, starting alternatingly on all cells on the start line, so called *normal start (NS)*, vs. second, starting from a random point anywhere on the map,

called *random start (RS)*, each with initial velocity 0. The random start variant turned out to yield much more effective and robust learning. Intuitively, RS seems to be a more challenging task as there is more that the policy needs to learn. Still, for NS it takes the policy a long time to reach the goal at all, while with RS this happens more quickly yielding earlier and more robust learning also farther away from the goal.

Consider Figure 10, which depicts the training curve of two policies for the Ring map (see Figure 3), one trained in the NS setting and the other in the RS setting. The training plot shows the sliding mean of the returns achieved during the training episodes. For the RS mode, the goal line is already reached shortly after the training starts, as indicated by the dashed orange line, and the return of the policy increases steadily until a plateau is reached, where the policy only improves slightly. In contrast, for the NS mode, the goal line is reached for the first time after about 17 000 episodes, indicated by the blue dashed line, and therefore just then receives the first positive reward. Thus, the policy can only start to learn how to reach the goal after these 17 000 episodes, which explains the abrupt increase of achieved returns afterwards.

Note that, the average values of achieved returns in the end of training cannot directly be compared to each other. As the episodes trained with random start in average are shorter, as they regularly start closer to the goal line, the achieved returns are discounted less and therefore are higher (see Equation 2.2).

The insights in the quality of the learned NNs, and the differences between normal and random start gained by checking the learning curve and training parameters will become even more clear and detailed with the help of DSMC as shown in the following.

DSMC Case Studies in Racetrack. We now demonstrate the Deep Statistical Model Checking approach for NN policy verification through case studies in Racetrack. There are a variety of use cases for DSMC, pertaining to end users and domain engineers alike. These can be grouped in two main categories, which we will inspect in the following.

- **Quality Assurance.** DSMC can be a tool for end users or engineers in system approval or certification regarding safety, robustness, absence of deadlocks, or performance metrics. The generic connection to model checking furthermore enables the comparison of decision-making agents to provably optimal choices on moderate-size models: taking out the agent, the original MDP results and can be submitted to standard exhaustive probabilistic model checking. In our evaluations, we use the probabilistic explicit-state model checker MCSTA [128] for this purpose.

- **Learning Pipeline Assessment.** DSMC can serve as a tool for NN engineers designing the NN learning pipeline in the first place. This is because the DSMC analysis can

reveal specific deficiencies in that pipeline. For example, we show how heat maps can highlight *where* the decision-making agents are unsafe. Furthermore, we exhibit cases where NN decision-making agents turn out highly unsafe despite this phenomenon not being derivable from standard measures of learning performance. Such problems would likely have remained undetected without DSMC.

Throughout the experiments, we use MODES with an error bound $\mathcal{P}(error > \varepsilon) < \delta$, where $\varepsilon = 0.01$ and $\delta = 0.05$, i.e., a confidence of 95%. We set the maximal run length to $10\,000$ steps. Unless otherwise stated, we set the slippery-noise level in Racetrack, i.e., the probability of action failure, to 20%. The NN oracles are learnt by training runs starting anywhere on the map, i.e., in a random start setting. We also illustrate how DSMC can highlight the deficiencies of the alternate approach starting on the start line only. All experiments were run on an Intel Core i7-4790 CPU @ 3.60GHz (4 cores, 8 threads) with 32 GB RAM and a 450 GB HDD.

5.2.1. Quality Assurance in System Approval

The variety in abstract property specification gives versatility to the quality assurance process. This is important in particular because, as previously argued in Section 2.4, the relevant quality properties will typically not be identical to the objectives used for NN learning. In the Racetrack example, NN learning optimizes the expected return subject to fine-tuned reward and discount values. For the quality assurance we consider the crash probability and the goal probability in JANI, namely $\Diamond\, crashed$ ("eventually crashed") for the former, and $\neg crashed\ \mathcal{U}\ goal$ ("not crashed until reaching goal") for the latter. Further properties of interest could be, e.g., bounded goal probability (How likely is it that we will reach the goal within a given number of steps?), expected number of steps to goal, or risk of stalling.

The strength of DSMC analysis is that in addition to pointing out *that* an NN oracle has deficiencies, it is also possible to show *where*, i.e., in which regions of the MDP state space $\mathcal{S}$. In cyber-physical systems it is natural to use the spatial dimension underlying $\mathcal{S}$ for systematizing the analysis and visualizing its result. This delivers not only a yes/no answer, but an actual quality report. We illustrate the results through the use of heat maps over the Racetrack road map, designed for this purpose by us.

Figure 11 shows quality assurance results for crash probabilities in all the Racetrack benchmarks considered in this thesis introduced in Section 3.3, using for each the best NN oracle from reinforcement learning, i.e., the one yielding highest returns in the training plots. We are aware that for highly critical and dependable software systems, an acceptable error rate is often (much) lower than the values we present for Racetrack in the following evaluations. Our aim was not to optimize the decision-making agents as much as possible but

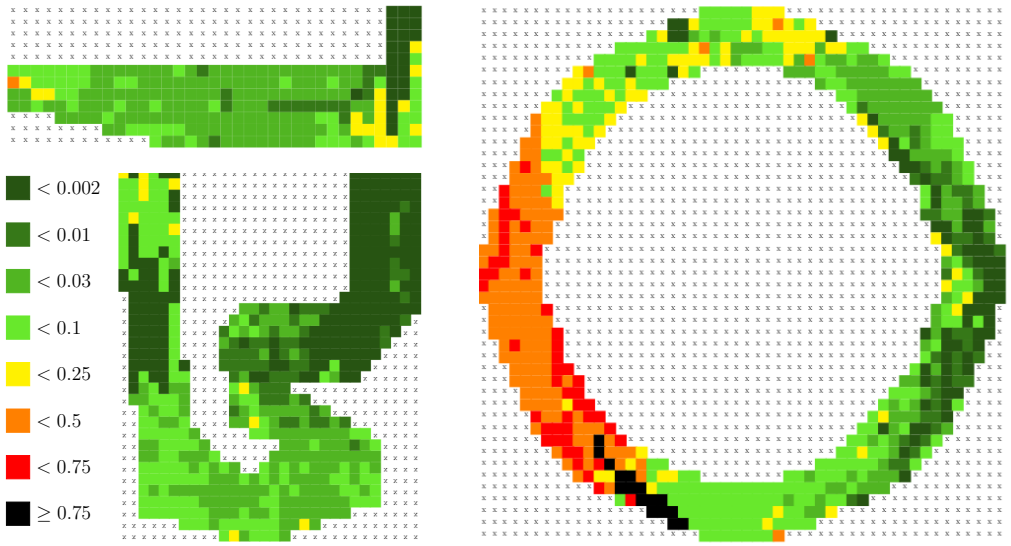


Figure 11.: Heat maps of NN induced crash probabilities for all Racetrack benchmarks.

to present illustrative results demonstrating the potential of the DSMC approach. In addition, due to the noise probability, even an optimal policy cannot reach the goal with certainty (see Section 3.3). The heat maps use a simple color scheme to illustrate the analysis results for human analysts. Similar color schemes will be used in all plots below.

From the displayed DSMC results, quality assurance analysts can directly conclude that the NN oracles are relatively safe in Barto-small (left top) and in Barto-large (left bottom) with crash probabilities mostly below 0.1; but not on Ring (right) where crash probabilities are above 0.25 on significant parts of the map.

Generally, the crash probability increases with the distance to the goal line. Some interesting subtleties are also visible, for example that crash probabilities are relatively high in the left-turn before the goal in Barto-small.

The next results in Figure 12 illustrate the quality-assurance versatility afforded by DSMC through an analysis quite different from the previous one. Assume that the NN analysts here decide to evaluate the goal probability ($\neg crashed\ \mathcal{U}\ goal$), (a quality stronger than not crashing because that may be achieved by idling). Apart from the original setting, they consider a stress-test scenario where the road is significantly more slippery than during NN training, namely 50% instead of 20% noise. They finally decide to compare with optimal goal probabilities, computable with the probabilistic model checker MCSTA, so that they can see whether any deficiencies are due to the NN, or are unavoidable given the high amount of noise.

The figure shows the outcome for Barto-large. One of the deficiencies is immediately apparent, the NN policy does not pass the stress test. Its goal probability matches the optimal

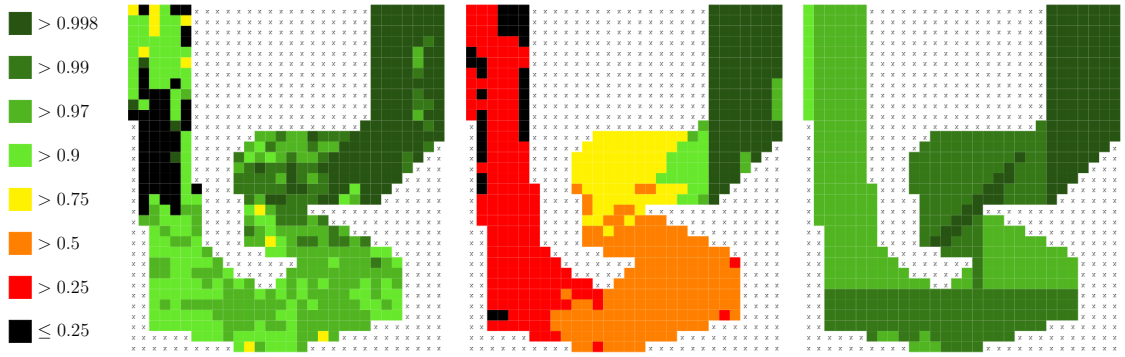


Figure 12.: Goal probability of NN oracle on the Barto-big benchmark trained and executed with 20% noise (left) vs. stress-test executed with 50% noise using the same NN (middle) vs. optimal policies obtained by exhaustive probabilistic model checking with 50% noise (right).

values only near the goal line, and exhibits significant deficiencies elsewhere. Striking is the small area close to the start line, where the goal probability increases slightly from below 25% to $25 - 50\%$, which is caused by the high amount of noise leading the car out of the area where the NN learned to drive in circles or to stop on the track without continuing to drive further. Based on these insights, the quality analysts can now decide whether to relax the stress-test (after all, even optimal behavior here does not reach the goal with certainty), or whether to reject these NN polices and request re-training, e.g., with a focus on specific areas.

5.2.2. Learning Pipeline Analysis and Revision

DSMC can yield important insights not only for quality assurance, but also for the engineers designing the NN learning pipeline in the first place. For this purpose, there are two distinct scenarios:

(i) The engineers run the same success tests as in quality assurance, and re-train if a test is not passed.

(ii) The engineers assess different properties of interest to the learning process itself (e.g., expected length of policy runs), or assess the impact of different hyperparameter settings.

In both scenarios, the DSMC analysis results point to specific state space regions that require improvement. This can be directly operationalized to revise the learning pipeline by starting more training runs from states in the critical regions.

Figures 11 and 12 above have already demonstrated (i). Next, we demonstrate (ii) through two case studies analyzing different hyperparameter settings.

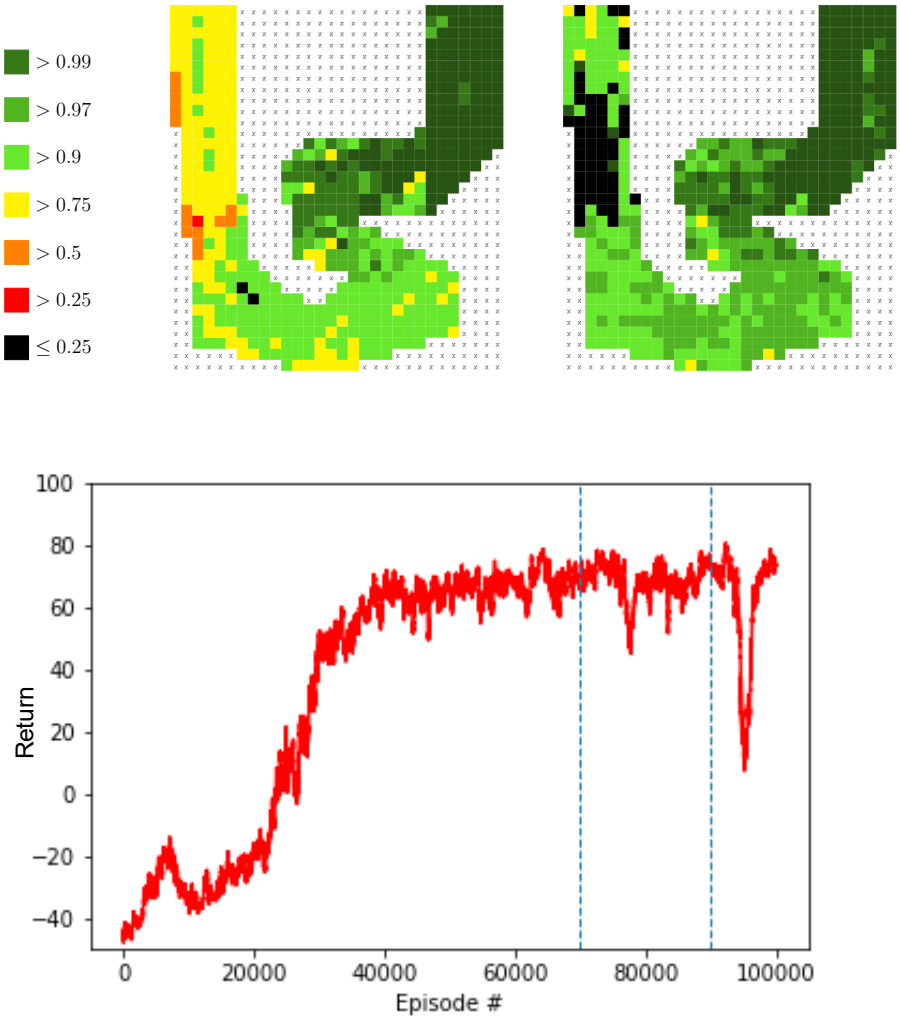


Figure 13.: Goal probabilities on the Barto-big benchmark, for NN oracles learnt over $n = 70\,000$ (top left), and $n = 90\,000$ (top right) training episodes, together with the Q-learning curve (bottom).

Our first case study, for which the results are depicted in Figure 13, analyzes the number n of training episodes as a central hyperparameter of the learning pipeline. The only information available in deep Q-learning for the choice of this hyperparameter is the learning curve, i.e., the expected return as a function of n, depicted on the bottom of the figure. Yet, as our DSMC analysis here shows, this information is insufficient to obtain reliable policies. In Barto-big, the highest return is obtained after $n = 90\,000$ episodes. From $n = 70\,000$ to $n = 90\,000$, the return slightly increases. Yet we see in Figure 13 that the additional $20\,000$ training episodes, while increasing overall goal probability, lead to highly deficient behavior in an area near the start of the map, where the goal probability drops below 0.25. If provided with that information, the engineers can focus additional training on that area, for instance.

In our next case study, we assume that the NN engineers decide to analyze the impact of starting training runs on (a) the start line vs. (b) random points anywhere on the map.

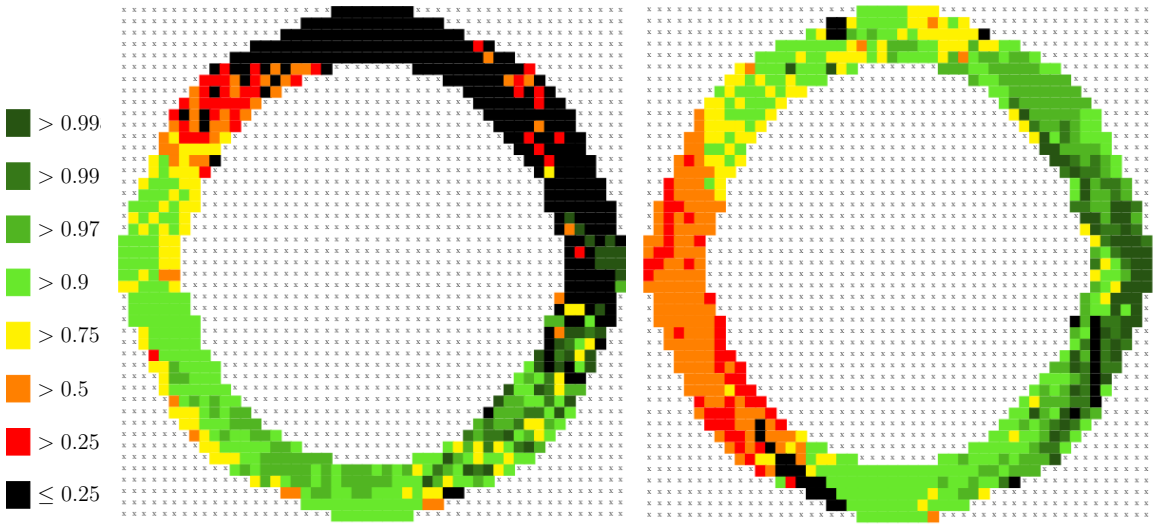


Figure 14.: Goal probabilities in Ring for NN oracles where training was carried out with reinforcing runs from (a) the start line only vs. (b) from anywhere on the map.

Figure 14 shows the results for the Ring map, where they are most striking. In variant (a), the top part of the Racetrack map was completely ignored by the learning process. Looking into this issue, one finds that during training the first solution happens to be found via the bottom route. From there on, the reinforcement learning process has a strong bias to that route, preventing any further exploration of other routes.

Phenomena like this are highly detrimental if the learnt policy needs to be broadly robust across most of the environment. The deficiency is obvious given the DSMC analysis results, and these results make it obvious how the problem can be fixed, namely by focusing the (re-)training on the more difficult areas. But neither can be seen in the learning curves, like the one for this Ring map example for random and normal start in Figure 10.

5.2.3. Computational Effort of the Analysis

As discussed, it can be highly demanding or infeasible to verify the input/output behavior of even a single NN decision episode, and that complexity is potentially compounded by the state space explosion problem when endeavoring to verify the behavior induced by an NN oracle. Deep Statistical Model Checking carries promise as a lightweight approach to this formidable problem, as no state space needs to be stored and on the NN side it merely requires to call the NN on sample inputs. In addition, it is efficiently parallelizable, just like SMC. Yet (1) the approach might suffer from an excessive number of sample runs needed to obtain sufficient confidence, and/or (2) the overhead of NN calls might severely hamper its runtime feasibility.

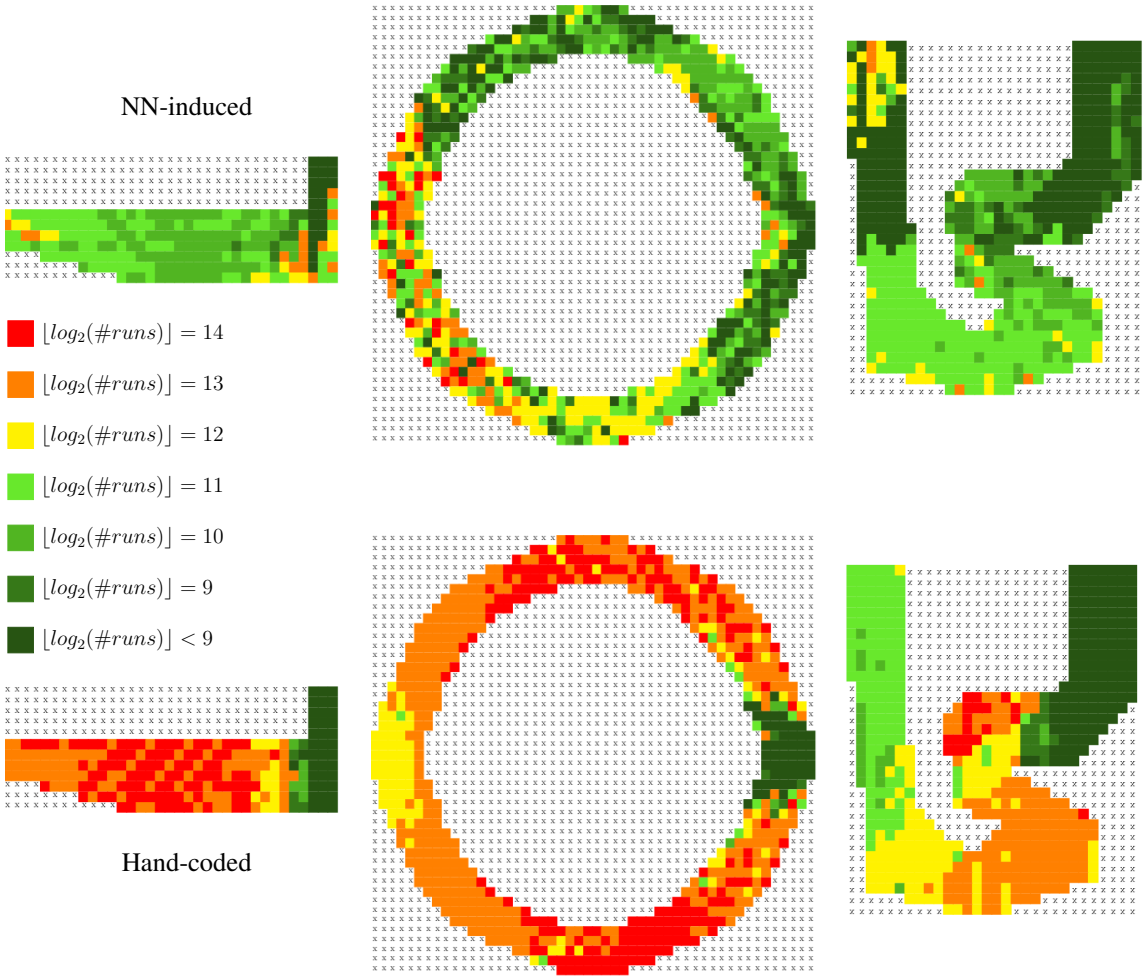


Figure 15.: Heat maps showing the computational effort needed by DSMC, measured by the number of sample runs performed by MODES to analyze goal probability for each map location. Results are shown for the policies induced by our learnt NN in the top row, vs. a simple hand-coded policy (see text) at the bottom. Each cell on the map shows $\lfloor \log_2(\#runs) \rfloor$.

Figure 15 shows data regarding (1). We compare the effort for analyzing our NN policies to that required for analyzing a conventional hand-coded policy that we incorporated into our JANI models.

The hand-coded policy implements a simple reactive controller that brakes if a wall is near, and otherwise accelerates towards the goal. In more detail, the controller is directly implemented in the JANI model and resolves nondeterminism using shared action synchronization. The controller's overall goal is to navigate the car towards the closest goal cell according to Manhattan distance. To avoid crashes, the controller tries to ensure that the car's speed is low enough to come to a complete stand before hitting a wall. In the case when keeping the current speed of the car would lead to a crash, the car is decelerated accordingly. Otherwise, the controller chooses acceleration values in x and y directions according to

the difference between the car's current position and that of the targeted goal cell. If the resulting speed vector is still considered to be safe, i.e., satisfies the aforementioned property, then the acceleration vector is executed. Otherwise, the speed is left unchanged. When the straight line connecting the car's start position and the targeted goal position contains walls, following this policy will eventually cause the car to stop in front of a wall. In that case, the controller moves the car alongside the wall as long as progress towards the goal becomes possible again.

As the heat maps show, the effort for checking the hand-coded policy with MODES is higher. This is due to a tendency to more risky behavior in the hand-coded policy, resulting in higher variance which leads to more sample runs needed to gain the required statistical confidence.

We do not provide an extra figure for the goal probabilities of the NN because for Barto-small and Ring the heat maps look exactly the same as in Figure 11 when interpreting the colors with the legend for goal probabilities given in Figure 12. This is because $1 - \textit{goal probability}$ results in the same value as the *crash probability* but *only if* no stalling occurs in the traces generated by DSMC *and* if all traces reach a goal cell or crash at some point, i.e., do not end up driving a cycle trajectory. This is the case for Barto-small and Ring. For Barto-big the goal probability heat map is depicted in Figure 12 on the left. Comparing it to the respective crash probability heat map in Figure 11, one can see that especially in the first half of the map there are a lot of start states in which stalling occurs or neither a goal is reached nor a crash happens.

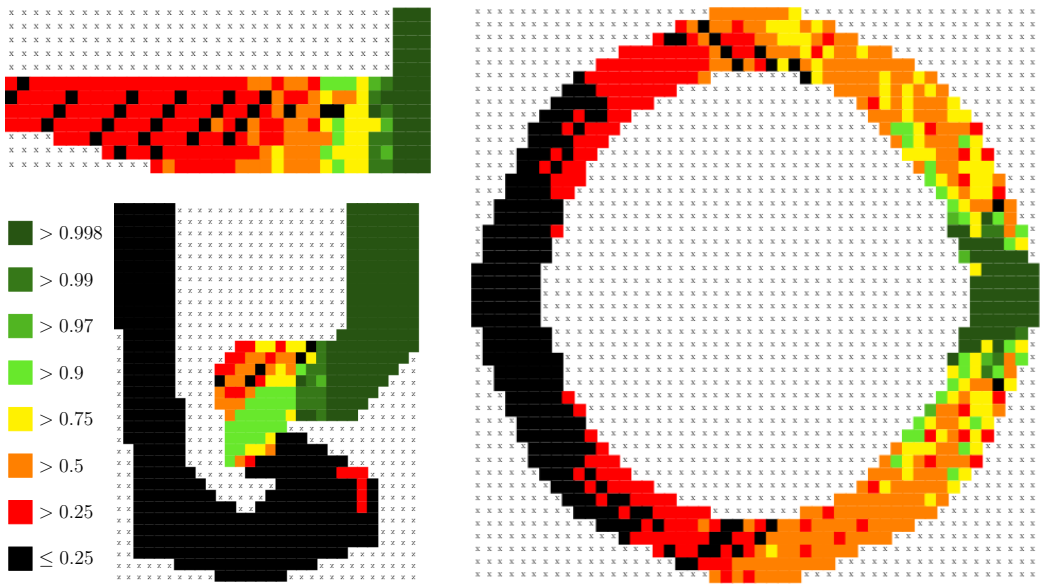


Figure 16.: Heat maps of goal probabilities of the hand-coded controller for all Racetrack benchmarks.

The more risky behavior of the hand-coded policy is clearly visible in the heat maps showing their goal probability on all maps in Figure 16. The probability to reach the goal is in large areas only moderately worse than that of the best NN policies. But especially near the start cells in Ring, and in the first half of Barto-big, it is clearly worse. In Barto-big the goal reachability probability rises once the vehicle is beyond the curves when coming closer to the goal, and when it is not needed to navigate narrow curves anymore. This behavior is obviously caused by the controller's strategy to navigate very closely along the wall if the direct straight connection to the goal is blocked. The NN policies instead learned to keep a safety distance to the walls to prevent crashes when noise occurs.

Regarding (2), the runtime overhead for NN calls is actually negligible in our study. Each call takes between 1 and 4 ms. There is an additional overhead for constructing the NN prior to the analysis, but that takes at most 6 ms. Deeper analysis of DSMC scalability is done in the next section.

5.3. Benchmarking and Scalability Study

As the previous chapter showed, the DSMC approach enables detailed insights into questions like "How high is the NN-induced safety risk?" or "Does further training improve the NN?". The case studies so far were of limited size though, leaving open the question how the DSMC approach scales up. This section sheds light on that question by presenting a profound scalability study, systematically extending the previous case studies.

We present a thorough analysis of DSMC performance and its scalability, depending on multiple dimensions as a function of model size as well as other factors, like the degree of NN training affecting the NN quality. For this study, we extend the Racetrack case studies originally considered to systematically scale the size of the map. In addition, the effect of the training degree on the effort needed in DSMC is measured over different stages of reinforcement learning.

5.3.1. Scalability Study Design

In our scalability study, we concentrate on the Barto-big Racetrack map. The scaled JANI benchmarks in our case studies are generated with the same tools in the same formalism as the benchmarks used in the previous sections, but in a setting with a noise probability of 10% to make the task easier on very large maps. The objective in the scalability context is calculating the maximal goal reachability probability per state.

Scaling Racetrack. DSMC runtime is influenced by multiple factors, including map shape, policy quality, and of course the map size, i.e., the model size of the system or scenario under inspection. We scale the Barto-big track shape up by using finer discretizations, thereby effectively making the track larger to navigate. This scaling approach is simple and canonical, and facilitates a detailed, direct comparison across different sizes. Specifically, we scale by a factor N where every cell in the original map is replaced by a square of N^2 smaller cells. The map growth thus is quadratic in N, i.e., the original map has a size of 30×33 cells, while with $N = 2$ we get 60×66 cells, with $N = 3$ we get 90×99 cells, and so on.

Study Setup. This size scaling may impact the performance of DSMC, measurable, e.g., in the number of sample runs or the runtime, in several ways:

(i) Analyzing policy behavior by starting from every map cell (with initial velocity 0), the number of calls to DSMC equals the number of cells after scaling.

(ii) The MDP becomes larger and individual policy runs become longer, which may affect the number of sample runs required to obtain the desired statistical confidence in the analysis result and the execution time needed per run.

(iii) The quality of an NN oracle – its ability to successfully navigate the map – may affect the number of sample runs required in DSMC.

We now summarize the results of our study examining these effects. We consider (iii) first, as it turns out to influence DSMC performance quite substantially, thus being important to understand as a prerequisite for our scalability study. We analyze (iii) as a function of training degree, which is of interest in itself if one is interested in analyzing the NN oracle under training at different stages, which is a natural application of DSMC. Given our insights into (iii), we then turn to our study of (i) and (ii) using NN oracles of comparable quality.

All experiments were run on 5 virtual machines having an AMD EPYC Processor at approximately 2.5 GHz using Ubuntu 18.04 with 8 vCPUs and 16 GB RAM. A total of $114\,129$ processing hours have been invested in this study, i.e., reproducing already a fraction of these results takes a lot of time. All scripts and infrastructure used for the scalability study is available online[4] [172] together with all other supplementary material on DSMC.

5.3.2. Performance as a Function of Training Episodes

To evaluate the impact of training strength on the runtime of DSMC, we extracted networks for the Barto-big map shape, e.g., depicted in Figure 3, after 5k, 10k, 15k, 20k, and 25k training episodes for scaling factor $N = 1$, and for $N = 2$ additionally after 30k, 35k, 40k,

[4] http://doi.org/10.5281/zenodo.6362696

and 45k training episodes, because for the latter training takes longer. Figure 17 summarizes the results by averaging.

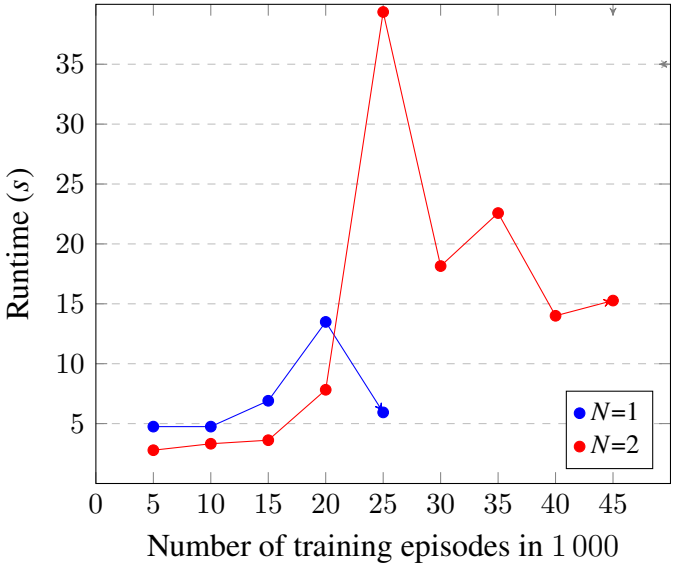
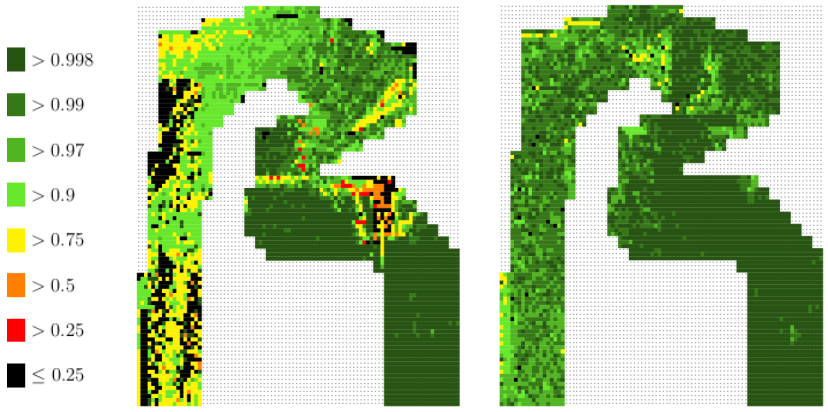


Figure 17.: Average runtime of DSMC per map cell over training episodes.

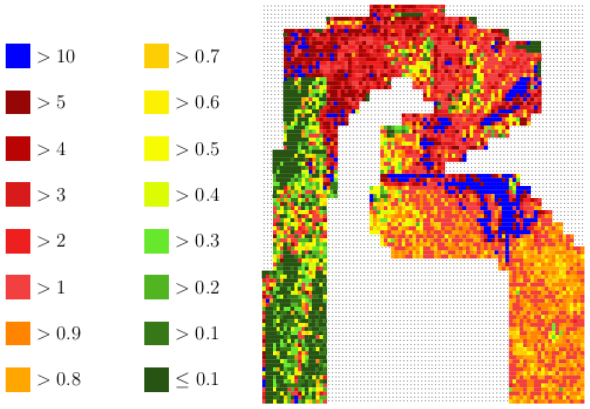
DSMC exhibits an easy-hard-easy pattern as the training degree grows, because low- and high-quality networks can be checked quickly whereas NNs with rather undecided action outputs need more time in the statistical model checker. This is characteristic, since for other scaling factors N the same pattern emerges. Indeed, the pattern is easily explained and makes sense. Little-trained NN oracles tend to crash quickly and thus are easy to analyze. Strongly trained policies tend to reach the goal reliably with little variance. The performance of DSMC is more stable in terms of the runtime difference among cells in the same area on the map. This again results in high statistical confidence after relatively few sample runs. The hard cases lie in the middle, where the NN oracle exhibits high variance between runs that crash and ones that reach the goal. The reason is that the oracle is more undecided or has a more risky driving behavior, necessitating more analysis effort, because more SMC runs are needed to obtain a result with the statistical guarantees.

To corroborate these findings, let us have a closer look at the dependency between oracle quality and DSMC runtime. Fixing $N = 3$, we examine two NN action oracles σ_{bad} and σ_{good} of different quality (same number of training episodes but different hyperparameters), analyzing their goal probability and DSMC runtime locally, specific to different regions of the map, in difference to the global analysis considered in Figure 17. Figure 18 shows the data.

In Figure 18 (a) and (b), we depict for two different NN action oracles σ_{bad} and σ_{good} for each map cell the goal probability when starting the policy from that cell with an initial velocity of 0. This goal probability was determined by running DSMC on the respective



(a) (b)



(c)

Figure 18.: (a), (b): Goal probability per cell for $N = 3$ with a bad-quality NN oracle σ_{bad} (a) vs. a good-quality NN oracle σ_{good} (b). (c): DSMC runtime difference quotient $\frac{\sigma_{bad}}{\sigma_{good}}$ per cell.

MDP state. In Figure 18 (c), we depict the difference in runtime between (a) and (b), namely the quotient of DSMC runtime for σ_{bad} over DSMC runtime for σ_{good} on a cell-by-cell basis. Briefly put, dark green to yellow colors mean that DSMC on σ_{bad} takes less time than DSMC on σ_{good}, orange to light red means that both are analyzed in similar runtime, darker red to blue means that σ_{bad} takes more time to analyze up to a factor of more than 10. The exact color-coding legend can be found on the left of (c).

The heat maps clearly show the effect of local policy quality on DSMC runtime. Near the starting line, where σ_{bad} typically does not reach the goal, and crashes frequently and quickly (black stripes and cells in (a)), σ_{bad} is much easier to analyze than σ_{good} (dark green stripes and cells in (c)). This changes drastically in the first curve of the map, where σ_{bad} exhibits high variance and becomes much harder to analyze than σ_{good}. As we move closer to the goal, this latter phenomenon gradually diminishes, except for the last curve in which

σ_{bad} has again lower goal reachability probabilities (light green, yellow, orange cells) than before, resulting in higher DSMC runtimes (dark red, blue cells) because of higher variance. At first glance, one would expect that in the black cells where the quality of σ_{bad} is really low, also its DSMC runtime should be much lower than for σ_{good}, similar to the behavior in the beginning of the track. But since we are now closer to the goal and the quality of σ_{good} is much better than in the beginning of the track, its runtime is also much lower and in the case of Barto-big even lower than the runtime for σ_{bad} in these cells. Additionally, the runtimes of DSMC for cells so close to the goal are in general quite low, which makes the measures quite sensitive.

5.3.3. Scalability Over Instance Size

We now turn to the main purpose of our study, examining DSMC scalability as a function of instance size. Given the above insights, in this part of the study we only compare NN oracles of similar global quality, as measured by the training return they achieve and a DSMC quality analysis. Furthermore, to account for variance in local policy quality (which is impossible to avoid), we train and analyze five different NN oracles for each value of N. In addition, if not stated otherwise, the results show averages over all cells on the map factoring out complexity source (i) from above, i.e., that the number of calls to DSMC varies with the map size because one call of DSMC is performed per map cell, which is a trivial phenomenon here due to our complete coverage of cells on the track.

Figure 19 (a) displays the size of the MDP state spaces, given in terms of the number of states to be considered by the analysis. The plots in (b) and (c) present our main scalability result as functions of the map size, in terms of (b) average DSMC runtime per map cell (initialized with velocity zero) and (c) average number of sample runs per map cell. We detail these results for the most demanding policy (max), and for the easiest policy (min) at each scale, together with the average (avg).

The model sizes shown in (a) indicate that the analyzed MDPs are quite non-trivial, with millions of states already for $N = 1$ and $N = 2$, and going up to almost 150 million states for $N = 5$. Against this background, (b) clearly shows that the effort needed by DSMC increases linearly as a function of map size, e.g., because the run length grows. This is corroborated by (c) which shows that the required number of sample runs barely has any tendency to increase with increasing map size at all. The scaling curve is dominated instead by the amount of variance across different policies.

We also ran these scalability experiments with lesser training, choosing low and middle quality policies following related work [59] as ones that deliver 20% and 50% of the maximal achieved return for the good policies during training, respectively. The results for these settings depicted in Figure 20 and 21, respectively, show similar tendencies as the ones

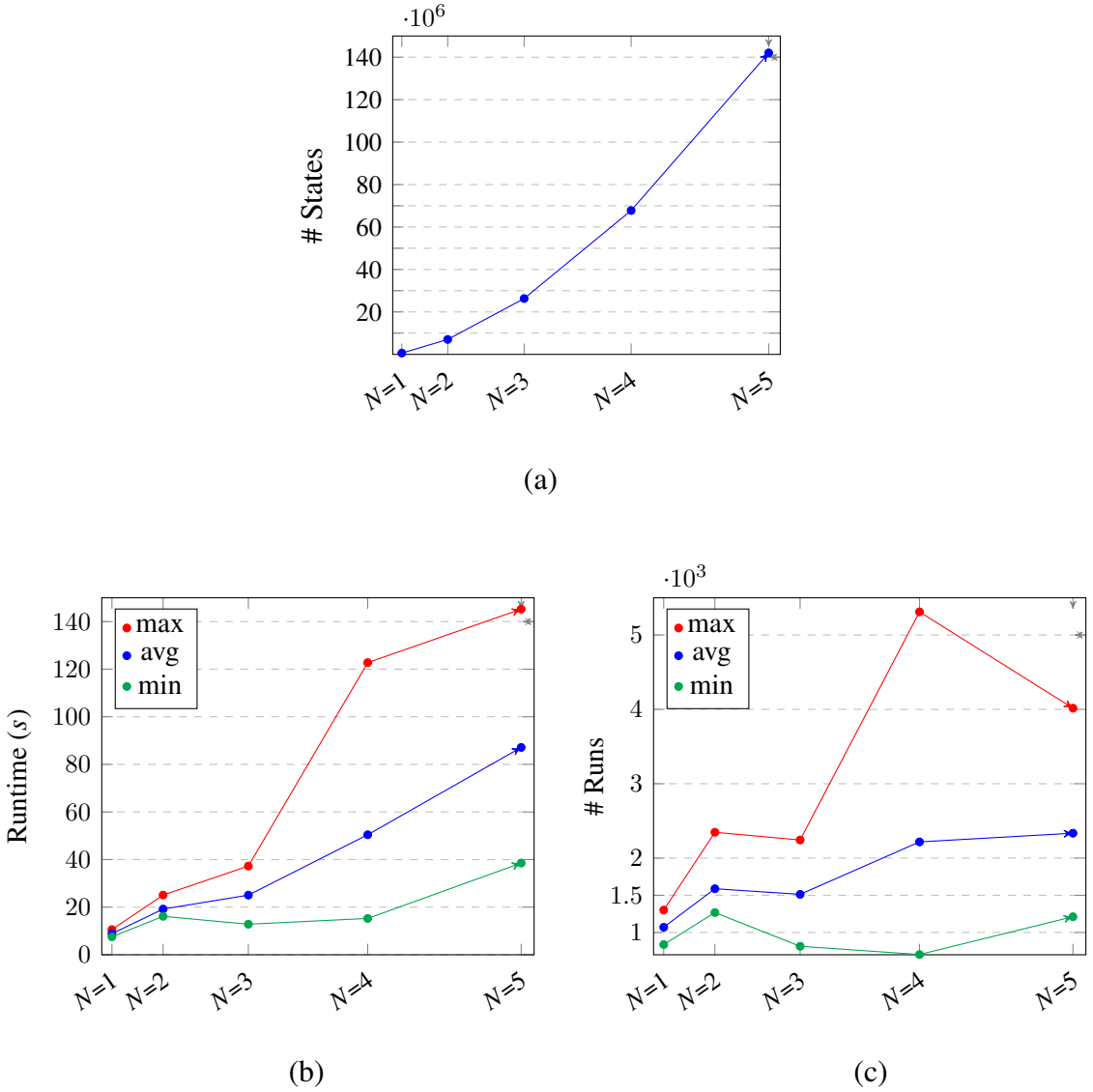
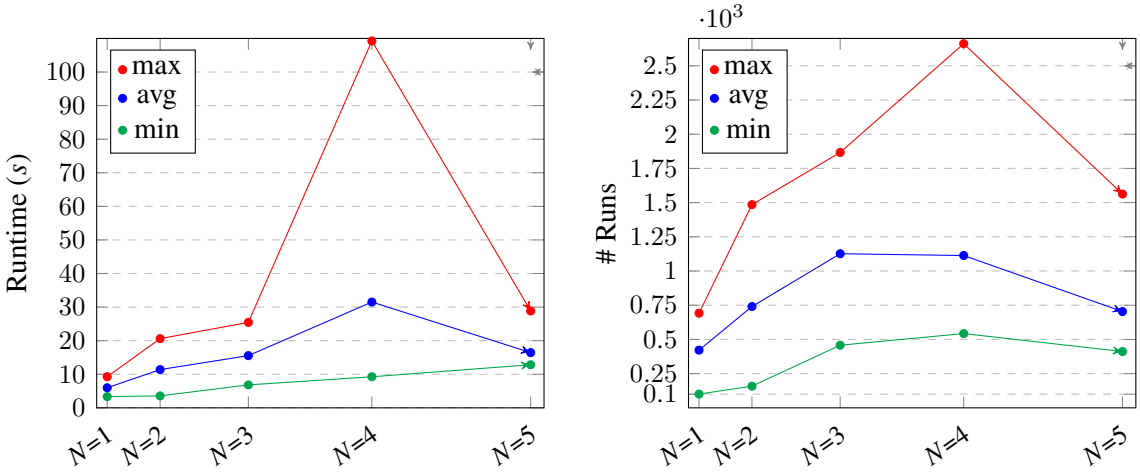


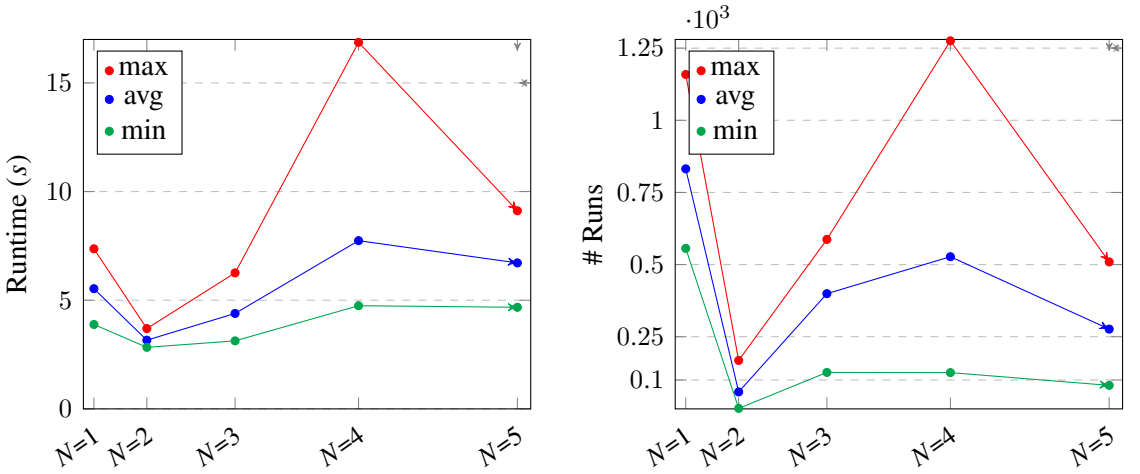
Figure 19.: (a) Total number of states in the MDPs of the scaled maps.
For good quality policies: (b) runtime of DSMC per map cell; (c) number of runs in DSMC per map cell. Each shown as a function of map size. (b) and (c) show min/average/max over 5 policies.

above in terms of the scaling behavior over N. Note that we used different scales to make the trend of the curves still visible by using intuitive value representations and scales. One should not be mislead by the outliers of the curve showing the maximum, and also not by the results for $N = 5$ which deviate minimally because of the difficulty of the largest instance. In terms of scaling over training degree, as discussed in the previous section, low-quality policies are much easier to analyze, as expected. For middle-quality policies, the results are less conclusive, with DSMC effort roughly similar to high-quality policies but with more variance. We conclude from this that the hard region as displayed in Figure 17 tends to be narrow, and correlates only loosely with training return.



(a) (b)

Figure 20.: Middle (70% return) policies: (a) runtime of DSMC per map cell; (b) number of runs in DSMC per map cell. Each shown as a function of map size, min/average/max over 5 policies.

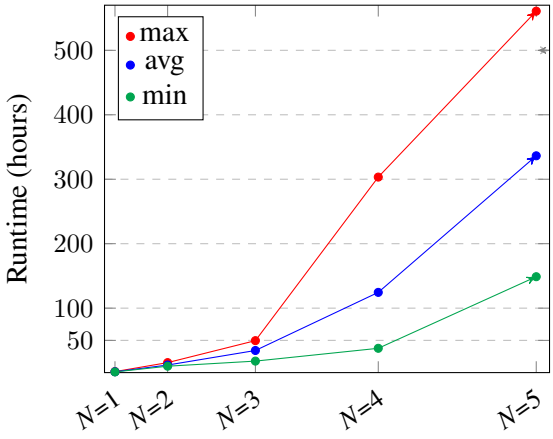


(a) (b)

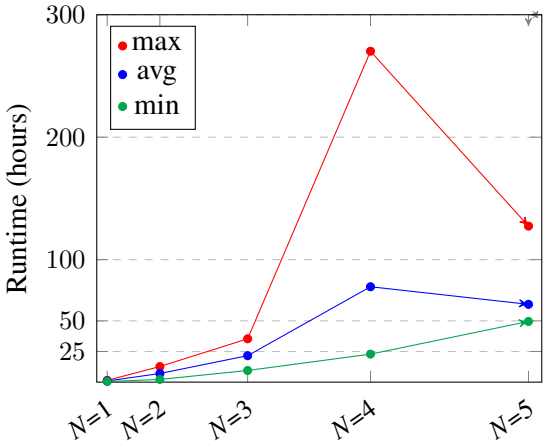
Figure 21.: Bad (20% return) policies: (a) runtime of DSMC per map cell; (b) number of runs in DSMC per map cell. Each shown as a function of map size, min/average/max over 5 policies.

Together, these findings indicate that DSMC can be scalable in non-trivial application scenarios. The data confirms the expected result that, all other circumstances being equal, run length is the determining factor for DSMC performance, and thus the advantages of statistical model checking carry over to DSMC.

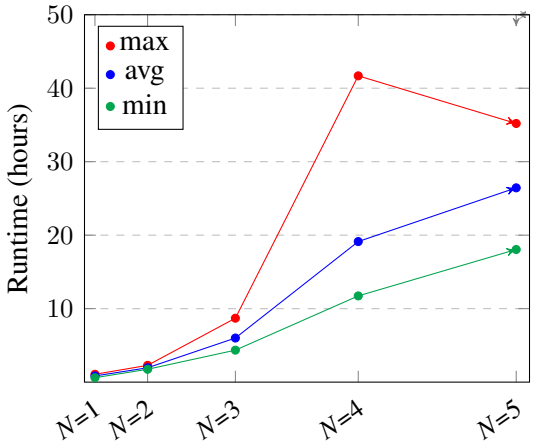
Next, we have a look at a straightforward measure for scalability over instance sizes, which is the overall runtime of DSMC executed on all cells of a map. In contrast to the previous



(a)



(b)



(c)

Figure 22.: Accumulated runtime of DSMC over whole map as function of map size for (a) good, (b) middle, and (c) bad quality policies; max, min, avg over 5 policies.

evaluations, this does not factor out complexity source (i), i.e., it is clear that the runtime has to increase for larger map instances just because more DSMC calls are needed and therefore this measure does not give the fine grained analysis insights obtained above.

Figure 22 (a) shows that the accumulated effort for DSMC across all map cells grows substantially as a function of N for the good policies, simply due to map size. This illustrates that an exhaustive analysis of the state space is highly demanding in these benchmarks. Note though, that this task is trivial to parallelize, so that it can still be feasible to check large fractions of the state space. Indeed, this was exploited in our experimental setup, running on a cluster of multicores. As already seen in the previous evaluations, for this setting again, low-quality policies are much easier to analyze (see Figure 22 (c)), as expected, and the results for middle-quality policies are less conclusive because of higher variance (see Figure 22 (b)).

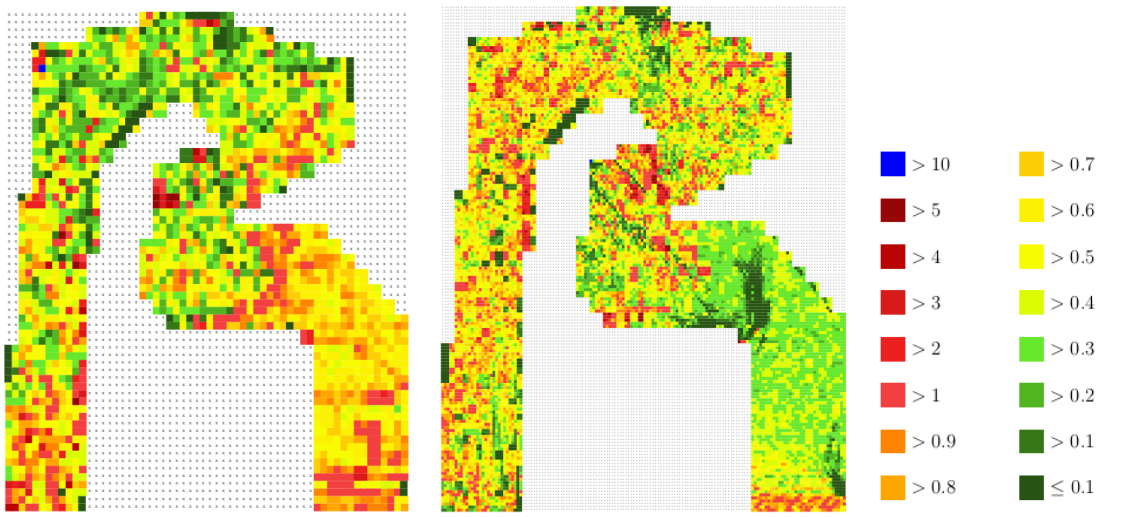


Figure 23.: DSMC runtime difference quotient $\frac{small-map}{large-map}$ per cell for $N = 1$ vs. $N = 2$ (left) and $N = 2$ vs. $N = 5$ (right).

With the help of heat maps it is also possible to compare even more detailed data w.r.t. scalability. Figure 23 provides a fine-grained view of differences in DSMC performance as a function of scaling size, comparing different NN action policies, first, one for $N = 1$ vs. one for $N = 2$ (left), and second, one for $N = 2$ vs. one for $N = 5$ (right). Each cell in the heat maps shows the quotient of DSMC runtime of the policy for the smaller map over the policy for the larger map, like it was done in Figure 18 (c) for the comparison of a good and a bad quality policy for $N = 3$. Map cells are aligned and compared across different map sizes according to their positions in the respective discretization. This means, a single cell of the smaller map is compared with all cells of the larger map it is partitioned into when scaling up. Therefore, the maps in Figure 23 each have the size of the larger map in the comparison.

In both heat maps "strong" colors are rare, i.e., there is only little dark green and dark red/blue. The runtime differences hence are mostly not extreme, corroborating our observations from Figure 19. There is however a certain degree of variation, which turns out to be again mostly caused by policy quality differences.

To understand this, consider first the left-hand side heat map. Near the start line and the goal of the track, orange and yellow dominate – indicating similar runtimes – because DSMC analysis for both values of N tends to be quick. This is different in the remaining middle part of the track, where there is more policy-success variance, and hence more sample runs are needed, for both values of N. The smaller map size for $N = 1$ then results in significantly smaller runtimes.

In the right-hand side heat map, the picture is not as clear. Differences are again small close to the goal (light green this time as the size gap from $N = 2$ to $N = 3$ is larger), but elsewhere the picture is very mixed. The latter is due to local policy-quality variation, which is more

pronounced in the larger maps. All the areas with distinctly large performance differences, e.g., the dark green stripe in the last curve, are due to poor quality of one of the two policies.

Summarizing the findings from all parts of the scalability study, we find that the policy quality yields a characteristic easy-hard-easy pattern in the effort of the DSMC analysis caused by the nature of the NN action policies at different stages. The results of the study on many instances of scaled Racetrack benchmarks turn out to be quite favorable, attesting to the potential of DSMC. With respect to instance size, our finding is that the average DSMC runtime per initial state grows linearly, indicating that DSMC is indeed scalable and inherits the beneficial properties of statistical model checking.

5.4. DSMC Visualization in TraceVis

In the domain of cyber-physical systems, which can be modeled by state spaces predestined for easy and vivid visualization, it is natural to explore the use of visualization to support DSMC users, like human analysts and domain engineers, considered already in the DSMC evaluation scenarios above.

The author of this thesis assisted in the design and construction of an interactive visualization tool for DSMC results of the Racetrack case study, called TraceVis, by implementing additional features in the Racetrack model and in modes to extract the required DSMC metadata used for visualization. In addition, the author contributed by giving advice on which data could be visualized or which features would be helpful for a domain engineer and for quality analysis. The tool implementation and design was done by David Groß and Stefan Gumhold. With this short section, we want to give an outlook on how valuable DSMC can be to gain deep insights in the behavior of learned agents to strengthen the contribution of DSMC even further.

The TraceVis tool enables exploration of crash probabilities for particular wall segments and goal reachability probabilities for individual goal states as a function of start position and velocity. This means, the more general goal and crash properties from above were refined to *crash-probability-into-x-y* and *goal-probability-into-x-y* where x and y specify the exact coordinates where the crash happens or where the goal is reached, respectively. Like before, the individual start position for which the property is evaluated is fixed by the initial state of the model given to DSMC.

The tool furthermore supports the in-depth examination of policy traces generated by DSMC, in aggregated form as well as individually. This demonstrates how visualization can foster the effective analysis of DSMC results, and it forms a first step in combining model checking and visualization in the analysis of NN behavior.

At this point, we want to highlight that with the usage of the term *trace* in this chapter the sample executions performed in SMC (see Section 2.2.2) are referred to, instead of the traces induced by paths as specified in Definition 15.

Since Racetrack acts in a 2-dimensional space, in a 3-dimensional visualization space TraceVis can make use of the third dimension to map additional features. TraceVis is implemented as a plugin to the CGV-Framework [114], which allows rapid prototyping of interactive 3D tools.

The concrete visualization techniques used in the tool and its implementation is out of scope of the thesis. Details can be found in the two respective papers [103, 113]. We concentrate on the information and data visualizable in the tool as well as the data collection and processing. We start with outlining the concept of the tool in terms of the data space it is able to visualize.

Data Collection. We collected extensive information about the to-be-analyzed action policies from modes, allowing to analyze policy behavior as a function of start position p *and* start velocity v, in combination with showing not only whether the policy succeeded or crashed but also *where*. To this end, we ran separate DSMC runs with modes for every pair (p, v), with properties encoding every possible terminal (goal/crash) position. The number of calls to DSMC is thus given by the map size and the number of boundary wall cells and goals, with a constant factor of 25 for the start velocities in $\{-2,-1,0,1,2\} \times \{-2,-1,0,1,2\}$. We ignored start velocities that immediately lead to a crash in the first step.

We furthermore collected all policy traces generated by modes during DSMC, with detailed per-step information: position, velocity, action taken by policy, and a Boolean indicating whether the action succeeded or failed, i.e., whether noise occurred. In Barto-big, to keep computation times reasonable, we generated this data only for 7 of the 25 possible start velocities.

Computation and export of this data for Barto-small and Barto-big took 17 and 20 hours, respectively, on 25 virtual machines having an AMD EPYC Processor at approximately 2.5 GHz using Ubuntu 18.04 with 8 vCPUs and 16 GB RAM. The data comprises 5 473 (3 826) start configurations consuming 1.25 MB (1.18 MB) for probabilities, and 15.3 GB (13.4 GB) for traces of Barto-small (reduced traces of Barto-big) on disk, in a concise text file format organized in two folders for probabilities and traces with one file per start configuration. The largest trace file has 13 MB on disk and contains 18 270 traces of average length 44 and maximal length 65. The data and infrastructure used for demonstrating the tool is publicly available, together with a video in which the tool usage is demonstrated, in the Zenodo archive[5] [172] of the DSMC material.

[5] http://doi.org/10.5281/zenodo.6362696

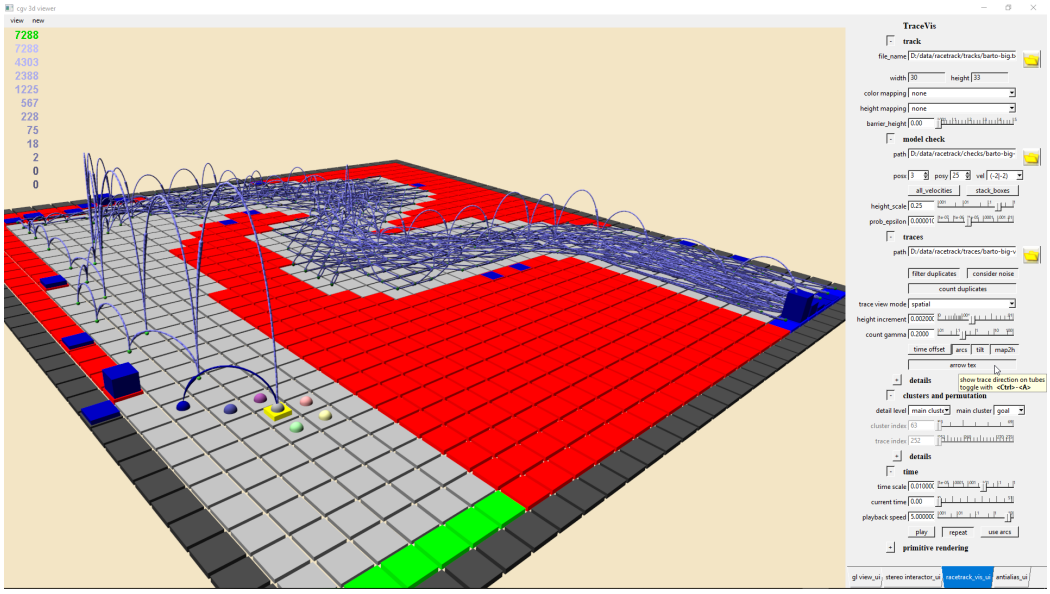


Figure 24.: Overview screenshot of TraceVis. Interactive 3D view of the track and traces, classical UI on the right showing current tool state and providing tooltip based help.

Tool Overview. Figure 24 illustrates the design of TraceVis. Each track position is depicted by a color coded box: start and goal locations in green and blue, respectively, walls in red, other track locations in light gray or color-mapped, and an additional row of dark gray boundary cells added around the track.

A visualization, similar to the one we used in the heat maps for manual evaluation of the DSMC results, serves as overview over all start configurations. The user can select individual start configurations to view crash and goal probabilities. In addition, it is possible to dive into more detail by switching to trace visualization mode where the corresponding trace file is read on the fly. The traces can further be navigated from main clusters down to single traces. All these visualization modes are demonstrated in the next sections.

5.4.1. Visualizing Probabilities

We next describe the techniques for visualizing crash and goal probabilities for individual wall and goal states as a function of start position p and start velocity v.

From the DSMC analysis with the refined crash and goal reachability probability properties, it is possible to calculate for each start configuration (p, v), and each wall, boundary, and goal location q the probability that traces from (p, v) end in q, by executing DSMC on a property considering exactly this start and goal, respectively, crash position. Visualizing the entire probability mapping $\mathcal{P}(p, v, q)$ in a single image or 3D-scene seems futile. Therefore, TraceVis supports the selection of a single p and/or a single v at a time. The velocity is visualized by a bent arrow with a direction dependent color scale also used for the respective

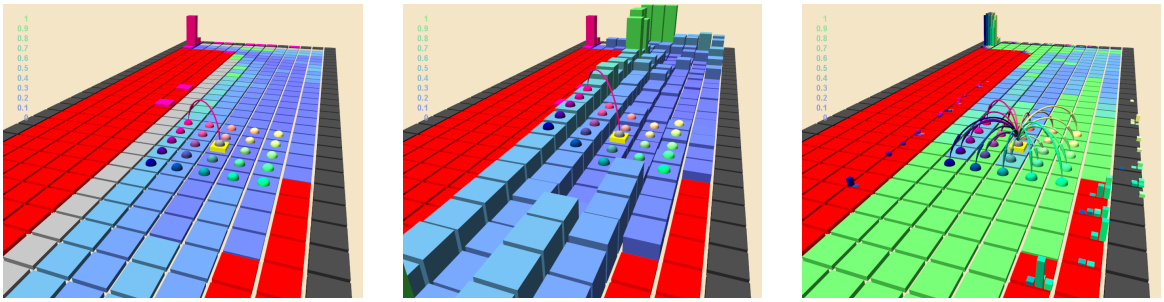


Figure 25.: Start configuration selection and different probability visualization approaches. Left: selected start configuration $(\hat{p}, \hat{v})$ shown as yellow box and pink arrow. Summed crash probabilities $\sum_{\tilde{q}} P(p, \hat{v}, \tilde{q})$ over all wall cells $\tilde{q}$ mapped to color of all valid track locations p. Pink bar charts show crash and goal probabilities for selected start configuration $(\hat{p}, \hat{v})$. Middle: Same as left, with additional mapping of summed crash probabilities to height of track boxes. Right: *All velocity* mode shows summed probabilities aggregated over all start velocities – here the aggregation function is the maximum. Colored charts show crash and goal probabilities for all start configurations $(\hat{p}, v)$, fixing $\hat{p}$ and ranging over all velocities.

crash and goal probabilities as shown in Figure 25 (left). It is also possible to select multiple or even all velocities for a starting position as illustrated in Figure 25 (right).

The crash and goal probabilities for each terminal position q for the selected start configurations is depicted by colored bar charts, visible in all parts of Figure 25. The bar heights indicate the probability of crashing or reaching the goal, respectively.

We extended the original heat maps (e.g., in Figure 11) by the option to adjust the height of the track boxes according to the crash probability, as shown in Figure 25 (middle). It is also possible to aggregate the probabilities per start location over all start velocities with one of the aggregation operators $\min$, $\max$ or range $= \max - \min$. These visualizations can be used as a guidance to find start configurations of interest and continuing further investigation from there.

5.4.2. Visualizing Policy Traces

Once a start configuration of interest is found, a natural means to investigate further is to inspect the actual policy traces generated by DSMC starting from there. TraceVis supports this in depth through the techniques we describe next. We modified the implementation of modes in such a way that the traces can be logged in a file with all information needed for the visualization. This file is then read by TraceVis. Details about the data collection are described in Section 5.4 above.

Figure 26 illustrates the three distinct modes to visualize traces: *stacked*, *spatial*, and *spacetime*. Traces are visualized as colored 3D tubes or, in the case of an aggregated view, as

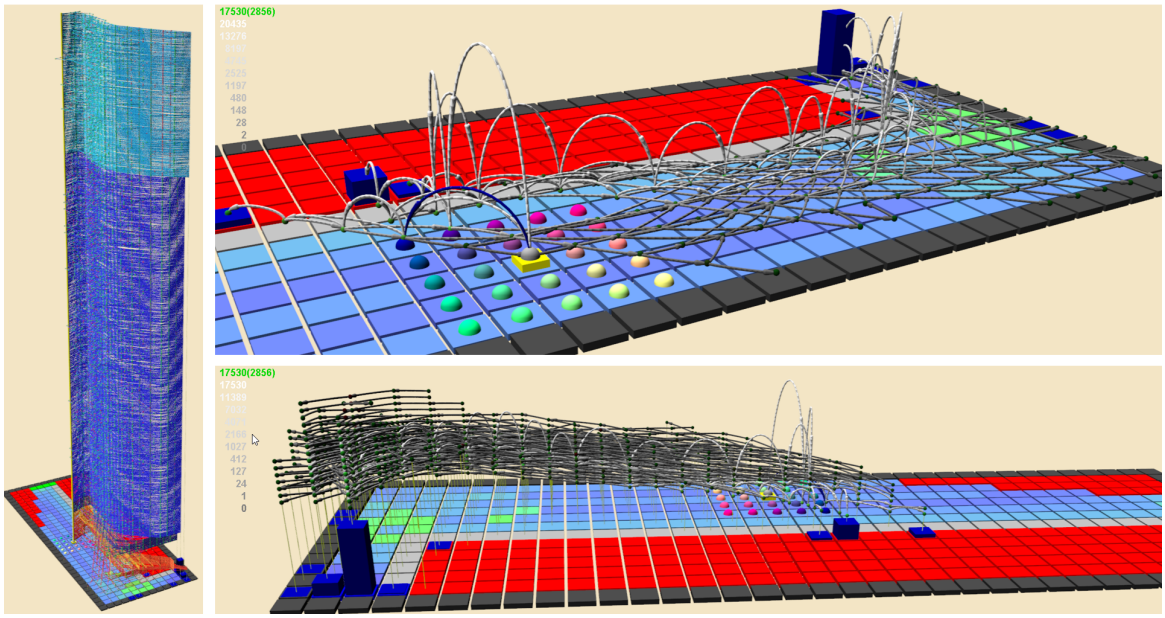


Figure 26.: Comparison of different trace rendering modes for a start configuration with 17 530 traces of which 2 856 remain after duplicate filtering. Left: *stacked* rendering of 2 856 traces, sorted and color coded by end location. Top right: *spatial* aggregation showing segments as arcs with appearance counts mapped to height and luminance. Bottom right: *spacetime* mode disaggregates segments over time, mapping time to height.

bent arrows with an arrow texture on the tubes to indicate the direction.

In *stacked* mode, all traces of the DSMC evaluation calculated for a specific start configuration are shown stacked vertically above the track (see Figure 26, left). Traces are sorted by their end location, and are arranged into two main clusters: one for traces that end at a goal position, colored blue to cyan, and one for those that crash, colored red to orange.

Given the number of traces, simply visualizing the set of all traces is often not helpful. Hence, two aggregation modes are available aggregating over discrete time and space. In the *spatial* aggregation, depicted in Figure 26 in the top right, segment histograms map the number of occurrences of the segments, defined by a common start and end state, in the DSMC traces to the height and to the luminance. On the bottom right of the figure the *spacetime* mode is depicted which disaggregates segments over time, mapping time to height, which allows to observe if an agent temporarily stops or if a segment is visited multiple times. Time is mapped to an increasing height offset. This allows the user to efficiently identify faster and slower runs. It is possible to navigate through clusters of traces down to individual traces.

Red or green spheres are placed at the map cells where two segments are connected, to indicate if noise occurred in the trace at that point or not, respectively. This is especially useful for the in-depth examination of individual traces, visualizing the policy reacting to

action failures at difficult track locations. For aggregated views, the color is interpolated between red and green according to the noise frequency in the traces produced by MODES. In the limit, this would result in the noise probability of the Racetrack configuration used.

5.4.3. Using TRACEVIS

To illustrate the use of TRACEVIS for policy behavior analysis, we shortly highlight some interesting observations supported by TRACEVIS when analyzing the NN policies in Racetrack.

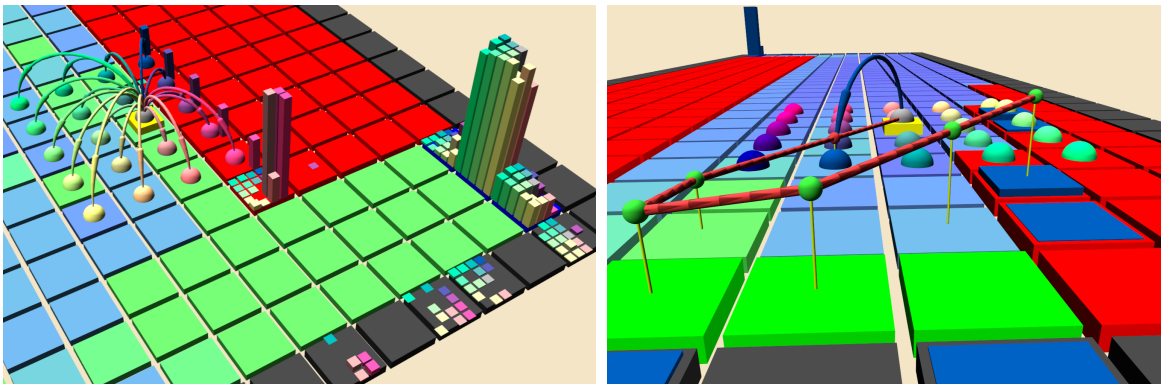


Figure 27.: Left: Unsafe behavior near goal line. Overview of crash and goal probabilities across start velocities. Right: Unsafe turning of an individual trace between walls.

Figure 27 on the left shows the crash and goal probabilities for a position just before the goal curve in Barto-small. We can see that overall the policy has a high chance of reaching the goal line. However, there are two start velocities not directed right away into the wall for which that is not the case, because the policy tends to "cut the corner" and crash.

On the right, a trace is depicted with the start position in a tight spot between walls on the left and right, with a start velocity away from the goal and to the left. The safest decision would be to "turn around on the spot", i.e., decelerate, get left-right velocity down to 0 and then accelerate to the goal. Instead, the policy over-accelerates to the right, going for a curve that would only just avoid the right-hand side wall if no noise occurs, i.e., it relies on action success, and is brittle to action failure (red balls), as we see in the crashing trace.

Note that TRACEVIS is key to all these observations. We miss them if we aggregate over start velocities (or fix these to 0 as in the previous sections), if we aggregate over crash positions, or if we have no in-depth visualization of policy traces.

During the construction of our evaluation setup we ourselves benefited from the visualization. Interestingly, TRACEVIS enabled us to find bugs in our own technology stack. Apart from initial data discrepancies due to bugs in cross-tool communication, this pertained also to a bug in our JANI model introduced when modifying it for the data extraction needed for the TRACEVIS tool. Examining crash probabilities as a function of start velocity as illustrated

above, we observed unintuitive results where start velocities heading directly into a wall did not lead to a crash. This behavior prompted us to re-examine the JANI model, identifying a bug in the vehicle automaton (where an initialization value was set incorrectly).

Similarly, we observed discrepancies between the crash probabilities computed by MODES vs. the traces visualized in TRACEVIS, leading us to identify a bug in the process of giving MODES' traces to TRACEVIS.

Such a faulty behavior would not have been visible in the simple heat maps as these ignore start velocities. Such bugs would be exceedingly hard to identify based solely on MODES, given the overwhelming amount of log data. Hence, the visualization in TRACEVIS can be useful also for debugging the model itself and for verifying the model construction, arguably a crucial part of model checking.

5.5. Related Work

As mentioned at the beginning of Chapter 5, the need to analyze and verify NNs is becoming more and more important. Thus, several quite different methods have been invented for automated NN analysis, e.g., special methods based on SAT-modulo-theories [85, 156, 170], abstract interpretation [99, 201], or quantitative analysis [71, 268] have been developed. But all these techniques have in common that they try to verify individual NN decision episodes, i.e., the behavior of a single input/output function call. The field of analyzing NNs taking the decisions in the context of a larger system with uncertainty, which we enter with our work here, is quite new and unexplored.

Only very recently, there have been endeavors into the direction of the verification of the overall system encompassing an NN. One of them is in the area of predicate abstraction on the state space induced by the NN action policy [264] on which safety properties can be verified using satisfiability checks. Another work [59] presents an approach called *Neuro-Aware Program Analysis* to verify safety properties of systems that symbiotically combine existing program and neural-network analyzers based on abstract interpretation.

There are a lot of other works combining formal methods with NNs, which we want to summarize in the following. For example, strategy synthesis for partially observable MDPs (POMDPs) to find strategies that fulfill certain probabilistic timed properties, has been conceived. In that approach a recurrent neural network (RNN) is trained to encode POMDP strategies. The RNN is then used to construct a Markov chain for which the temporal property can be checked using standard verification techniques [56]. The key difference to our work is that the Markov chain induced by a strategy given by the RNN is fully built and not simulated

to check if a given property holds. If it does not hold, a counterexample is generated for the purpose of locally improving the strategy.

Another work combining formal methods and machine learning studies the behavior of NN structures by extracting a decision-tree model of it over which reasoning is possible using model checking [5]. Based on this decision-tree model a synthesis procedure is proposed to create a stand-alone correct-by-design controller with performances similar to NNs trained with reinforcement learning methods. This controller can be integrated in a bounded model checking procedure to find retraining opportunities for the original NN.

To be able to add features to NNs acting as a controller without retraining and loosing too much performance, quantitative runtime shields have been proposed [14]. The shields may alter the command given by the controller before passing it to the system under control. To generate these shields, a stochastic model of the system is built. The goal is to construct a shield interfering as less as possible by guaranteeing the best performance. The controller's performance and the number of shield interferences is defined by quantitative measures on weighted automata. The shield construction task can then be reduced to finding an optimal strategy in a stochastic 2-player game on the stochastic model of the system by using a property combining requirement on the performance and number of interferences.

Furthermore, an iterative learning procedure consisting of SMT-solving and learning phases has been used to construct controllers for stochastic and partially unknown environments [165]. The problem is given as an MDP with an a-priori unknown cost function. Learning techniques can be used to get cost-optimal strategies but without safety guarantees. By first constructing a set of safe schedulers using an SMT-solver and then refining this set to an optimal scheduler, the problem can be solved.

In addition, a reinforcement learning algorithm has been proposed to synthesize policies which fulfill a given linear time property on an MDP [135]. By expressing the property as a Limit Deterministic Büchi Automaton, a reward function over the state-action pairs of the MDP can be defined such that the policy is only constructed by considering the part of the MDP which satisfies the property.

Another work on controller synthesis and verification uses policy refinement to construct strategies fulfilling temporal logic syntactically co-safe properties, which can be unbounded in time, on general MDPs (gMDPs) (discrete-time stochastic models over uncountable state spaces) by using approximately similar abstract models [118].

Statistical model checking for Bounded Linear Temporal Logic (BLTL) properties on MDPs has been done by first resolving the nondeterminism probabilistically and afterwards improving these resolutions of nondeterminism w.r.t. the satisfaction of the property with the help of reinforcement learning by building candidates for schedulers maximizing, respectively minimizing, the probability of the property [139].

Reachability properties have been verified on neural agent-environment systems represented as feed-forward ReLU NNs by expressing the problem as a mixed-integer linear program [3]. This approach has been applied to arbitrary-step reachability properties and properties asking if an action will be applied. An extension of this work [4] also supports agents defined on recurrent NNs [137] using a simplified version of linear temporal logic on bounded executions.

Our DSMC approach and the infrastructure we implemented in MODES has already been used to foster robust and safe behavior in deep reinforcement learning (DRL). It has been demonstrated that DSMC can be applied during DRL to determine state space regions with weak performance to concentrate on them during the learning process [105, 110]. The author of this thesis was involved in the development of this feedback mechanism used during the learning process. Especially in safety-critical applications DRL has two main deficiencies: (i) the training objective maximizes average rewards, which may disregard rare but critical situations and hence lack local robustness; (ii) optimization objectives targeting safety typically yield degenerated reward structures which for DRL to work must be replaced with proxy objectives. That is why we incorporated evaluation stages (ES) into DRL, leveraging DSMC. Our ES apply DSMC at regular intervals to determine state space regions with weak performance. We adapted the subsequent DRL training priorities based on the outcome, focusing DRL on critical situations, and allowing to foster arbitrary objectives. Our results show that DSMC-based ES can significantly improve both (i) and (ii).

With the help of MoGym, this technique can now be done much more integrated, and there is room for further implementations in this direction in our toolchain.

Regarding the TraceVis tool, in the context of explainable AI research a lot of recent works have been devoted to interactive visualization of NNs [149]. Goals for such techniques include interpretability, explainability, NN debugging as well as model comparison and selection. Most of the work has been dedicated to NNs for image analysis tasks. Only few recent works address the debugging and interpretation of deep networks used in reinforcement learning [267, 274]. These are dedicated to deep Q-Learning of agents playing Atari Retro Games, where high state-space dimensionality is the core problem addressed. To solve such issues, an analysis of the MDP through spacetime clustering of the state space has been made available [274]. The resulting hierarchical decomposition into skills allows for a better interpretation of the strategy of the learned agents. Wang et al. [267] developed a visual analysis tool with multiple coordinated views supporting a hierarchical navigation from an overview of the learning process down to individual traces of moves.

5.6. Discussion

NNs are an increasingly widespread decision-making component in intelligent systems. Verifying the overall behavior of systems incorporating such components is a grand challenge. When such a network is integrated into a control loop, the verification needs to intertwine controller and network verification [59].

This chapter has described the cornerstones of an effective methodology, called Deep Statistical Model Checking, which is a promising approach to address this challenge, leveraging the strength of statistical model checking as a lightweight approach for the purpose of checking the behavior of systems incorporating NNs, or trained decision-making agents in general, treated as blackbox functions that merely need to be called and not analyzed.

From a general perspective, DSMC provides a refined form of SMC for MDPs where thus far only implicitly defined random action policies have been available. If those were applied to Racetrack, goal probabilities smaller than 1% would result – except for cells directly at the goal line. DSMC instead can harvest available data for a far better suited action policy, in the form of an NN oracle trained on the data at hand. Of course, other forms of oracles, based on, e.g., random forests, can be considered with DSMC right away, too.

We have built up a tool infrastructure around DSMC for different contexts. This includes (i) benchmarking with Racetrack variants, (ii) DSMC evaluation of NNs and arbitrary oracles in MODES, and (iii) MOGYM usable for learning formal models and directly assessing their quality with DSMC, as well as (iv) the TRACEVIS tool for getting even deeper insights into the results of DSMC and also into the learning process. We hope that with this tool infrastructure we made DSMC accessible for a variety of users and purposes.

The most important aspects of the DSMC approach are (i) its genericity – in that it provides a generic and scalable basis for analyzing learnt action policies; (ii) its openness – since the approach is put into practice using the JANI format, supported by many tools for probabilistic or statistical model checking. Another important contribution basically decoupled from the DSMC approach is (iii) the focus on the Racetrack benchmark all evaluations were performed on, which focuses on a formal and extensible, but abstract fragment of the autonomous driving challenge. We consider these contributions as a conceptual nucleus of broader activities to foster the scientific understanding of neural network efficacy, by providing the formal and technological framework for precise, yet scalable problem analysis.

We have contributed case studies suggesting that the DSMC approach is indeed useful and feasible. In addition to these case studies, the benchmarking and scalability study gave evidence that DSMC is indeed scalable. The advantages of statistical model checking are inherited in our study, exhibiting a linear runtime increase per state as a function of instance size. We have furthermore shown that there are significant interactions between policy quality

and analysis performance, which become important when using DSMC during the training process, e.g., to identify weak-quality regions for re-training [105, 110].

Note also, that DSMC is highly parallelizable in terms of all its major activities, (i) statistical model checking (independent sample runs), (ii) NN evaluation (GPU/TPU hardware), and (iii) sweeping a state space partition. So, by leveraging large amounts of hardware, there is hope that very large scalability challenges can be tackled. We hope that the studies provide a compelling basis for further research on Deep Statistical Model Checking.

Our Racetrack case study makes it easy to produce heat maps, as a way to represent a partitioned perspective on the state space and sampling states as representatives. We believe that such a representative analysis makes sense in many scenarios, e.g., to provide an overview for human users. An open question is how to partition states, or how to support users in doing so. Physical location might work in many cases, especially for cyber-physical systems.

Inspired by the simple but helpful illustration of the DSMC results with heat maps TraceVis has been developed for visualizing and navigating DSMC results as well as for deeply understanding the underlying causes by examining the actual policy traces.

Even though we focused on the Racetrack benchmark, we showed that our contribution is not limited to it since we opened the DSMC approach to other formal models with MoGym, and argued that many cyber-physical systems fit into the DSMC framework, i.e., many ideas and concepts will carry over to other and more complex domains. That said, we believe that the Racetrack case study was useful, and remains useful, to focus on key aspects of many cyber-physical systems. Apart from the extension of our study to more general Racetrack maps and to examples with larger state spaces, an important scaling dimension yet to be evaluated is NN complexity. In particular, convolutional networks from computer vision are of interest, in a context where the inputs given to a decision entity are images. Such an architecture is possible in principle, but would require an extension of DSMC to incorporate a model-to-NN adapter producing or approximating the image based on the MDP state. Inspirations for this could be taken from the field of semantic segmentation, image segmentation, and instance segmentation [89, 241, 255], which has already been applied in the autonomous driving context in the opposite direction to abstract the view of the car to the inputs relevant for the decision-making agent.

In addition, an extension of the DSMC approach to other model types such as PTA would also be beneficial, and as already mentioned earlier, a multi-agent setting for DSMC is also possible as a straightforward extension of the approach.

Recently, SMC has been extended to parametric Markov chains [25] to check the flight plan of unmanned aerial vehicles. This application and case study is quite close to Racetrack and the autonomous driving challenge. It would be worth exploring the combination of both strategies in a common use case featuring NN decision-making and parametric models.

6. Conclusion

We formulated two main goals in the introduction of this thesis. In the first goal, we stated that we intend to contribute research enabling us to check properties of complex systems, e.g., to measure safety risks of such systems. The second goal was to conceive approaches to make systems more perspicuous, i.e., we wanted to develop tools helping in explaining and in giving reasons why certain decisions were taken in such complex systems or what the causes of errors were.

With the contributions presented in the thesis, we met our goals by introducing new methodologies to analyze complex systems' behaviors, to assess their quality, and to make them more perspicuous. These methodologies have in common that they avoid the exploration of the whole state space of the system under investigation by only considering a small part of the state space of the model relevant to answer the properties of interest. Of course, there is plenty of room for improvements and continuations of our work. Some of these ideas for further and also new developments are discussed in the following after summarizing our achievements.

Prior to describing the core contributions of the thesis, we first layed the foundations for benchmarking and testing the algorithmic innovations by introducing the Quantitative Verification Benchmark Set and the QComp competition to which we contributed significantly. In addition, we discussed the Racetrack benchmark and its variants, which model the autonomous driving challenge as a formal testbed, and are quite flexible w.r.t. the considered abstraction level.

With Modysh and Deep Statistical Model Checking, we contributed two new and, w.r.t. to the application purpose, quite different, promising approaches to make the cyber-physical world of today and of the future more perspicuous and safe. But Modysh and DSMC also have a strong commonality w.r.t. to the procedure they use. Both approaches try to tackle issues rooted in the state space explosion problem by considering only a small part of the state space of the model sufficient to answer the properties of interest.

Modysh is a more classical verification approach at the interface between probabilistic model checking and planning. It uses the knowledge about dynamic heuristic search gained

in the automated planning community to address the state space explosion problem of exhaustive model checkers. We adapted and modified the established LRTDP and FRET algorithms such that they are applicable to MDPs with positive and zero-valued rewards, on all properties considered interesting in the model checking community (except long-run averages and nested properties). These properties are extremal reachability probabilities, expected reward properties, and bounded versions thereof. To the best of our knowledge, we are the first to enhance FRET-LRTDP to work for these problems, and to give correctness and optimality proofs for the adapted algorithms on these problems. Modysh does not only enlarge the supported set of properties in the Modest Toolset but it especially also enlarges the set of property types and problem classes solvable with the combination of FRET and LRTDP. By only visiting parts of the state space relevant to solve the property of interest, Modysh is able to deliver results for benchmarks not solvable by any other state-of-the-art model checking tool so far. We demonstrated its strengths in a comprehensive benchmarking study in a setting similar to the QComp competition. With Modysh, planning approaches are now readily available for model checking benchmarks without the need for a priori translations to planning languages or formalisms. The development of Modysh contributes significantly to our first goal.

For the quality assessment of trained decision-making agents, like neural networks, we introduced a new technique based on statistical model checking which we call Deep Statistical Model Checking. The idea behind DSMC is to use the decision-making agent as an oracle to resolve nondeterminism in the formal MDP model of the environment the agent is trained to act in to examine its quality with statistical model checking. We have built up tool infrastructure for DSMC in different areas and contexts, comprising model checking, learning, and visualization. DSMC is one of the first approaches for verification of neural network decision-making agents used in a larger system environment as a whole instead of only analyzing single decision episodes. With DSMC, we tackle the entire complexity of analyzing the NN in combination with the analysis of the potentially quite large system it acts in. In other words, DSMC can handle the complexity of NN analysis in face of the state space explosion problem. In a comprehensive scalability study, we have demonstrated that it is indeed a lightweight approach to which the properties of SMC carry over. We have illustrated its broad applicability for learning pipeline assessment and quality assurance of trained decision-making agents by delivering insights which cannot be obtained from standard learning performance measures. With the integration of DSMC into MoGym, we have provided a single tool incorporating all functionality from learning on formal models to quality assessment. With the heat maps we used in our comprehensive case studies and especially with the visualizations in TraceVis, we have contributed to the solution of the

notorious problem of the complex internal NN function representation which is usually not suited to human inspection and highly complex to analyze automatically. We have shown that DSMC can make systems more perspicuous for people from different backgrounds, like domain engineers, learning experts, engineers in system approval or certification as well as end users, who want to assess the quality of the neural network's decisions to gain an understanding why the system behaves in a certain way, or why a certain decision is taken. This shows how DSMC mainly contributes to reaching our second goal and at the same time supports us in achieving the first goal.

To summarize, in this thesis we showed how to combine knowledge developed over years in the model checking community with approaches from the planning community and developments of the learning area to solve recent problems arising in a cyber-physical world. We layed the foundations for more cooperations to join forces to make it possible to trust and rely on increasingly complex systems.

More concretely, we believe that further approaches of the heuristic search and planning community can be ported to the model checking area, e.g., by extending Modysh with the implementation of other heuristics helpful for specific system model structures, or by exploring the trade-off between memory usage and runtime even further in such algorithms which explore only the parts relevant for the problem under investigation. In addition, it would be beneficial to develop a sound version of the algorithms in Modysh to obtain results with more strict guarantees on the result precision, inspired by sound value iteration, interval iteration, and optimistic value iteration [23, 48, 117, 130, 228].

Furthermore, extending both approaches, the implementation of Modysh as well as DSMC, to more automata types and problem definitions is worth exploring. Such an extension of DSMC could, e.g., also encompass parametric models harvesting SMC for parametric Markov chains [25] in combination with DSMC in use cases close to the automated driving challenge.

Speaking about enlarging the domain of inputs, it is definitely necessary to investigate the application of DSMC in settings with quite different types of decision-making agents, e.g., convolutional networks. Such an implementation is already prepared for in the interfaces of MoGym and modes, but most likely would require an intermediate layer or adapter similar to applying semantic segmentation or instance segmentation [89, 241, 255], as done in other systems attacking the autonomous driving challenge.

In addition, it would be beneficial to extend DSMC to be able to handle a multi-agent setting, where more than one automaton is controlled by decision-making agents.

Since neural networks and other trained decision-making agents are more and more integrated and deeply interweaved inside cyber-physical systems, the model checking community has to work hand in hand with the learning and AI experts to intertwine

the verification of the system and the decision-making agent even more. We did already make a first step into this direction with the functionality provided by MoGym, and of course with the evaluation stages made possible with the interleaving of DSMC and deep Q-learning [105, 110], but there is more potential to investigate.

In addition, because of the genericity, scalability, and openness of the approach, one could imagine to make the DSMC infrastructure accessible to even more user groups and purposes, always depending on where and with which interests users interact with decision-making agents. Such adaptions and extensions mainly depend on the intended application for which the users want to get more information on the system's functionality and quality. Porting of visualization approaches can play a major role here. We started to contribute research in this direction with the heat maps, and especially the TraceVis tool, which both build up on the scheme of partitioning the state space into representatives w.r.t. to the physical location. Such an analysis of representatives of states providing an overview could make sense in many cyber-physical systems.

Of course, the support which can be provided by visualization approaches should be explored for many more model checking problems to make systems more perspicuous and accessible by more user groups. Making formal analysis results accessible for human inspection seems to be a key instrument for the future.

For all these future works, Racetrack constitutes a challenging benchmark because it is a formal and precise testbed, and because of its flexibility and extensibility into the direction of a more and more realistic environment resembling the autonomous driving challenge.

Bibliography

[1] Forest Agostinelli, Stephen McAleer, Alexander Shmakov, and Pierre Baldi. Solving the Rubik's Cube with Deep Reinforcement Learning and Search. *Nature Machine Intelligence*, 1(8):356–363, 2019.

[2] Sheldon B. Akers, Jr. On a Theory of Boolean Functions. *Journal of the Society for Industrial and Applied Mathematics*, 7(4):487–498, 1959.

[3] Michael Akintunde, Alessio Lomuscio, Lalit Maganti, and Edoardo Pirovano. Reachability Analysis for Neural Agent-Environment Systems. In Michael Thielscher, Francesca Toni, and Frank Wolter, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Sixteenth International Conference, KR 2018, Tempe, Arizona, 30 October - 2 November 2018*, pages 184–193. AAAI Press, 2018.

[4] Michael E. Akintunde, Andreea Kevorchian, Alessio Lomuscio, and Edoardo Pirovano. Verification of RNN-Based Neural Agent-Environment Systems. 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 6006–6013. AAAI Press, 2019.

[5] Parand Alizadeh Alamdari, Guy Avni, Thomas A. Henzinger, and Anna Lukina. Formal Methods with a Touch of Magic. In *2020 Formal Methods in Computer Aided Design, FMCAD 2020, Haifa, Israel, September 21-24, 2020*, pages 138–147. IEEE, 2020.

[6] Aws Albarghouthi, Jorge A Baier, and Sheila A McIlraith. On the Use of Planning Technology for Verification. In *VVPS'09. Proceedings of the ICAPS Workshop on Verification & Validation of Planning & Scheduling Systems*, 2009.

[7] Husain Aljazzar and Stefan Leue. Generation of Counterexamples for Model Checking of Markov Decision Processes. In *QEST 2009, Sixth International Conference on the Quantitative Evaluation of Systems, Budapest, Hungary, 13-16 September 2009*, pages 197–206. IEEE Computer Society, 2009.

[8] Rajeev Alur. *Principles of Cyber-Physical Systems*. The MIT Press, 2015.

[9] Todd R. Andel and Alec Yasinsac. On the Credibility of Manet Simulations. *Computer*, 39(7):48–54, 2006.

[10] Robert B Ash, B Robert, Catherine A Doleans-Dade, and A Catherine. *Probability and Measure Theory*. Academic Press, 2000.

[11] Pranav Ashok, Tomás Brázdil, Jan Křetínský, and Ondrej Slámecka. Monte Carlo Tree Search for Verifying Reachability in Markov Decision Processes. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Verification - 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part II*, volume 11245 of *Lecture Notes in Computer Science*, pages 322–335. Springer, 2018.

[12] Pranav Ashok, Yuliya Butkova, Holger Hermanns, and Jan Křetínský. Continuous-Time Markov Decisions Based on Partial Exploration. In Shuvendu K. Lahiri and Chao Wang, editors, *Automated Technology for Verification and Analysis - 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings*, volume 11138 of *Lecture Notes in Computer Science*, pages 317–334. Springer, 2018.

[13] Pranav Ashok, Jan Křetínský, and Maximilian Weininger. PAC Statistical Model Checking for Markov Decision Processes and Stochastic Games. 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 497–519. Springer, 2019.

[14] Guy Avni, Roderick Bloem, Krishnendu Chatterjee, Thomas A. Henzinger, Bettina Könighofer, and Stefan Pranger. Run-Time Optimization for Learned Controllers Through Quantitative Games. 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 630–649. Springer, 2019.

[15] Christel Baier. Probabilistic Model Checking. 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 1–23. IOS Press, 2016.

[16] Christel Baier, Maria Christakis, Timo P. Gros, David Groß, Stefan Gumhold, Holger Hermanns, Jörg Hoffmann, and Michaela Klauck. Lab Conditions for Research on Explainable Automated Decisions. In Fredrik Heintz, Michela Milano, and Barry O'Sullivan, editors, *Trustworthy AI - Integrating Learning, Optimization and Reasoning - First International Workshop, TAILOR 2020, Virtual Event, September 4-5, 2020, Revised Selected Papers*, volume 12641 of *Lecture Notes in Computer Science*, pages 83–90. Springer, 2020.

[17] Christel Baier, Edmund M. Clarke, Vasiliki Hartonas-Garmhausen, Marta Z. Kwiatkowska, and Mark Ryan. Symbolic Model Checking for Probabilistic Processes. In Pierpaolo Degano, Roberto Gorrieri, and Alberto Marchetti-Spaccamela, editors, *Automata, Languages and Programming, 24th International Colloquium, ICALP'97, Bologna, Italy, 7-11 July 1997, Proceedings*, volume 1256 of *Lecture Notes in Computer Science*, pages 430–440. Springer, 1997.

[18] Christel Baier, Pedro R. D'Argenio, and Marcus Größer. Partial Order Reduction for Probabilistic Branching Time. *Electronic Notes in Theoretical Compututer Science*, 153(2):97–116, 2006.

[19] Christel Baier, Luca de Alfaro, Vojtech Forejt, and Marta Kwiatkowska. Model Checking Probabilistic Systems. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 963–999. Springer, 2018.

[20] Christel Baier, Clemens Dubslaff, Holger Hermanns, Michaela Klauck, Sascha Klüppelholz, and Maximilian A. Köhl. Components in Probabilistic Systems: Suitable by Construction. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part I*, volume 12476 of *Lecture Notes in Computer Science*, pages 240–261. Springer, 2020.

[21] Christel Baier, Holger Hermanns, and Joost-Pieter Katoen. The 10, 000 Facets of MDP Model Checking. In Bernhard Steffen and Gerhard J. Woeginger, editors, *Computing and Software Science - State of the Art and Perspectives*, volume 10000 of *Lecture Notes in Computer Science*, pages 420–451. Springer, 2019.

[22] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, 2008.

[23] Christel Baier, Joachim Klein, Linda Leuschner, David Parker, and Sascha Wunderlich. Ensuring the Reliability of Your Model Checker: Interval Iteration for Markov Decision Processes. In Rupak Majumdar and Viktor Kuncak, editors, *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I*, volume 10426 of *Lecture Notes in Computer Science*, pages 160–180. Springer, 2017.

[24] Jorge A. Baier and Sheila A. McIlraith. Planning with First-Order Temporally Extended Goals using Heuristic Search. In *Proceedings, The Twenty-First National Conference on Artificial Intelligence and the Eighteenth Innovative Applications of Artificial Intelligence Conference, July 16-20, 2006, Boston, Massachusetts, USA*, pages 788–795. AAAI Press, 2006.

[25] Ran Bao, J. Christian Attiogbé, Benoît Delahaye, Paulin Fournier, and Didier Lime. Parametric Statistical Model Checking of UAV Flight Plan. In Jorge A. Pérez and Nobuko Yoshida, editors, *Formal Techniques for Distributed Objects, Components, and Systems - 39th IFIP WG 6.1 International Conference, FORTE 2019, Held as Part of the 14th International Federated Conference on Distributed Computing Techniques, DisCoTec 2019, Kongens Lyngby, Denmark, June 17-21, 2019, Proceedings*, volume 11535 of *Lecture Notes in Computer Science*, pages 57–74. Springer, 2019.

[26] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). https://smtlib.cs.uiowa.edu/, 2016.

[27] Andrew G. Barto, Steven J. Bradtke, and Satinder P. Singh. Learning to Act Using Real-Time Dynamic Programming. *Artificial Intelligence*, 72(1-2):81–138, 1995.

[28] Ananda Basu, Saddek Bensalem, Marius Bozga, Benoît Delahaye, and Axel Legay. Statistical Abstraction and Model-Checking of Large Heterogeneous Systems. *International Journal on Software Tools for Technology Transfer*, 14(1):53–72, 2012.

[29] Peter Baumgartner, Sylvie Thiébaux, and Felipe W. Trevizan. Heuristic Search Planning With Multi-Objective Probabilistic LTL Constraints. In Michael Thielscher, Francesca Toni, and Frank Wolter, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Sixteenth International Conference, KR 2018, Tempe, Arizona, 30 October - 2 November 2018*, pages 415–424. AAAI Press, 2018.

[30] Marc G. Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The Arcade Learning Environment: An Evaluation Platform for General Agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.

[31] R. Bellman, R.E. Bellman, and Rand Corporation. *Dynamic Programming*. Rand Corporation research study. Princeton University Press, 1957.

[32] Richard Bellman. A Markovian Decision Process. *Journal of Mathematics and Mechanics*, 6(5):679–684, 1957.

[33] Jon Louis Bentley and Andrew Chi-Chih Yao. An Almost Optimal Algorithm for Unbounded Searching. *Information Processing Letters*, 5(3):82–87, 1976.

[34] Dimitri Bertsekas, P Tsitsiklis, and N John. *Parallel and Distributed Computation: Numeral Methods*. Prentice-Hall Inc., 1989.

[35] Dimitri P Bertsekas. *Dynamic Programming and Optimal Control, Vol. 1*. Athena Scientific, 1995.

[36] Dimitri P Bertsekas. *Dynamic Programming and Optimal Control, Vol. 2*. Athena Scientific, 1995.

[37] Dirk Beyer. Software Verification with Validation of Results - (Report on SV-COMP 2017). In Axel Legay and Tiziana Margaria, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part II*, volume 10206 of *Lecture Notes in Computer Science*, pages 331–349, 2017.

[38] Dirk Beyer and Thomas Lemberger. Software Verification: Testing vs. Model Checking - A Comparative Evaluation of the State of the Art. In Ofer Strichman and Rachel Tzoref-Brill, editors, *Hardware and Software: Verification and Testing - 13th International Haifa Verification Conference, HVC 2017, Haifa, Israel, November 13-15, 2017, Proceedings*, volume 10629 of *Lecture Notes in Computer Science*, pages 99–114. Springer, 2017.

[39] Wolfgang Bibel. *Automated Theorem Proving, 2nd Edition*. Artificial Intelligence. Vieweg, 1987.

[40] Keshav Bimbraw. Autonomous Cars: Past, Present and Future - A Review of the Developments in the Last Century, the Present Scenario and the Expected Future of Autonomous Vehicle Technology. In Joaquim Filipe, Kurosh Madani, Oleg Yu. Gusikhin, and Jurek Z. Sasiadek, editors, *ICINCO 2015 - Proceedings of the 12th International Conference on Informatics in Control, Automation and Robotics, Volume 1, Colmar, Alsace, France, 21-23 July, 2015*, pages 191–198. SciTePress, 2015.

[41] John G. Blitch. Artificial Intelligence Technologies for Robot Assisted Urban Search and Rescue. *Expert Systems with Applications*, 11(2):109–124, 1996.

[42] Stefan Blom and Jaco van de Pol. State Space Reduction by Proving Confluence. 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 596–609. Springer, 2002.

[43] Jonathan Bogdoll, Luis María Ferrer Fioriti, Arnd Hartmanns, and Holger Hermanns. Partial Order Methods for Statistical Model Checking and Simulation. In Roberto Bruni and Jürgen Dingel, editors, *Formal Techniques for Distributed Systems - Joint 13th IFIP WG 6.1 International Conference, FMOODS 2011, and 31st IFIP WG 6.1 International Conference, FORTE 2011, Reykjavik, Iceland, June 6-9, 2011. Proceedings*, volume 6722 of *Lecture Notes in Computer Science*, pages 59–74. Springer, 2011.

[44] Jonathan Bogdoll, Arnd Hartmanns, and Holger Hermanns. Simulation and Statistical Model Checking for Modestly Nondeterministic Models. In Jens B. Schmitt, editor, *Measurement, Modelling, and Evaluation of Computing Systems and Dependability and Fault Tolerance - 16th International GI/ITG Conference, MMB & DFT 2012, Kaiserslautern, Germany, March 19-21, 2012. Proceedings*, volume 7201 of *Lecture Notes in Computer Science*, pages 249–252. Springer, 2012.

[45] Blai Bonet and Hector Geffner. Labeled RTDP: Improving the Convergence of Real-Time Dynamic Programming. In Enrico Giunchiglia, Nicola Muscettola, and Dana S. Nau, editors, *Proceedings of the Thirteenth International Conference on Automated Planning and Scheduling (ICAPS 2003), June 9-13, 2003, Trento, Italy*, pages 12–21. AAAI, 2003.

[46] Blai Bonet and Hector Geffner. Learning Depth-First Search: A Unified Approach to Heuristic Search in Deterministic and Non-Deterministic Settings, and Its Application to MDPs. In Derek Long, Stephen F. Smith, Daniel Borrajo, and Lee McCluskey, editors, *Proceedings of the Sixteenth International Conference on Automated Planning and Scheduling, ICAPS 2006, Cumbria, UK, June 6-10, 2006*, pages 142–151. AAAI, 2006.

[47] Luca Bortolussi, Dimitrios Milios, and Guido Sanguinetti. Smoothed Model Checking for Uncertain Continuous-Time Markov Chains. *Information and Computation*, 247:235–253, 2016.

[48] Tomás Brázdil, Krishnendu Chatterjee, Martin Chmelik, Vojtech Forejt, Jan Křetínský, Marta Z. Kwiatkowska, David Parker, and Mateusz Ujma. Verification of Markov Decision Processes Using Learning Algorithms. In Franck Cassez and Jean-François Raskin, editors, *Automated Technology for Verification and Analysis - 12th International Symposium, ATVA 2014, Sydney, NSW, Australia, November 3-7, 2014, Proceedings*, volume 8837 of *Lecture Notes in Computer Science*, pages 98–114. Springer, 2014.

[49] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI Gym. *CoRR*, abs/1606.01540, 2016.

[50] Carlos E. Budde, Pedro R. D'Argenio, and Arnd Hartmanns. Better Automated Importance Splitting for Transient Rare Events. In Kim Guldstrand Larsen, Oleg Sokolsky, and Ji Wang, editors, *Dependable Software Engineering. Theories, Tools, and Applications - Third International Symposium, SETTA 2017, Changsha, China, October 23-25, 2017, Proceedings*, volume 10606 of *Lecture Notes in Computer Science*, pages 42–58. Springer, 2017.

[51] Carlos E. Budde, Pedro R. D'Argenio, Arnd Hartmanns, and Sean Sedwards. An Efficient Statistical Model Checker for Nondeterminism and Rare Events. *International Journal on Software Tools for Technology Transfer*, 22(6):759–780, 2020.

[52] Carlos E. Budde, Pedro R. D'Argenio, and Raúl E. Monti. Compositional Construction of Importance Functions in Fully Automated Importance Splitting. In Antonio Puliafito, Kishor S. Trivedi, Bruno Tuffin, Marco Scarpa, Fumio Machida, and Javier Alonso, editors, *10th EAI International Conference on Performance Evaluation Methodologies and Tools, VALUETOOLS 2016, Taormina, Italy, 25th-28th Oct 2016*. ACM, 2016.

[53] Carlos E. Budde, Christian Dehnert, Ernst Moritz Hahn, Arnd Hartmanns, Sebastian Junges, and Andrea Turrini. JANI: Quantitative Model and Tool Interaction. In Axel Legay and Tiziana Margaria, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part II*, volume 10206 of *Lecture Notes in Computer Science*, pages 151–168, 2017.

[54] Carlos E. Budde, Arnd Hartmanns, Michaela Klauck, Jan Křetínský, David Parker, Tim Quatmann, Andrea Turrini, and Zhen Zhang. On Correctness, Precision, and Performance in Quantitative Verification - QComp 2020 Competition Report. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Tools and Trends - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part IV*, volume 12479 of *Lecture Notes in Computer Science*, pages 216–241. Springer, 2020.

[55] Sinan Çalisir and Meltem Kurt PehlIvanoõlu. Model-Free Reinforcement Learning Algorithms: A Survey. In *27th Signal Processing and Communications Applications Conference, SIU 2019, Sivas, Turkey, April 24-26, 2019*, pages 1–4. IEEE, 2019.

[56] Steven Carr, Nils Jansen, Ralf Wimmer, Alexandru Constantin Serban, Bernd Becker, and Ufuk Topcu. Counterexample-Guided Strategy Improvement for POMDPs Using Recurrent Neural Networks. In Sarit Kraus, editor, *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, pages 5532–5539. ijcai.org, 2019.

[57] David Cavin, Yoav Sasson, and André Schiper. On the Accuracy of MANET Simulators. In *Proceedings of the 2002 Workshop on Principles of Mobile Computing, POMC 2002, October 30-31, 2002, Toulouse, France*, pages 38–43. ACM, 2002.

[58] Ming-Wei Chang, Lev-Arie Ratinov, Nicholas Rizzolo, and Dan Roth. Learning and Inference with Constraints. In Dieter Fox and Carla P. Gomes, editors, *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008*, pages 1513–1518. AAAI Press, 2008.

[59] Maria Christakis, Hasan Ferit Eniser, Holger Hermanns, Jörg Hoffmann, Yugesh Kothari, Jianlin Li, Jorge A. Navas, and Valentin Wüstholz. Automated Safety Verification of Programs Invoking Neural Networks. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I*, volume 12759 of *Lecture Notes in Computer Science*, pages 201–224. Springer, 2021.

[60] Kai Lai Chung and Kailai Zhong. *A Course in Probability Theory*. Academic Press, 2001.

[61] Frank Ciesinski, Christel Baier, Marcus Größer, and Joachim Klein. Reduction Techniques for Model Checking Markov Decision Processes. In *Fifth International Conference on the Quantitative Evaluaiton of Systems (QEST 2008), 14-17 September 2008, Saint-Malo, France*, pages 45–54. IEEE Computer Society, 2008.

[62] Alessandro Cimatti, Stefan Edelkamp, Maria Fox, Daniele Magazzeni, and Erion Plaku. Automated Planning and Model Checking (Dagstuhl Seminar 14482). *Dagstuhl Reports*, 4(11):227–245, 2014.

[63] 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, Workshop, Yorktown Heights, New York, USA, May 1981*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 1981.

[64] Edmund M. Clarke, E. Allen Emerson, and Joseph Sifakis. Model Checking: Algorithmic Verification and Debugging. *Communications of the ACM*, 52(11):74–84, 2009.

[65] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-Guided Abstraction Refinement. In E. Allen Emerson and A. Prasad Sistla, editors, *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2000.

[66] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Progress on the State Explosion Problem in Model Checking. In Reinhard Wilhelm, editor, *Informatics - 10 Years Back. 10 Years Ahead*, volume 2000 of *Lecture Notes in Computer Science*, pages 176–194. Springer, 2001.

[67] Edmund M. Clarke, William Klieber, Milos Novácek, and Paolo Zuliani. Model Checking and the State Explosion Problem. In Bertrand Meyer and Martin Nordio, editors, *Tools for Practical Software Verification, LASER, International Summer School 2011, Elba Island, Italy, Revised Tutorial Lectures*, volume 7682 of *Lecture Notes in Computer Science*, pages 1–30. Springer, 2011.

[68] Edmund M. Clarke Jr., Orna Grumberg, Daniel Kroening, Doron Peled, and Helmut Veith. *Model Checking*. MIT Press, 2018.

[69] 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.

[70] Javier Criado, José Andrés Asensio, Nicolás Padilla, and Luis Iribarne. Integrating Cyber-Physical Systems in a Component-Based Approach for Smart Homes. *Sensors*, 18(7):2156, 2018.

[71] Francesco Croce, Maksym Andriushchenko, and Matthias Hein. Provable Robustness of ReLU Networks via Maximization of Linear Regions. In Kamalika Chaudhuri and Masashi Sugiyama, editors, *The 22nd International Conference on Artificial Intelligence and Statistics, AISTATS 2019, 16-18 April 2019, Naha, Okinawa, Japan*, volume 89 of *Proceedings of Machine Learning Research*, pages 2057–2066. PMLR, 2019.

[72] Xianzhong Cui and Kang G. Shin. Direct Control and Coordination Using Neural Networks. *IEEE Transactions on Systems, Man, and Cybernetics*, 23(3):686–697, 1993.

[73] Przemyslaw Daca, Thomas A. Henzinger, Jan Křetínský, and Tatjana Petrov. Faster Statistical Model Checking for Unbounded Temporal Properties. In Marsha Chechik and Jean-François Raskin, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, volume 9636 of *Lecture Notes in Computer Science*, pages 112–129. Springer, 2016.

[74] Pedro R. D'Argenio, Arnd Hartmanns, and Sean Sedwards. Lightweight Statistical Model Checking in Nondeterministic Continuous Time. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Verification - 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part II*, volume 11245 of *Lecture Notes in Computer Science*, pages 336–353. Springer, 2018.

[75] Alexandre David, Kim G. Larsen, Axel Legay, Marius Mikucionis, and Zheng Wang. Time for Statistical Model Checking of Real-Time Systems. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 349–355. Springer, 2011.

[76] Luca de Alfaro, Marta Z. Kwiatkowska, Gethin Norman, David Parker, and Roberto Segala. Symbolic Model Checking of Probabilistic Processes Using MTBDDs and the Kronecker Representation. In Susanne Graf and Michael I. Schwartzbach, editors, *Tools and Algorithms for Construction and Analysis of Systems, 6th International Conference, TACAS 2000, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000, Berlin, Germany, March 25 - April 2, 2000, Proceedings*, volume 1785 of *Lecture Notes in Computer Science*, pages 395–410. Springer, 2000.

[77] Christian Dehnert, Sebastian Junges, Joost-Pieter Katoen, and Matthias Volk. A Storm is Coming: A Modern Probabilistic Model Checker. In Rupak Majumdar and Viktor Kuncak, editors, *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*, volume 10427 of *Lecture Notes in Computer Science*, pages 592–600. Springer, 2017.

[78] Benoît Delahaye, Axel Legay, and Sean Sedwards. A Simple and Efficient Statistical Model Checking Algorithm to Evaluate Markov Decision Processes. 2013. Technical Report, hal-00856704.

[79] Nilanjan Dey, Amira S. Ashour, Fuqian Shi, Simon James Fong, and João Manuel RS. Tavares. Medical Cyber-Physical Systems: A Survey. *Journal of Medical Systems*, 42(4):1–13, 2018.

[80] Quang Do, Ben Martini, and Kim-Kwang Raymond Choo. Cyber-Physical Systems Information Gathering: A Smart Home Case Study. *Computer Networks*, 138:1–12, 2018.

[81] Carmel Domshlak and Jörg Hoffmann. Probabilistic Planning via Heuristic Forward Search and Weighted Model Counting. *Journal of Artificial Intelligence Research*, 30:565–620, 2007.

[82] Finale Doshi-Velez and Been Kim. Towards a Rigorous Science of Interpretable Machine Learning. *arXiv preprint arXiv:1702.08608*, 2017.

[83] Stefan Edelkamp. Promela Planning. In Thomas Ball and Sriram K. Rajamani, editors, *Model Checking Software, 10th International SPIN Workshop. Portland, OR, USA, May 9-10, 2003, Proceedings*, volume 2648 of *Lecture Notes in Computer Science*, pages 197–212. Springer, 2003.

[84] Stefan Edelkamp, Alberto Lluch-Lafuente, and Stefan Leue. Directed Explicit Model Checking with HSF-SPIN. In Matthew B. Dwyer, editor, *Model Checking Software, 8th International SPIN Workshop, Toronto, Canada, May 19-20, 2001, Proceedings*, volume 2057 of *Lecture Notes in Computer Science*, pages 57–79. Springer, 2001.

[85] Rüdiger Ehlers. Formal Verification of Piece-Wise Linear Feed-Forward Neural Networks. 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 269–286. Springer, 2017.

[86] 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.

[87] Rasha Faqeh, Christof Fetzer, Holger Hermanns, Jörg Hoffmann, Michaela Klauck, Maximilian A. Köhl, Marcel Steinmetz, and Christoph Weidenbach. Towards Dynamic Dependable Systems Through Evidence-Based Continuous Certification. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Engineering Principles - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part II*, volume 12477 of *Lecture Notes in Computer Science*, pages 416–439. Springer, 2020.

[88] Willliam Feller. *An Introduction to Probability Theory and Its Applications, Vol. 2*. John Wiley & Sons, 2008.

[89] Di Feng, Christian Haase-Schütz, Lars Rosenbaum, Heinz Hertlein, Claudius Gläser, Fabian Timm, Werner Wiesbeck, and Klaus Dietmayer. Deep Multi-Modal Object Detection and Semantic Segmentation for Autonomous Driving: Datasets, Methods, and Challenges. *IEEE Transactions on Intelligent Transportation Systems*, 22(3):1341–1360, 2021.

[90] Vojtech Forejt, Marta Z. Kwiatkowska, Gethin Norman, and David Parker. Automated Verification Techniques for Probabilistic Systems. In Marco Bernardo and Valérie Issarny, editors, *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*, volume 6659 of *Lecture Notes in Computer Science*, pages 53–113. Springer, 2011.

[91] Vincent François-Lavet, Peter Henderson, Riashat Islam, Marc G. Bellemare, and Joelle Pineau. An Introduction to Deep Reinforcement Learning. *Foundations and Trends in Machine Learning*, 11(3-4):219–354, 2018.

[92] Chen Fu, Ernst Moritz Hahn, Yong Li, Sven Schewe, Meng Sun, Andrea Turrini, and Lijun Zhang. EPMC Gets Knowledge in Multi-agent Systems. In Bernd Finkbeiner and Thomas Wies, editors, *Verification, Model Checking, and Abstract Interpretation - 23rd International Conference, VMCAI 2022, Philadelphia, PA, USA, January 16-18, 2022, Proceedings*, volume 13182 of *Lecture Notes in Computer Science*, pages 93–107. Springer, 2022.

[93] Masahiro Fujita, Patrick C. McGeer, and Jerry Chih-Yuan Yang. Multi-Terminal Binary Decision Diagrams: An Efficient Data Structure for Matrix Representation. *Formal Methods in System Design*, 10(2/3):149–169, 1997.

[94] Hubert Garavel, Maurice H. ter Beek, and Jaco van de Pol. The 2020 Expert Survey on Formal Methods. In Maurice H. ter Beek and Dejan Nickovic, editors, *Formal Methods for Industrial Critical Systems - 25th International Conference, FMICS 2020, Vienna, Austria, September 2-3, 2020, Proceedings*, volume 12327 of *Lecture Notes in Computer Science*, pages 3–69. Springer, 2020.

[95] Javier García and Fernando Fernández. A Comprehensive Survey on Safe Reinforcement Learning. *Journal of Machine Learning Research*, 16:1437–1480, 2015.

[96] Martin Gardner. Mathematical Games. *Scientific American*, 229:118–121, 1973.

[97] M.W Gardner and S.R Dorling. Artificial Neural Networks (the multilayer perceptron)—A Review of Applications in the Atmospheric Sciences. *Atmospheric Environment*, 32(14):2627 – 2636, 1998.

[98] Hector Geffner and Blai Bonet. *A Concise Introduction to Models and Methods for Automated Planning*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2013.

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

[100] Shalini Ghosh, Amaury Mercier, Dheeraj Pichapati, Susmit Jha, Vinod Yegneswaran, and Patrick Lincoln. Trusted Neural Networks for Safety-Constrained Autonomous Control. *CoRR*, abs/1805.07075, 2018.

[101] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer, 1996.

[102] Timo P. Gros. Tracking the Race: Analyzing Racetrack Agents Trained with Imitation Learning and Deep Reinforcement Learning. Master's thesis, Saarland University, May 2021.

[103] Timo P. Gros, David Groß, Stefan Gumhold, Jörg Hoffmann, Michaela Klauck, and Marcel Steinmetz. TraceVis: Towards Visualization for Deep Statistical Model Checking. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Tools and Trends - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part IV*, volume 12479 of *Lecture Notes in Computer Science*, pages 27–46. Springer, 2020.

[104] Timo P. Gros, David Groß, Stefan Gumhold, Jörg Hoffmann, Michaela Klauck, and Marcel Steinmetz. TraceVis: Visualization for DSMC: tool, demonstration video, data, 2020. available at http://doi.org/10.5281/zenodo.3961196.

[105] Timo P. Gros, Joschka Groß, Daniel Höller, Jörg Hoffmann, Michaela Klauck, Hendrik Meerkamp, and Verena Wolf. DSMC Evaluation Stages: Fostering Robust and Safe Behavior in Deep Reinforcement Learning, 2022. under submission.

[106] Timo P. Gros, Holger Hermanns, Jörg Hoffmann, Michaela Klauck, and Marcel Steinmetz. Deep Statistical Model Checking. In Alexey Gotsman and Ana Sokolova, editors, *Formal Techniques for Distributed Objects, Components, and Systems - 40th IFIP WG 6.1 International Conference, FORTE 2020, Held as Part of the 15th International Federated Conference on Distributed Computing Techniques, DisCoTec 2020, Valletta, Malta, June 15-19, 2020, Proceedings*, volume 12136 of *Lecture Notes in Computer Science*, pages 96–114. Springer, 2020.

[107] Timo P. Gros, Holger Hermanns, Jörg Hoffmann, Michaela Klauck, Maximilian A. Köhl, and Verena Wolf. MoGym: Using Formal Models for Training and Verifying Decision-making Agents. In *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings*, 2022.

[108] Timo P. Gros, Holger Hermanns, Jörg Hoffmann, Michaela Klauck, and Marcel Steinmetz. Models and Infrastructure used in "Deep Statistical Model Checking", 2020. available at http://doi.org/10.5281/zenodo.3760098.

[109] Timo P. Gros, Holger Hermanns, Jörg Hoffmann, Michaela Klauck, and Marcel Steinmetz. Analyzing Neural Network Behavior through Deep Statistical Model Checking, 2022. under submission.

[110] Timo P. Gros, Daniel Höller, Jörg Hoffmann, Michaela Klauck, Hendrik Meerkamp, and Verena Wolf. DSMC Evaluation Stages: Fostering Robust and Safe Behavior in Deep Reinforcement Learning. In Alessandro Abate and Andrea Marin, editors, *Quantitative Evaluation of Systems - 18th International Conference, QEST 2021, Paris, France, August 23-27, 2021, Proceedings*, volume 12846 of *Lecture Notes in Computer Science*, pages 197–216. Springer, 2021.

[111] Timo P. Gros, Daniel Höller, Jörg Hoffmann, and Verena Wolf. Tracking the Race Between Deep Reinforcement Learning and Imitation Learning. In Marco Gribaudo, David N. Jansen, and Anne Remke, editors, *Quantitative Evaluation of Systems - 17th International Conference, QEST 2020, Vienna, Austria, August 31 - September 3, 2020, Proceedings*, volume 12289 of *Lecture Notes in Computer Science*, pages 11–17. Springer, 2020.

[112] Radu Grosu and Scott A. Smolka. Monte Carlo Model Checking. In Nicolas Halbwachs and Lenore D. Zuck, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, volume 3440 of *Lecture Notes in Computer Science*, pages 271–286. Springer, 2005.

[113] David Groß, Michaela Klauck, Timo P. Gros, Marcel Steinmetz, Jörg Hoffmann, and Stefan Gumhold. Glyph-based visual analysis of q-learning based action policy ensembles on racetrack. In *26th International Conference on Information Visualisation (IV)*, 2022.

[114] Stefan Gumhold. The Computer Graphics and Visualization Framework. https://github.com/sgumhold/cgv. Accessed: 18-05-2020.

[115] Niklas Gustafsson et al. TorchSharp. https://github.com/dotnet/TorchSharp, 2021. Accessed: 22-09-2021.

[116] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor. In Jennifer G. Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, volume 80 of *Proceedings of Machine Learning Research*, pages 1856–1865. PMLR, 2018.

[117] Serge Haddad and Benjamin Monmege. Interval Iteration Algorithm for MDPs and IMDPs. *Theoretical Computer Science*, 735:111–131, 2018.

[118] Sofie Haesaert, Sadegh Soudjani, and Alessandro Abate. Temporal Logic Control of General Markov Decision Processes by Approximate Policy Refinement. In Alessandro Abate, Antoine Girard, and Maurice Heemels, editors, *6th IFAC Conference on Analysis and Design of Hybrid Systems, ADHS 2018, Oxford, UK, July 11-13, 2018*, volume 51, Nr. 16 of *IFAC-PapersOnLine*, pages 73–78. Elsevier, 2018.

[119] Ernst Moritz Hahn and Arnd Hartmanns. A Comparison of Time- and Reward-Bounded Probabilistic Model Checking Techniques. In Martin Fränzle, Deepak Kapur, and Naijun Zhan, editors, *Dependable Software Engineering: Theories, Tools, and Applications - Second International Symposium, SETTA 2016, Beijing, China, November 9-11, 2016, Proceedings*, volume 9984 of *Lecture Notes in Computer Science*, pages 85–100, 2016.

[120] Ernst Moritz Hahn and Arnd Hartmanns. Efficient Algorithms for Time- and Cost-Bounded Probabilistic Model Checking. *CoRR*, abs/1605.05551, 2016.

[121] Ernst Moritz Hahn, Arnd Hartmanns, Christian Hensel, Michaela Klauck, Joachim Klein, Jan Křetínský, David Parker, Tim Quatmann, Enno Ruijters, and Marcel Steinmetz. The 2019 Comparison of Tools for the Analysis of Quantitative Formal Models - (QComp 2019 Competition Report). In Dirk Beyer, Marieke Huisman, Fabrice Kordon, and Bernhard Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part III*, volume 11429 of *Lecture Notes in Computer Science*, pages 69–92. Springer, 2019.

[122] Ernst Moritz Hahn, Arnd Hartmanns, and Holger Hermanns. Reachability and Reward Checking for Stochastic Timed Automata. *Electronic Communication of the European Association of Software Science and Technology*, 70, 2014.

[123] Ernst Moritz Hahn, Arnd Hartmanns, Holger Hermanns, and Joost-Pieter Katoen. A Compositional Modelling and Analysis Framework for Stochastic Hybrid Systems. *Formal Methods in System Design*, 43(2):191–232, 2013.

[124] Ernst Moritz Hahn, Yi Li, Sven Schewe, Andrea Turrini, and Lijun Zhang. IscasMC: A Web-Based Probabilistic Model Checker. In Cliff B. Jones, Pekka Pihlajasaari, and Jun Sun, editors, *FM 2014: Formal Methods - 19th International Symposium, Singapore, May 12-16, 2014. Proceedings*, volume 8442 of *Lecture Notes in Computer Science*, pages 312–317. Springer, 2014.

[125] Eric A. Hansen and Shlomo Zilberstein. LAO*: A Heuristic Search Algorithm That Finds Solutions with Loops. *Artificial Intelligence*, 129(1-2):35–62, 2001.

[126] Hans Hansson and Bengt Jonsson. A Logic for Reasoning about Time and Reliability. *Formal Aspects of Computing*, 6(5):512–535, 1994.

[127] Arnd Hartmanns. *On the Analysis of Stochastic Timed Systems*. PhD thesis, Saarland University, Germany, 2015.

[128] Arnd Hartmanns and Holger Hermanns. The Modest Toolset: An Integrated Environment for Quantitative Modelling and Verification. In Erika Ábrahám and Klaus Havelund, editors, *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*, volume 8413 of *Lecture Notes in Computer Science*, pages 593–598. Springer, 2014.

[129] Arnd Hartmanns and Holger Hermanns. Explicit Model Checking of Very Large MDP Using Partitioning and Secondary Storage. In Bernd Finkbeiner, Geguang Pu, and Lijun Zhang, editors, *Automated Technology for Verification and Analysis - 13th International Symposium, ATVA 2015, Shanghai, China, October 12-15, 2015, Proceedings*, volume 9364 of *Lecture Notes in Computer Science*, pages 131–147. Springer, 2015.

[130] Arnd Hartmanns and Benjamin Lucien Kaminski. Optimistic Value Iteration. In Shuvendu K. Lahiri and Chao Wang, editors, *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II*, volume 12225 of *Lecture Notes in Computer Science*, pages 488–511. Springer, 2020.

[131] Arnd Hartmanns and Michaela Klauck. The 2020 Comparison of Tools for the Analysis of Quantitative Formal Models: Results and Reproduction, 2020. available at http://doi.org/10.5281/zenodo.3965312.

[132] Arnd Hartmanns, Michaela Klauck, David Parker, Tim Quatmann, and Enno Ruijters. The Quantitative Verification Benchmark Set. In Tomás Vojnar and Lijun Zhang, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April*

6-11, 2019, Proceedings, Part I, volume 11427 of *Lecture Notes in Computer Science*, pages 344–350. Springer, 2019.

[133] Arnd Hartmanns and Mark Timmer. On-the-Fly Confluence Detection for Statistical Model Checking. In Guillaume Brat, Neha Rungta, and Arnaud Venet, editors, *NASA Formal Methods, 5th International Symposium, NFM 2013, Moffett Field, CA, USA, May 14-16, 2013. Proceedings*, volume 7871 of *Lecture Notes in Computer Science*, pages 337–351. Springer, 2013.

[134] Arnd Hartmanns and Mark Timmer. Sound Statistical Model Checking for MDP Using Partial Order and Confluence Reduction. *International Journal on Software Tools for Technology Transfer*, 17(4):429–456, 2015.

[135] Mohammadhosein Hasanbeig, Alessandro Abate, and Daniel Kroening. Logically-Correct Reinforcement Learning. *CoRR*, abs/1801.08099, 2018.

[136] Hassan Hatefi-Ardakani. *Finite Horizon Analysis of Markov Automata*. PhD thesis, Saarland University, Germany, 2017.

[137] Matthew J. Hausknecht and Peter Stone. Deep Recurrent Q-Learning for Partially Observable MDPs. In *2015 AAAI Fall Symposia, Arlington, Virginia, USA, November 12-14, 2015*, pages 29–37. AAAI Press, 2015.

[138] Malte Helmert. The Fast Downward Planning System. *CoRR*, abs/1109.6051, 2011.

[139] David Henriques, João G. Martins, Paolo Zuliani, André Platzer, and Edmund M. Clarke. Statistical Model Checking for Markov Decision Processes. In *Ninth International Conference on Quantitative Evaluation of Systems, QEST 2012, London, United Kingdom, September 17-20, 2012*, pages 84–93. IEEE Computer Society, 2012.

[140] Christian Hensel. *The Probabilistic Model Checker Storm: Symbolic Methods for Probabilistic Model Checking*. PhD thesis, RWTH Aachen University, Germany, 2018.

[141] Thomas Hérault, Richard Lassaigne, Frédéric Magniette, and Sylvain Peyronnet. Approximate Probabilistic Model Checking. In Bernhard Steffen and Giorgio Levi, editors, *Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004, Venice, Italy, January 11-13, 2004, Proceedings*, volume 2937 of *Lecture Notes in Computer Science*, pages 73–84. Springer, 2004.

[142] Holger Hermanns, Marta Z. Kwiatkowska, Gethin Norman, David Parker, and Markus Siegle. On the Use of MTBDDs for Performability Analysis and Verification of Stochastic Systems. *Journal of Logical and Algebraic Methods in Programming*, 56(1-2):23–67, 2003.

[143] Holger Hermanns, Joachim Meyer-Kayser, and Martin Siegle. Multi Terminal Binary Decision Diagrams to Represent and Analyse Continuous Time Markov Chains. In B. Plateau, W. Stewart, and M. Silva, editors, *Proc. 3rd International Workshop on Numerical Solution of Markov Chains (NSMC'99)*, pages 188–207. Prensas Universitarias de Zaragoza, 1999.

[144] Holger Hermanns, Björn Wachter, and Lijun Zhang. Probabilistic CEGAR. In Aarti Gupta and Sharad Malik, editors, *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings*, volume 5123 of *Lecture Notes in Computer Science*, pages 162–175. Springer, 2008.

[145] Josefa Z Hernández, Sascha Ossowski, and Ana Garcıa-Serrano. Multiagent Architectures for Intelligent Traffic Management Systems. *Transportation Research Part C: Emerging Technologies*, 10(5-6):473–506, 2002.

[146] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, and B. Kingsbury. Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups. *IEEE Signal Processing Magazine*, 29(6):82–97, 2012.

[147] Jonathan Ho and Stefano Ermon. Generative Adversarial Imitation Learning. In Daniel D. Lee, Masashi Sugiyama, Ulrike von Luxburg, Isabelle Guyon, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, pages 4565–4573, 2016.

[148] Jörg Hoffmann, Holger Hermanns, Michaela Klauck, Marcel Steinmetz, Erez Karpas, and Daniele Magazzeni. Let's Learn Their Language? A Case for Planning with Automata-Network Languages from Model Checking. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pages 13569–13575. AAAI Press, 2020.

[149] Fred Hohman, Minsuk Kahng, Robert Pienta, and Duen Horng Chau. Visual Analytics in Deep Learning: An Interrogative Survey for the Next Frontiers. *IEEE Transactions on Visualization and Computer Graphics*, 25(8):2674–2693, 2019.

[150] Gerard J. Holzmann. *The SPIN Model Checker - Primer and Reference Manual*. Addison-Wesley, 2004.

[151] Taekeun Hong, Jin-A. Choi, Kiho Lim, and Pankoo Kim. Enhancing Personalized Ads Using Interest Category Classification of SNS Users Based on Deep Neural Networks. *Sensors*, 21(1):199, 2021.

[152] Kurt Hornik, Maxwell B. Stinchcombe, and Halbert White. Multilayer Feedforward Networks are Universal Approximators. *Neural Networks*, 2:359–366, 1989.

[153] R.A. Howard. *Dynamic Probabilistic Systems: Markov Models*, volume 1 of *Dover Books on Mathematics*. Dover Publications, 2012.

[154] R.A. Howard. *Dynamic Probabilistic Systems: Semi-Markov and Decision Processes*, volume 2 of *Dover Books on Mathematics*. Dover Publications, 2013.

[155] Ronald A Howard. *Dynamic Programming and Markov Processes.* John Wiley, 1960.

[156] Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. Safety Verification of Deep Neural Networks. In Rupak Majumdar and Viktor Kuncak, editors, *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I*, volume 10426 of *Lecture Notes in Computer Science*, pages 3–29. Springer, 2017.

[157] Marieke Huisman, Dilian Gurov, and Alexander Malkis. Formal Methods: From Academia to Industrial Practice. A Travel Guide. *CoRR*, abs/2002.07279, 2020.

[158] Olle Häggström. *Finite Markov Chains and Algorithmic Applications*. London Mathematical Society Student Texts. Cambridge University Press, 2002.

[159] Amos Israeli and Marc Jalfon. Token Management Schemes and Random Walks Yield Self-Stabilizing Mutual Exclusion. In Cynthia Dwork, editor, *Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing, Quebec City, Quebec, Canada, August 22-24, 1990*, pages 119–131. ACM, 1990.

[160] Masoumeh T. Izadi. Sequential Decision Making Under Uncertainty. In Jean-Daniel Zucker and Lorenza Saitta, editors, *Abstraction, Reformulation and Approximation, 6th International Symposium, SARA 2005, Airth Castle, Scotland, UK, July 26-29, 2005, Proceedings*, volume 3607 of *Lecture Notes in Computer Science*, pages 360–361. Springer, 2005.

[161] The JANI Specification. https://www.jani-spec.org/. Accessed: 24-02-2022.

[162] Nasser Jazdi. Cyber Physical Systems in the Context of Industry 4.0. In *2014 IEEE International Conference on Automation, Quality and Testing, Robotics*, pages 1–4. IEEE, 2014.

[163] Cyrille Jégourel, Axel Legay, and Sean Sedwards. A Platform for High Performance Statistical Model Checking - PLASMA. In Cormac Flanagan and Barbara König, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, volume 7214 of *Lecture Notes in Computer Science*, pages 498–503. Springer, 2012.

[164] Sebastian Junges. *Parameter Synthesis in Markov models*. PhD thesis, RWTH Aachen University, Germany, 2020.

[165] Sebastian Junges, Nils Jansen, Christian Dehnert, Ufuk Topcu, and Joost-Pieter Katoen. Safety-Constrained Reinforcement Learning for MDPs. In Marsha Chechik and Jean-François Raskin, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, volume 9636 of *Lecture Notes in Computer Science*, pages 130–146. Springer, 2016.

[166] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement Learning: A Survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[167] Lukasz Kaiser, Mohammad Babaeizadeh, Piotr Milos, Blazej Osinski, Roy H. Campbell, Konrad Czechowski, Dumitru Erhan, Chelsea Finn, Piotr Kozakowski, Sergey Levine, Afroz Mohiuddin, Ryan Sepassi, George Tucker, and Henryk Michalewski. Model Based Reinforcement Learning for Atari. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020.

[168] 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.

[169] Joost-Pieter Katoen and Ivan S. Zapreev. Simulation-Based CTMC Model Checking: An Empirical Evaluation. In *QEST 2009, Sixth International Conference on the Quantitative Evaluation of Systems, Budapest, Hungary, 13-16 September 2009*, pages 31–40. IEEE Computer Society, 2009.

[170] Guy Katz, Clark W. Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. In Rupak Majumdar and Viktor Kuncak, editors, *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I*, volume 10426 of *Lecture Notes in Computer Science*, pages 97–117. Springer, 2017.

[171] Michaela Klauck. Bridging Concurrency Models to Probabilistic Planning and Implementing FRET-π LRTDP in the Modest Toolset. Master's thesis, Saarland University, March 2018.

[172] Michaela Klauck. Artifact of Infrastructure and Tools in the Context of Deep Statistical Model Checking, 2022. available at http://doi.org/10.5281/zenodo.6362696.

[173] Michaela Klauck and Holger Hermanns. A Modest Approach to Dynamic Heuristic Search in Probabilistic Model Checking. In Alessandro Abate and Andrea Marin, editors, *Quantitative Evaluation of Systems - 18th International Conference, QEST 2021, Paris, France, August 23-27, 2021, Proceedings*, volume 12846 of *Lecture Notes in Computer Science*, pages 15–38. Springer, 2021.

[174] Michaela Klauck and Holger Hermanns. Artifact accompanying the paper "A Modest Approach to Dynamic Heuristic Search in Probabilistic Model Checking", 2021. available at http://doi.org/10.5281/zenodo.4922360.

[175] Michaela Klauck, Marcel Steinmetz, Jörg Hoffmann, and Holger Hermanns. Compiling Probabilistic Model Checking into Probabilistic Planning. In Mathijs de Weerdt, Sven Koenig, Gabriele Röger, and Matthijs T. J. Spaan, editors, *Proceedings of the Twenty-Eighth International Conference on Automated Planning and Scheduling, ICAPS 2018, Delft, The Netherlands, June 24-29, 2018*, pages 150–154. AAAI Press, 2018.

[176] Michaela Klauck, Marcel Steinmetz, Jörg Hoffmann, and Holger Hermanns. Bridging the Gap Between Probabilistic Model Checking and Probabilistic Planning: Survey, Compilations, and Empirical Comparison. *Journal of Artificial Intelligence Research*, 68:247–310, 2020.

[177] Joachim Klein, Christel Baier, Philipp Chrszon, Marcus Daum, Clemens Dubslaff, Sascha Klüppelholz, Steffen Märcker, and David Müller. Advances in Symbolic Probabilistic Model Checking with PRISM. In Marsha Chechik and Jean-François Raskin, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, volume 9636 of *Lecture Notes in Computer Science*, pages 349–366. Springer, 2016.

[178] Maximilian A. Köhl, Michaela Klauck, and Holger Hermanns. Momba: JANI Meets Python. In Jan Friso Groote and Kim Guldstrand Larsen, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part II*, volume 12652 of *Lecture Notes in Computer Science*, pages 389–398. Springer, 2021.

[179] Andrey Kolobov. *Scalable Methods and Expressive Models for Planning Under Uncertainty*. PhD thesis, University of Washington, 2013.

[180] Andrey Kolobov, Mausam, Daniel S. Weld, and Hector Geffner. Heuristic Search for Generalized Stochastic Shortest Path MDPs. In Fahiem Bacchus, Carmel Domshlak, Stefan Edelkamp, and Malte Helmert, editors, *Proceedings of the 21st International Conference on Automated Planning and Scheduling, ICAPS 2011, Freiburg, Germany June 11-16, 2011*. AAAI, 2011.

[181] Saul A Kripke. Semantical Considerations on Modal Logic. *Acta Philosophica Fennica*, 16, 1963.

[182] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In Peter L. Bartlett, Fernando C. N. Pereira, Christopher J. C. Burges, Léon Bottou, and Kilian Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States*, pages 1106–1114, 2012.

[183] Divya Kumar and Krishn Kumar Mishra. The Impacts of Test Automation on Software's Cost, Quality and Time to Market. *Procedia Computer Science*, 79:8–15, 2016.

[184] Nicholas Kushmerick, Steve Hanks, and Daniel S. Weld. An Algorithm for Probabilistic Planning. *Artificial Intelligence*, 76(1-2):239–286, 1995.

[185] Jan Křetínský and Tobias Meggendorfer. Efficient Strategy Iteration for Mean Payoff in Markov Decision Processes. 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 380–399. Springer, 2017.

[186] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. Probabilistic Symbolic Model Checking with PRISM: a Hybrid Approach. *International Journal on Software Tools for Technology Transfer*, 6(2):128–142, 2004.

[187] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. Symmetry Reduction for Probabilistic Model Checking. In Thomas Ball and Robert B. Jones, editors, *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4144 of *Lecture Notes in Computer Science*, pages 234–248. Springer, 2006.

[188] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. Stochastic Model Checking. In Marco Bernardo and Jane Hillston, editors, *Formal Methods for Performance Evaluation, 7th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2007, Bertinoro, Italy, May 28-June 2, 2007, Advanced Lectures*, volume 4486 of *Lecture Notes in Computer Science*, pages 220–270. Springer, 2007.

[189] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. PRISM 4.0: Verification of Probabilistic Real-Time Systems. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 585–591. Springer, 2011.

[190] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. The PRISM Benchmark Suite. In *Ninth International Conference on Quantitative Evaluation of Systems, QEST 2012, London, United Kingdom, September 17-20, 2012*, pages 203–204. IEEE Computer Society, 2012.

[191] Marta Z. Kwiatkowska, Gethin Norman, and Jeremy Sproston. Probabilistic Model Checking of the IEEE 802.11 Wireless Local Area Network Protocol. In Holger Hermanns and Roberto Segala, editors, *Process Algebra and Probabilistic Methods, Performance Modeling and Verification, Second Joint International Workshop PAPM-PROBMIV 2002, Copenhagen, Denmark, July 25-26, 2002, Proceedings*, volume 2399 of *Lecture Notes in Computer Science*, pages 169–187. Springer, 2002.

[192] Maksim Lapin, Matthias Hein, and Bernt Schiele. Top-k Multiclass SVM. In Corinna Cortes, Neil D. Lawrence, Daniel D. Lee, Masashi Sugiyama, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, pages 325–333, 2015.

[193] Maksim Lapin, Matthias Hein, and Bernt Schiele. Loss Functions for Top-k Error: Analysis and Insights. In *2016 IEEE Conference on Computer Vision and Pattern*

Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016, pages 1468–1477. IEEE Computer Society, 2016.

[194] Kim G. Larsen and Axel Legay. Statistical Model Checking: Past, Present, and Future. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part I*, volume 9952 of *Lecture Notes in Computer Science*, pages 3–15, 2016.

[195] Pierre L'Ecuyer, François LeGland, Pascal Lezaud, and Bruno Tuffin. Splitting Techniques. In Gerardo Rubino and Bruno Tuffin, editors, *Rare Event Simulation using Monte Carlo Methods*, pages 39–61. Wiley, 2009.

[196] Insup Lee and Oleg Sokolsky. Medical Cyber Physical Systems. In *Design Automation Conference*, pages 743–748. IEEE, 2010.

[197] Axel Legay, Benoît Delahaye, and Saddek Bensalem. Statistical Model Checking: An Overview. In Howard Barringer, Yliès Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon J. Pace, Grigore Rosu, Oleg Sokolsky, and Nikolai Tillmann, editors, *Runtime Verification - First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings*, volume 6418 of *Lecture Notes in Computer Science*, pages 122–135. Springer, 2010.

[198] Axel Legay, Anna Lukina, Louis-Marie Traonouez, Junxing Yang, Scott A. Smolka, and Radu Grosu. Statistical Model Checking. In Bernhard Steffen and Gerhard J. Woeginger, editors, *Computing and Software Science - State of the Art and Perspectives*, volume 10000 of *Lecture Notes in Computer Science*, pages 478–504. Springer, 2019.

[199] Axel Legay, Sean Sedwards, and Louis-Marie Traonouez. Scalable Verification of Markov Decision Processes. In Carlos Canal and Akram Idani, editors, *Software Engineering and Formal Methods - SEFM 2014 Collocated Workshops: HOFM, SAFOME, OpenCert, MoKMaSD, WS-FMDS, Grenoble, France, September 1-2, 2014, Revised Selected Papers*, volume 8938 of *Lecture Notes in Computer Science*, pages 350–362. Springer, 2014.

[200] Daniel Lehmann and Michael O. Rabin. On the Advantages of Free Choice: A Symmetric and Fully Distributed Solution to the Dining Philosophers Problem. In John White, Richard J. Lipton, and Patricia C. Goldberg, editors, *Conference Record of the Eighth Annual ACM Symposium on Principles of Programming Languages, Williamsburg, Virginia, USA, January 1981*, pages 133–138. ACM Press, 1981.

[201] Jianlin Li, Jiangchao Liu, Pengfei Yang, Liqian Chen, Xiaowei Huang, and Lijun Zhang. Analyzing Deep Neural Networks with Symbolic Propagation: Towards Higher Precision and Faster Verification. In Bor-Yuh Evan Chang, editor, *Static Analysis - 26th International Symposium, SAS 2019, Porto, Portugal, October 8-11, 2019, Proceedings*, volume 11822 of *Lecture Notes in Computer Science*, pages 296–319. Springer, 2019.

[202] Renjue Li, Jianlin Li, Cheng-Chao Huang, Pengfei Yang, Xiaowei Huang, Lijun Zhang, Bai Xue, and Holger Hermanns. PRODeep: a platform for robustness verification of deep neural networks. In Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann, editors, *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, pages 1630–1634. ACM, 2020.

[203] Michael L. Littman, Judy Goldsmith, and Martin Mundhenk. The Computational Complexity of Probabilistic Planning. *Journal of Artificial Intelligence Research*, 9:1–36, 1998.

[204] H. Brendan McMahan and Geoffrey J. Gordon. Fast Exact Planning in Markov Decision Processes. In Susanne Biundo, Karen L. Myers, and Kanna Rajan, editors, *Proceedings of the Fifteenth International Conference on Automated Planning and Scheduling (ICAPS 2005), June 5-10 2005, Monterey, California, USA*, pages 151–160. AAAI, 2005.

[205] H. Brendan McMahan, Maxim Likhachev, and Geoffrey J. Gordon. Bounded Real-Time Dynamic Programming: RTDP with Monotone Upper Bounds and Performance Guarantees. In Luc De Raedt and Stefan Wrobel, editors, *Machine Learning, Proceedings of the Twenty-Second International Conference (ICML 2005), Bonn, Germany, August 7-11, 2005*, volume 119 of *ACM International Conference Proceeding Series*, pages 569–576. ACM, 2005.

[206] Kenneth L. McMillan. The SMV Language, 1998. Cadence Berkeley Labs.

[207] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

[208] Andrew S. Miner and David Parker. Symbolic Representations and Analysis of Large Probabilistic Systems. In Christel Baier, Boudewijn R. Haverkort, Holger Hermanns, Joost-Pieter Katoen, and Markus Siegle, editors, *Validation of Stochastic Systems - A Guide to Current Research*, volume 2925 of *Lecture Notes in Computer Science*, pages 296–338. Springer, 2004.

[209] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin A. Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level Control Through Deep Reinforcement Learning. *Nature*, 518:529–533, 2015.

[210] Thomas M. Moerland, Joost Broekens, and Catholijn M. Jonker. Model-based Reinforcement Learning: A Survey. *CoRR*, abs/2006.16712, 2020.

[211] Christopher Z Mooney. *Monte Carlo Simulation*. 116. Sage, 1997.

[212] Dana S. Nau. Current Trends in Automated Planning. *AI Magazine*, 28(4):43–58, 2007.

[213] Michael Negnevitsky. *Artificial Intelligence: A Guide to Intelligent Systems*. Pearson Education, 2005.

[214] Gethin Norman, David Parker, and Jeremy Sproston. Model Checking for Probabilistic Timed Automata. *Formal Methods in System Design*, 43(2):164–190, 2013.

[215] Masashi Okamoto. Some Inequalities Relating to the Partial Sum of Binomial Probabilities. *Annals of the Institute of Statistical Mathematics*, 10(1):29–35, 1959.

[216] David Anthony Parker. *Implementation of Symbolic Model Checking for Probabilistic Systems*. PhD thesis, University of Birmingham, UK, 2003.

[217] Deepak Pathak, Pulkit Agrawal, Alexei A. Efros, and Trevor Darrell. Curiosity-driven Exploration by Self-supervised Prediction. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, volume 70 of *Proceedings of Machine Learning Research*, pages 2778–2787. PMLR, 2017.

[218] Fabio Patrizi, Nir Lipovetzky, Giuseppe De Giacomo, and Hector Geffner. Computing Infinite Plans for LTL Goals Using a Classical Planner. In Toby Walsh, editor, *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16-22, 2011*, pages 2003–2008. IJCAI/AAAI, 2011.

[219] Doron A. Peled. Combining Partial Order Reductions with On-the-fly Model-Checking. In David L. Dill, editor, *Computer Aided Verification, 6th International Conference, CAV '94, Stanford, California, USA, June 21-23, 1994, Proceedings*, volume 818 of *Lecture Notes in Computer Science*, pages 377–390. Springer, 1994.

[220] Luis Enrique Pineda, Yi Lu, Shlomo Zilberstein, and Claudia V. Goldman. Fault-Tolerant Planning under Uncertainty. In Francesca Rossi, editor, *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*, pages 2350–2356. IJCAI/AAAI, 2013.

[221] Luis Enrique Pineda and Shlomo Zilberstein. Planning Under Uncertainty Using Reduced Models: Revisiting Determinization. In Steve A. Chien, Minh Binh Do, Alan Fern, and Wheeler Ruml, editors, *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling, ICAPS 2014, Portsmouth, New Hampshire, USA, June 21-26, 2014*. AAAI, 2014.

[222] Strategic Planning. The Economic Impacts of Inadequate Infrastructure for Software Testing. *National Institute of Standards and Technology*, 2002.

[223] André Platzer. *Logical Foundations of Cyber-Physical Systems*. Springer, 2018.

[224] Amir Pnueli. The Temporal Logic of Programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 46–57. IEEE Computer Society, 1977.

[225] Amir Pnueli and Lenore D. Zuck. Verification of Multiprocess Probabilistic Protocols. *Distributed Computing*, 1(1):53–72, 1986.

[226] Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley Series in Probability and Statistics. Wiley, 1994.

[227] PyTorch. https://pytorch.org/. Accessed: 22-09-2021.

[228] Tim Quatmann and Joost-Pieter Katoen. Sound Value Iteration. 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 643–661. Springer, 2018.

[229] Michael O. Rabin. N-Process Mutual Exclusion with Bounded Waiting by 4 Log_2 N-Valued Shared Variable. *Journal of Computer and System Sciences*, 25(1):66–75, 1982.

[230] Vahid Rafe, Mohsen Rahmani, and Kianoosh Rashidi. A Survey on Coping with the State Space Explosion Problem in Model Checking. *International Research Journal of Applied and Basic Sciences*, 4(6):1379–1384, 2013.

[231] Rohit Raj and Nandini Rai. Voice Controlled Cyber-Physical System for Smart Home. In Doina Bein, editor, *Proceedings of the Workshop Program of the 19th International Conference on Distributed Computing and Networking, Varanasi, India, January 04-07, 2018*, pages 28:1–28:5. ACM, 2018.

[232] Vennila Ramalingam, B. Palaniappan, N. Panchanatham, and S. Palanivel. Measuring Advertisement Effectiveness - A Neural Network Approach. *Expert Systems with Applications*, 31(1):159–163, 2006.

[233] Samik Raychaudhuri. Introduction to Monte Carlo Simulation. In Scott J. Mason, Raymond R. Hill, Lars Mönch, Oliver Rose, Thomas Jefferson, and John W. Fowler, editors, *Proceedings of the 2008 Winter Simulation Conference, Global Gateway to Discovery, WSC 2008, InterContinental Hotel, Miami, Florida, USA, December 7-10, 2008*, pages 91–100. WSC, 2008.

[234] Sheldon M Ross. *A First Course in Probability*, volume 8. Pearson Prentice Hall Upper Saddle River, NJ, 2010.

[235] Sebastian Ruder. An Overview of Gradient Descent Optimization Algorithms. *CoRR*, abs/1609.04747, 2016.

[236] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (4th Edition)*. Pearson, 2020.

[237] Duygu Sarikaya, Jason J. Corso, and Khurshid A. Guru. Detection and Localization of Robotic Tools in Robot-Assisted Surgery Videos Using Deep Neural Networks for Region Proposal and Detection. *IEEE Transactions on Medical Imaging*, 36(7):1542–1549, 2017.

[238] Warren S Sarle. Neural Networks and Statistical Models. In *Proceedings of the Nineteenth Annual SAS Users Group International Conference*, 1994.

[239] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal Policy Optimization Algorithms. *CoRR*, abs/1707.06347, 2017.

[240] Koushik Sen, Mahesh Viswanathan, and Gul Agha. On Statistical Model Checking of Stochastic Systems. In Kousha Etessami and Sriram K. Rajamani, editors, *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*, volume 3576 of *Lecture Notes in Computer Science*, pages 266–280. Springer, 2005.

[241] Mennatullah Siam, Mostafa Gamal, Moemen Abdel-Razek, Senthil Kumar Yogamani, Martin Jägersand, and Hong Zhang. A Comparative Study of Real-Time Semantic Segmentation for Autonomous Driving. In *2018 IEEE Conference on Computer Vision and Pattern Recognition Workshops, CVPR Workshops 2018, Salt Lake City, UT, USA, June 18-22, 2018*, pages 587–597. Computer Vision Foundation / IEEE Computer Society, 2018.

[242] Joseph Sifakis. A Unified Approach for Studying the Properties of Transition Systems. *Theoretical Computer Science*, 18:227–258, 1982.

[243] David Silver, Aja Huang, Christopher J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature*, 529:484–503, 2016.

[244] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. A General Reinforcement Learning Algorithm that Masters Chess, shogi, and Go through self-play. *Science*, 362(6419):1140–1144, 2018.

[245] Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin T. Vechev. An Abstract Domain for Certifying Neural Networks. *Proceedings of the ACM on Programming Languages*, 3(POPL):41:1–41:30, 2019.

[246] Trey Smith and Reid G. Simmons. Focused Real-Time Dynamic Programming for MDPs: Squeezing More Out of a Heuristic. In *Proceedings, The Twenty-First National Conference on Artificial Intelligence and the Eighteenth Innovative Applications of Artificial Intelligence Conference, July 16-20, 2006, Boston, Massachusetts, USA*, pages 1227–1232. AAAI Press, 2006.

[247] Jonathan Sprauel, Andrey Kolobov, and Florent Teichteil-Königsbuch. Saturated Path-Constrained MDP: Planning under Uncertainty and Deterministic Model-Checking Constraints. In Carla E. Brodley and Peter Stone, editors, *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada*, pages 2367–2373. AAAI Press, 2014.

[248] Marcel Steinmetz, Jörg Hoffmann, and Olivier Buffet. Goal Probability Analysis in Probabilistic Planning: Exploring and Enhancing the State of the Art. *Journal of Artificial Intelligence Research*, 57:229–271, 2016.

[249] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning - An Introduction*. Adaptive Computation and Machine Learning. MIT Press, 1998.

[250] Robert Endre Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[251] Florent Teichteil-Königsbuch. Path-Constrained Markov Decision Processes: Bridging the Gap between Probabilistic Model-Checking and Decision-Theoretic Planning. In Luc De Raedt, Christian Bessiere, Didier Dubois, Patrick Doherty, Paolo Frasconi, Fredrik Heintz, and Peter J. F. Lucas, editors, *ECAI 2012 - 20th European Conference on Artificial Intelligence. Including Prestigious Applications of Artificial Intelligence (PAIS-2012) System Demonstrations Track, Montpellier, France, August 27-31 , 2012*, volume 242 of *Frontiers in Artificial Intelligence and Applications*, pages 744–749. IOS Press, 2012.

[252] Henk Tijms. *Understanding Probability*. Cambridge University Press, 2012.

[253] Mark Timmer, Mariëlle Stoelinga, and Jaco van de Pol. Confluence Reduction for Probabilistic Systems. In Parosh Aziz Abdulla and K. Rustan M. Leino, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 17th International Conference, TACAS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*, volume 6605 of *Lecture Notes in Computer Science*, pages 311–325. Springer, 2011.

[254] Claire J. Tomlin, George J. Pappas, and Shankar Sastry. Conflict Resolution for Air Traffic Management: A Study in Multiagent Hybrid Systems. *IEEE Transactions on Automatic Control.*, 43(4):509–521, 1998.

[255] Michael Treml, José Arjona-Medina, Thomas Unterthiner, Rupesh Durgesh, Felix Friedmann, Peter Schuberth, Andreas Mayr, Martin Heusel, Markus Hofmarcher, Michael Widrich, et al. Speeding up Semantic Segmentation for Autonomous Driving. In *NIPS 2016 Workshop - MLITS Machine Learning for Intelligent Transportation Systems, Neural Information Processing Systems 2016, Barcelona, Spain*, 2016.

[256] Felipe W. Trevizan, Florent Teichteil-Königsbuch, and Sylvie Thiébaux. Efficient Solutions for Stochastic Shortest Path Problems with Dead Ends. In Gal Elidan, Kristian Kersting, and Alexander T. Ihler, editors, *Proceedings of the Thirty-Third Conference on Uncertainty in Artificial Intelligence, UAI 2017, Sydney, Australia, August 11-15, 2017*. AUAI Press, 2017.

[257] Michael Ummels and Christel Baier. Computing Quantiles in Markov Reward Models. In Frank Pfenning, editor, *Foundations of Software Science and Computation Structures - 16th International Conference, FOSSACS 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 7794 of *Lecture Notes in Computer Science*, pages 353–368. Springer, 2013.

[258] Mauro Vallati, Lukás Chrpa, and Thomas Leo McCluskey. What you always wanted to know about the deterministic part of the International Planning Competition (IPC) 2014 (but were too afraid to ask). *Knowledge Engineering Review*, 33:e3, 2018.

[259] Martijn van Otterlo and Marco A. Wiering. Reinforcement Learning and Markov Decision Processes. In Marco A. Wiering and Martijn van Otterlo, editors, *Reinforcement Learning*, volume 12 of *Adaptation, Learning, and Optimization*, pages 3–42. Springer, 2012.

[260] Moshe Y. Vardi. Probabilistic Linear-Time Model Checking: An Overview of the Automata-Theoretic Approach. In Joost-Pieter Katoen, editor, *Formal Methods for Real-Time and Probabilistic Systems, 5th International AMAST Workshop, ARTS'99, Bamberg, Germany, May 26-28, 1999. Proceedings*, volume 1601 of *Lecture Notes in Computer Science*, pages 265–276. Springer, 1999.

[261] 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. IEEE Computer Society, 1986.

[262] Abhinav Verma, Vijayaraghavan Murali, Rishabh Singh, Pushmeet Kohli, and Swarat Chaudhuri. Programmatically Interpretable Reinforcement Learning. In Jennifer G. Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, volume 80 of *Proceedings of Machine Learning Research*, pages 5052–5061. PMLR, 2018.

[263] Oriol Vinyals, Igor Babuschkin, Wojciech M. Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H. Choi, Richard Powell, Timo Ewalds, Petko Georgiev, Junhyuk Oh, Dan Horgan, Manuel Kroiss, Ivo Danihelka, Aja Huang, Laurent Sifre, Trevor Cai, John P. Agapiou, Max Jaderberg, Alexander S. Vezhnevets, Remi Leblond, Tobias Pohlen, Valentin Dalibard, David Budden, Yury Sulsky, James Molloy, Tom L. Paine, Caglar Gulcehre, Ziyu Wang, Tobias Pfaff, Yuhuai Wu, Roman Ring, Dani Yogatama, Dario Wünsch, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy Lillicrap, Koray Kavukcuoglu, Demis Hassabis, Chris Apps, and David Silver. Grandmaster Level in StarCraft II Using Multi-Agent Reinforcement Learning. *Nature*, 2019.

[264] Marcel Vinzent and Jörg Hoffmann. Neural Network Action Policy Verification via Predicate Abstraction. 2021. PRL Workshop – Bridging the Gap Between AI Planning and Reinforcement Learning at the Thirty-First International Conference on Automated Planning and Scheduling, ICAPS 2021, Guangzhou, China (virtual), August 2-13, 2021.

[265] Daisuke Wakabayashi. Self-Driving Uber Car Kills Pedestrian in Arizona, Where Robots Roam. *The New York Times*, 2018.

[266] Abraham Wald. Sequential Tests of Statistical Hypotheses. *The Annals of Mathematical Statistics*, 16(2):117–186, 1945.

[267] Junpeng Wang, Liang Gou, Han-Wei Shen, and Hao Yang. DQNViz: A Visual Analytics Approach to Understand Deep Q-Networks. *IEEE Transactions on Visualization and Computer Graphics*, 25(1):288–298, 2019.

[268] Matthew Wicker, Xiaowei Huang, and Marta Kwiatkowska. Feature-Guided Black-Box Safety Testing of Deep Neural Networks. In Dirk Beyer and Marieke Huisman, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part I*, volume 10805 of *Lecture Notes in Computer Science*, pages 408–426. Springer, 2018.

[269] Bogdan M Wilamowski and J David Irwin. *Intelligent Systems*. CRC press, 2018.

[270] Sung Wook Yoon, Alan Fern, and Robert Givan. FF-Replan: A Baseline for Probabilistic Planning. In Mark S. Boddy, Maria Fox, and Sylvie Thiébaux, editors, *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling, ICAPS 2007, Providence, Rhode Island, USA, September 22-26, 2007*, page 352. AAAI, 2007.

[271] Håkan L. S. Younes, Marta Z. Kwiatkowska, Gethin Norman, and David Parker. Numerical vs. Statistical Probabilistic Model Checking. *International Journal on Software Tools for Technology Transfer*, 8(3):216–228, 2006.

[272] Håkan L. S. Younes, Michael L. Littman, David Weissman, and John Asmuth. The First Probabilistic Track of the International Planning Competition. *Journal of Artificial Intelligence Research*, 24:851–887, 2005.

[273] Håkan L. S. Younes and Reid G. Simmons. Probabilistic Verification of Discrete Event Systems Using Acceptance Sampling. 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 223–235. Springer, 2002.

[274] Tom Zahavy, Nir Ben-Zrihem, and Shie Mannor. Graying the black box: Understanding DQNs. In Maria-Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, volume 48 of *JMLR Workshop and Conference Proceedings*, pages 1899–1908. JMLR.org, 2016.

[275] Paolo Zuliani, André Platzer, and Edmund M. Clarke. Bayesian Statistical Model Checking with Application to Stateflow/Simulink Verification. *Formal Methods in System Design*, 43(2):338–367, 2013.

A. Appendix

A.1. MODEST MDP Model of Example 1, 2, and 3

This is the MDP model used in Example 1, 2, and 3 in the MODEST language. The results for the properties given in comments are obtained with the MCSTA explicit probabilistic model checking engine of the MODEST TOOLSET.

```
1  // Test with dead-end
2  action a, b, c, d;
3
4  int n = 0;
5  int rew = 0;
6
7  process test()
8  {
9    alt{
10     ::when(n==0) c palt{
11       :0.6: {= rew = 1, n=2 =}; test()
12       :0.4: {= rew = 1, n=1 =}; test()
13     }
14     ::when(n==0) a {= rew = 0, n=1 =}; test()
15     ::when(n==1) alt{
16           ::b {= rew = 2, n=2 =}; test()
17           ::d {= rew = 0, n=3 =}; test()
18     }
19   }
20 }
21
22 test()
23
24 property checkemax = xmax(s(rew), n==2);          // infinity
25 property checkemin = xmin(s(rew), n==2);          // 1.8
26 property checkpmax = pmax(<> (n==2));             // 1
27 property checkpmin = pmin(<> (n==2));             // 0
28
29
30
```

```
31 // Test without dead-end
32
33 action a, b, c;
34
35 int n = 0;
36 int rew = 0;
37
38 process Test()
39 {
40   alt{
41     ::when(n==0) c palt{
42       :0.6: {= rew = 1, n=2 =}; Test()
43       :0.4: {= rew = 1, n=1 =}; Test()
44     }
45     ::when(n==0) a {= rew = 0, n=1 =}; Test()
46     ::when(n==1) b {= rew = 2, n=2 =}; Test()
47   }
48 }
49
50 Test()
51
52 property checkEmax = Xmax(S(rew), n==2);              // 2
53 property checkEmin = Xmin(S(rew), n==2);              // 1.8
54 property checkPmax = Pmax(<> (n==2));                 // 1
55 property checkPmin = Pmin(<> (n==2));                 // 1
```

A.2. Modysh: Proof for *MinProb*

As announced in Section 4.1.1, this appendix provides a proof that GLRTDP solves *MinProb* properties on MDPs with positive and zero-valued rewards correctly by converging to the optimal fixpoint.
To show convergence to the optimal value function from below in case of an admissible initialization, we can argue along the invariant

$$\forall k, \pi : V_k(s) \leq \mathcal{P}_s^{\pi}(\Diamond G), \text{ where } \pi \text{ s.t. } \mathcal{P}_s^{\pi}(\Diamond G) = V^*(s)$$

stating that the value function for every state s in every iteration k is always at most the value under the optimal policy π starting in s. This means that an initially admissible value function always stays admissible. This is true for the admissible initialization when $k = 0$, because then $V_0(s) = 1$ if $s \in G$ and 0 otherwise. For all other iterations it holds that $V_{k+1}(s) \coloneqq \sum_{s'} \mathcal{P}(s, a, s') \cdot V_k(s')$ for some action a and we can derive that

$$\sum_{s'} \mathcal{P}(s, a, s') \cdot V_k(s') \leq \sum_{s'} \mathcal{P}(s, a, s') \cdot \min_{\pi} \mathcal{P}_{s'}^{\pi}(\Diamond G)$$
$$\leq \sum_{s'} \min_{\pi} (\mathcal{P}(s, a, s') \cdot \mathcal{P}_{s'}^{\pi}(\Diamond G)) \leq \min_{\pi} \sum_{s'} \mathcal{P}(s, a, s') \cdot \mathcal{P}_{s'}^{\pi}(\Diamond G).$$

The second inequality holds because π_{opt} is memory-less and independent of s'.
Now assume π_{opt} is such that $\mathcal{P}_s^{\pi_{opt}}(\Diamond G)$ is minimal for all s. Then for action $a = greedy(s, V_k)$ we have for any action b, and in particular for $b = \pi_{opt}(s)$,

$$\sum_{s'} \mathcal{P}(s, a, s') \cdot V_k(s') \leq \sum_{s'} \mathcal{P}(s, b, s') \cdot V_k(s').$$

Moreover $V_k(s') \leq \mathcal{P}_{s'}^{\pi_{opt}}(\Diamond G)$, which allows us to derive

$$\sum_{s'} \mathcal{P}(s, a, s') \cdot V_k(s') \leq \sum_{s'} \mathcal{P}(s, \pi_{opt}(s), s') \cdot \mathcal{P}_{s'}^{\pi_{opt}}(\Diamond G) = \mathcal{P}_s^{\pi_{opt}}(\Diamond G).$$

Claim: If V_k is a fixpoint for $k \to \infty$ then $\mathcal{P}^{\pi}(\Diamond G) = V_{\infty}(s_0)$ $\forall \pi$ greedy in V. (A.1)
Since $V^*(s) \coloneqq \min_{\pi} \mathcal{P}_s^{\pi}(\Diamond G)$ this means $V^*(s_0) \leq V_{\infty}(s_0)$, and with the result from above ($\forall k : V_k \leq V^*$) we can conclude $V^*(s_0) = V_{\infty}(s_0)$.
It remains to show that (A.1) holds:
Let $\pi_k \coloneqq greedy(V_k)$, i.e., a greedy policy with respect to the value function V_k, and $S_k = \{s \mid \mathcal{P}_{s_0}^{\pi_k}(\Diamond s) > 0\}$, i.e., all states reachable with this greedy policy, then $\max(residual(S_k)) \leq \delta_k$ and for $k \to \infty$ it holds that $\delta_k \to 0$. (Where $\max residual(S_k)$ is the maximal difference between a state's new and old value in iteration k over the state set S_k.)

To show that δ_k will approach 0 it is enough to argue about the states which will be updated an infinite number of times, i.e., in the end, about the states on optimal policies. These are the states in $S_\infty = \bigcap_{i \geq 0} \bigcup_{k \geq i} S_k$.
Let K be such that $\forall k \geq K : \bigcup_{i \geq k} S_i = S_\infty$, i.e., a step from which on we only consider states which will be infinitely often visited when running GLRTDP infinitely long. Assume we are in a step $j + 1 \geq K$. Let $s \in S_\infty$. We have to distinguish two cases:

- If s has not been updated then $V_{j+1}(s) = V_j(s)$.
- If s is the updated state then $V_{j+1}(s) = \min_\alpha \sum_{s'} \mathcal{P}(s, \alpha, s') \cdot V_j(s')$

But this is the same as for simple synchronous value iteration, for which convergence against the optimal fixpoint is proven. For our asynchronous case in GLRTDP we nevertheless have to guarantee fairness among the states in S_∞, i.e., we have to make sure that they are updated infinitely often. This is the case because each possible trial of S_∞ (there are finitely many trials) appears infinitely often, i.e., the states in this trial are updated infinitely often (by construction of GLRTDP when choosing the next greedy action). All other states not in S_∞ can be ignored because they will not influence the greedy policy and optimal values because they are already too large:

For any $s \in S \setminus S_\infty$ it holds that $V_\infty(s) = V_K(s) \leq V^*(s)$, and for any $s \in S_\infty$ by definition of S_∞ and K we know that an action leading again to a state in S_∞ will be chosen, i.e., an $a \in \pi_\infty$: $V_\infty(s) \leq \sum_{s' \in S_\infty} \mathcal{P}(s, \pi_\infty, s') \cdot V_\infty(s')$ but for every action we choose the greedy one and for any $k \geq K$ it holds that $V_k(s) \leq \sum_{s' \in S} \mathcal{P}(s, a, s') \cdot V_k(s') \leq V_\infty(s)$, i.e., the action in π_∞ must have been the greedy action not leading to $S \setminus S_\infty$. This means that V_∞ defines an optimal strategy on S_∞ for $s_0 \in S_\infty$ which is also an optimal strategy on S because no state $s' \in S \setminus S_\infty$ is visited even with $V_\infty(s') < V^*(s')$. In addition, the initial state lies in S_∞ by construction, i.e., $\mathcal{P}_{\min}(\Diamond G) = V^{\pi_{opt}}(s_0)$ reaches the fixpoint and is updated infinitely often.

In summary, when running GLRTDP in an infinite number of iterations, the value function for states in S_∞ will approach the optimal values of the minimal probability to reach the goal from below, will never get larger than the optimal value and the difference between V and V^* always becomes strictly smaller for these states. In addition, we can at some point stop updating the value function for parts of the state space because these values will not have an influence on the correct optimal result for the initial state. In our implementation GLRTDP is designed in such a way that it stops when the values on the optimal policy only change by less than ε, which is the same convergence criterion as for simple value iteration.

A.3. MODYSH: Proof for *MaxProb*

Taking up our promise from Section 4.1.1, we sketch a more formal proof answering why the presented combination of GLRTDP and FRET solves *MaxProb* properties on general MDP structures correctly, not only on problems having at least one almost-sure policy as proven in [180], by converging to the optimal fixpoint.

To show convergence to the optimal value function from above in case of an admissible initialization, we can argue along the invariant

$$\forall k, \pi : V_k(s) \geq \mathcal{P}_s^{\pi}(\Diamond G), \text{ where } \pi \text{ s.t. } \mathcal{P}_s^{\pi}(\Diamond G) = V^*(s)$$

stating that the value function for every state s in every iteration k is always greater or equal than the optimal value under the optimal policy π starting in s. This means that an initially admissible value function always stays admissible. This is true for the admissible initialization when $k = 0$, because then $V_0(s) = 0$ if $s \in \mathcal{S}_{\perp}$ and 1 otherwise. For all other iterations it holds that

$$V_{k+1}(s) \coloneqq \sum_{s'} \mathcal{P}(s, a, s') \cdot V_k(s')$$

for some action a and we can derive that

$$\sum_{s'} \mathcal{P}(s, a, s') \cdot V_k(s') \geq \sum_{s'} \mathcal{P}(s, a, s') \cdot \max_{\pi} \mathcal{P}_{s'}^{\pi}(\Diamond G)$$
$$\geq \sum_{s'} \max_{\pi} (\mathcal{P}(s, a, s') \cdot \mathcal{P}_{s'}^{\pi}(\Diamond G)) \geq \max_{\pi} \sum_{s'} \mathcal{P}(s, a, s') \cdot \mathcal{P}_{s'}^{\pi}(\Diamond G)$$

The second inequality holds because π_{opt} is memory-less and independent of s'.
Now assume π_{opt} is such that $\mathcal{P}_s^{\pi_{opt}}(\Diamond G)$ is maximal for all s. Then for action $a = greedy(s, V_k)$ we have for any action b, and in particular for $b = \pi_{opt}(s)$,

$$\sum_{s'} \mathcal{P}(s, a, s') \cdot V_k(s') \geq \sum_{s'} \mathcal{P}(s, b, s') \cdot V_k(s').$$

Moreover $V_k(s') \geq \mathcal{P}_{s'}^{\pi_{opt}}(\Diamond G)$ and hence

$$\sum_{s'} \mathcal{P}(s, a, s') \cdot V_k(s') \geq \sum_{s'} \mathcal{P}(s, \pi_{opt}(s), s') \cdot \mathcal{P}_{s'}^{\pi_{opt}}(\Diamond G) = \mathcal{P}_s^{\pi_{opt}}(\Diamond G).$$

Claim: If V_k is a fixpoint for $k \to \infty$ then $\mathcal{P}^{\pi}(\Diamond G) = V_{\infty}(s_0)$ $\forall \pi$ greedy in V. (A.2)
Since $V^*(s) \coloneqq \max_{\pi} \mathcal{P}_s^{\pi}(\Diamond G)$ this means $V^*(s_0) \geq V_{\infty}(s_0)$ and with the result from above ($\forall k : V_k \geq V^*$) we can conclude $V^*(s_0) = V_{\infty}(s_0)$.

It remains to show that (A.2) holds:
Let $\pi_k := greedy(V_k)$, i.e., a greedy policy with respect to the value function V_k and $S_k = \{s \mid \mathcal{P}_{s_0}^{\pi_k}(\Diamond s) > 0\}$, i.e., all states reachable with this greedy policy, then $\max(residual(S_k)) \leq \delta_k$ and for $k \to \infty$ it holds that $\delta_k \to 0$. (Where $\max residual(S_k)$ is the maximal difference between a state's new and old value in iteration k over the state set S_k)
To show that δ_k will approach 0 it is enough to argue about the states which will be updated an infinite number of times, i.e., in the end, about the states on optimal policies. These are the states in $S_\infty = \bigcap_{i \geq 0} \bigcup_{k \geq i} S_k$.
Let K be such that $\forall k \geq K : \bigcup_{i \geq k} S_i = S_\infty$, i.e., a step from which on we only consider states which will be infinitely often visited when running FRET-LRTDP infinitely long. Assume we are in a step $j + 1 \geq K$. Let $s \in S_\infty$. We have to distinguish two cases:

- If s has not been updated then $V_{j+1}(s) = V_j(s)$.
- If s is the updated state then $V_{j+1}(s) = \max_\alpha \sum_{s'} \mathcal{P}(s, \alpha, s') \cdot V_j(s')$

This is the same as for simple synchronous value iteration, for which convergence against the optimal fixpoint is proven. For our asynchronous case in GLRTDP we are left with the duty to guarantee fairness among the states in S_∞, i.e., we have to make sure that they are updated infinitely often. This is the case because each possible trial of S_∞ (there are finitely many trials) appears infinitely often, i.e., the states in this trial are updated infinitely often (by construction of GLRTDP when choosing the next greedy action). All other states not in S_∞ can be ignored because they will not influence the greedy policy and optimal values because they are already too large:

For any $s \in S \setminus S_\infty$ it holds that $V_\infty(s) = V_K(s) \geq V^*(s)$ and for any $s \in S_\infty$ by definition of S_∞ and K we know that an action leading again to a state in S_∞ will be chosen, i.e., an $a \in \pi_\infty$: $V_\infty(s) \geq \sum_{s' \in S_\infty} \mathcal{P}(s, \pi_\infty, s') \cdot V_\infty(s')$ but for every action we choose the greedy one and for any $k \geq K$ it holds that $V_k(s) \geq \sum_{s' \in S} \mathcal{P}(s, a, s') \cdot V_k(s') \geq V_\infty(s)$, i.e., the action in π_∞ must have been the greedy action not leading to $S \setminus S_\infty$.

This means that V_∞ defines an optimal strategy on S_∞ for $s_0 \in S_\infty$ which is also an optimal strategy on S because no state $s' \in S \setminus S_\infty$ is visited even with $V_\infty(s') > V^*(s')$.

In addition, the initial state lies in S_∞ by construction, i.e., $P_{\max}(\Diamond G) = V^{\pi_{opt}}(s_0)$ reaches the fixpoint and is updated infinitely often.

To sum up, this shows that when running FRET-LRTDP over an infinite number of iterations, the value function for states in S_∞ will approach the optimal values of the maximal probability to reach the goal from above, will never get smaller than the optimal value and the difference between V and V^* always becomes smaller for these states. In addition, we can at some point stop updating the value function for parts of the state space because these values will not have an influence on the correct optimal result for the initial state. In our

implementation FRET-LRTDP is designed in such a way that it stops when the values on the optimal policy only change by less than ε, which is the same convergence criterion as for simple value iteration.

A.4. Details of the Racetrack Implementation

For reference, this compendium provides details regarding Racetrack and our JANI models thereof appearing in Chapter 5. The models, along with all other infrastructure to generate Racetrack benchmarks from track shapes is publicly available in this Zenodo archive [172][1] containing the whole DSMC infrastructure.

States of the vehicle are described by two vectors: its current position (x, y) indexing a cell within the grid, and its current velocity $(d_x, d_y) \in \mathbb{Z}^2$ in x and y-direction. The state of the vehicle is updated at discrete steps. At each step, the speed of the vehicle can be controlled via 9 different actions corresponding to the acceleration vectors $(a_x, a_y) \in \{-1, 0, 1\}^2$. Acceleration is applied additively, i.e., the vehicle's new velocity vector (d'_x, d'_y) after applying acceleration (a_x, a_y) is given by $d'_x = d_x + a_x$ and $d'_y = d_y + a_y$. The position of the vehicle is updated according to the updated velocity vector, i.e., $x' = x + d'_x$ and $y' = y + d'_y$.

What we just specified is the deterministic variant of Racetrack. In the noisy variant, acceleration only succeeds with a probability of $p \in [0, 1)$, while with probability $(1 - p)$ the vehicle's velocity remains the same.

We say that the vehicle has *crashed* if the vehicle either moved out of the grid (i.e., its position no longer constitutes a valid grid coordinate), or the vehicle's last movement trajectory crossed a wall cell. Determining whether the vehicle has crashed is done by discretizing the trajectory from the vehicle's former position $(x_0, y_0) := (x, y)$ to its new position $(x_n, y_n) := (x', y')$ into a sequence of coordinates $T = \langle (x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n) \rangle$. Then, the vehicle has touched a wall if and only if T references a coordinate of a wall cell. Checking whether the vehicle traversed a goal cell is done in the same fashion. The trajectory discretization T is defined as follows:

$$
T = \begin{cases}
\langle (x, y) \rangle & \text{if } d_x = 0 \text{ and } d_y = 0 \quad (1) \\
\langle (x, y), (x + \sigma_x, y), (x + 2 \cdot \sigma_x, y) \ldots, (x', y') \rangle & \text{if } d_x \neq 0 \text{ and } d_y = 0 \quad (2) \\
\langle (x, y), (x, y + \sigma_y), (x, y + 2 \cdot \sigma_y) \ldots, (x', y') \rangle & \text{if } d_x = 0 \text{ and } d_y \neq 0 \quad (3) \\
\langle (x, y), (x + \sigma_x, \lfloor y + m_y \rceil), (x + 2 \cdot \sigma_x, \lfloor y + 2 \cdot m_y \rceil) \ldots, (x', y') \rangle & \text{if } d_x \neq 0 \text{ and } d_y \neq 0 \text{ and } |d_x| \geq |d_y| \quad (4) \\
\langle (x, y), (\lfloor x + m_x \rceil, y + \sigma_y), (\lfloor x + 2 \cdot m_x \rceil, y + 2 \cdot \sigma_y) \ldots, (x', y') \rangle & \text{if } d_x \neq 0 \text{ and } d_y \neq 0 \text{ and } |d_x| < |d_y| \quad (5)
\end{cases}
$$

where $\sigma_x = sgn(d_x)$, $\sigma_y = sgn(d_y)$ (sgn is the signum function returning the sing) and $m_x = \frac{d_x}{|d_y|}$, $m_y = \frac{d_y}{|d_x|}$. In words, if either the horizontal or vertical speed is 0 (cases 1 to 3), the trajectory contains exactly all grid coordinates on the straight line between (x, y)

[1] http://doi.org/10.5281/zenodo.6362696

and (x', y'). Otherwise, we linearly interpolate n points between the two positions and then for each such point round to the closest position on the map. In our model n is given by $\max\left(|d_x|, |d_y|\right)$, while the original discretization model [27, 45] always chooses $n = d_x$. The latter is problematic when having a velocity which moves more into the y (case 5) than into the x direction (case 4), as then only few points will be contained in the trajectory and counterintuitive results are produced.

Our JANI implementation is a straightforward encoding of the model described above. The track is encoded as a (constant) two-dimensional array whose size equals that of the grid. The JANI files of different Racetrack instances differ only in this array.

Vehicle movements and collision checks are represented by separate automata that synchronize using shared actions. The vehicle automaton keeps track of the current state of the vehicle via four bounded integer variables (position and directional velocity), and two Boolean variables, indicating whether the vehicle has crashed, or has reached a goal cell. The automaton starts at a location with one edge for each of the 9 different acceleration vectors. Each of the edges updates the velocity accordingly, and sends the start and resulting end coordinates to the collision check automaton. The collision check can respond with three different answers: “valid”, “crash”, or reached “goal”. If the trajectory was valid, the vehicle automaton transitions back to its initial location. Otherwise, the vehicle automaton transitions into a terminal location where no further moves are possible.

The collision check automaton takes care of two things. It first checks whether the vehicle’s destination lies within the grid. If so, it then iteratively computes the discretized trajectory T, and looks up for each referenced coordinate whether the corresponding entry in the grid array represents a wall or a goal cell. If the trajectory leads out of the track, or when an intersection of the trajectory with either a wall or a goal cell is detected, the result is immediately sent to the vehicle automaton. If the trajectory was completely generated without detecting a collision, the vehicle automaton’s request is answered with “valid”, and the location is reset, waiting for the next trajectory to test.

A.5. Supplementary Material

Below, we provide a list summarizing supplementary material published online (co-)authored by the author of this thesis which is related to the contents covered in this thesis.

- All works on research on perspicuous automated decisions which are centered around Racetrack in the Center for Perspicuous Computing (CPEC) are presented online at https://racetrack.perspicuous-computing.science/
- All infrastructure used to conduct QComp 2020 [121] and detailed results are available on the QComp 2020[2] website, and in the Zenodo archive[3] of the competition.
- The artifact of the tool paper on Modysh [173] is available on Zenodo[4] [174] with all infrastructure to reproduce the experiments. The tool itself can also be downloaded with the official version of the Modest Toolset[5].
- An artifact containing all tools, infrastructure, and data used for studies in the context of DSMC is archived at Zenodo[6] [172]. Beside the latest version of the DSMC implementation in modes, it summarizes the content of the following items.
- The models and infrastructure used for Deep Statistical Model Checking and the case studies on it can be found online[7] [108]. The infrastructure for the scalability study is also publicly available online[8].
- An artifact demonstrating MoGym and containing its tool infrastructure can be found here[9] [107]. In addition, a good reference for MoGym is this documentation[10].
- The TraceVis tool together with a demonstration video and the data together with the modes version used in the tool demonstration and evaluation can be found here[11] [104].

[2] https://qcomp.org/competition/2020/
[3] https://doi.org/10.5281/zenodo.3965313
[4] http://doi.org/10.5281/zenodo.4922360
[5] https://www.modestchecker.net/
[6] http://doi.org/10.5281/zenodo.6362696
[7] http://doi.org/10.5281/zenodo.3760098
[8] https://dcloud.cs.uni-saarland.de/s/Qs4DTLXx7FsMaRx
[9] https://doi.org/10.5281/zenodo.5643442
[10] https://momba.dev/gym/
[11] http://doi.org/10.5281/zenodo.3961196