A Learning Twist on Controllers: Synthesis via Partial Exploration and Concise Representations

Pranav Ashok

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktor der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:
Prof. Dr. Helmut Seidl

Prüfende der Dissertation:
1. Prof. Dr. rer. nat. Jan Křetínský
2. Prof. Kim Guldstrand Larsen R,
   Aalborg University, Denmark
3. Prof. Dr.-Ing. Matthias Althoff

Die Dissertation wurde am 28.09.2020 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 10.02.2021 angenommen.

Abstract

Quantitative model checking and controller synthesis deal with the task of synthesizing dependable controllers satisfying certain specifications for probabilistic and hybrid systems. However, commonly used techniques are susceptible to the curse of dimensionality. That is, when the number of state variables increase, the size of the state space grows exponentially. Further, even if this problem is handled and controllers are synthesized successfully, the resulting controllers are typically large and incomprehensible. This thesis investigates techniques to mitigate these two problems.

State-space explosion. We present techniques based on partial exploration for ϵ-optimal controller synthesis of discrete- and continuous-time Markov decision processes. Our approach makes use of simulations to identify a small part of the state space, sufficient to synthesize a controller with the desired guarantees. For Markov decision processes with the reachability objective, we combine the guaranteed simulation-based technique of bounded real-time dynamic programming, and the extremely scalable game solving technique of Monte Carlo tree search. The resulting algorithm benefits from the best of both worlds. For continuous-time Markov decision processes with the time-bounded reachability objective (TBR), we use simulations to obtain under- and over-approximating sub-models. On them, we run existing TBR algorithms to obtain lower and upper bounds on the optimal TBR value. The sub-model is expanded with the help of more simulations until corresponding bounds sufficiently converge, allowing to extract an ϵ-optimal controller.

Large and incomprehensible controllers. We propose the use of decision trees to represent controllers synthesized from probabilistic and hybrid systems. Decision tree learning can exploit internal structures of these controllers to group sets of states with the same control action. We extend standard decision tree learning algorithms with the ability to represent controllers with correctness guarantees without losing these guarantees. Based on this, we develop two tools. First, we extend the state-of-the-art control synthesis tool `Uppaal Stratego` to produce safe, near-optimal, and small controllers for hybrid Markov decision processes. We introduce a novel *safe pruning* technique that allows one to explore the size-optimality trade-off without losing safety guarantees of the original controller. Next, we introduce the `dtControl` toolkit, which can convert any non-randomized memoryless controller available in the form of a lookup table into a decision tree. We also present a novel determinization strategy that can locally determinize parts of a permissive controller in order to obtain significant reductions in the size of the resulting decision tree. We demonstrate our results on controllers obtained from the hybrid systems tool `SCOTS` as well as `Uppaal Stratego`.

Zusammenfassung

Quantitative Modellüberprüfung und Reglersynthese beschäftigen sich mit der Aufgabe, zuverlässige Regler zu synthetisieren, die bestimmte Spezifikationen für probabilistische und hybride Systeme erfüllen. Häufig verwendete Techniken leiden jedoch unter dem sogenannten Fluch der Dimensionalität. Das heißt die Größe des Zustandsraums wächst exponentiell mit der Anzahl der Zustandsvariablen.

Außerdem, selbst wenn, trotz dieses Problems, Regler erfolgreich synthetisiert werden, sind die resultierenden Regler normalerweise groß und unverständlich. In dieser Arbeit werden Techniken untersucht, die diese beiden Probleme mildern.

Zustandsraumexplosion. Wir stellen Techniken vor, die auf einer partiellen Erkundung für eine ϵ-optimale Reglersynthese von zeitdiskreten und zeitkontinuierlichen Markov-Entscheidungsprozessen basieren. Unser Ansatz nutzt Simulationen, um einen kleinen Teil des Zustandsraums zu identifizieren, der ausreicht, um einen Regler mit den gewünschten Garantien zu synthetisieren. Für Markov-Entscheidungsprozesse mit dem Ziel der Erreichbarkeit kombinieren wir die garantierte simulationsbasierte Technik der beschränkten dynamischen Echtzeitprogrammierung und die extrem gut skalierbare Spiele-Lösungstechnik der Monte-Carlo-Baumsuche. Der resultierende Algorithmus profitiert vom Besten aus beiden Welten. Für zeitkontinuierliche Markov-Entscheidungsprozesse mit dem zeitgebundenen Erreichbarkeitsziel (TBR, aus dem Englischen "time-bounded reachability") verwenden wir Simulationen, um unter- und überapproximierende Teilmodelle zu erhalten. Auf diesen lassen wir existierende TBR-Algorithmen laufen, um untere und obere Grenzen für den optimalen TBR-Wert zu erhalten. Das Teilmodell wird mit Hilfe weiterer Simulationen erweitert, bis die entsprechenden Grenzen ausreichend konvergieren, so dass ein ϵ-optimaler Regler extrahiert werden kann.

Kleine und erklärbare Controller. Wir schlagen vor, Entscheidungsbäume zu nutzen, um Regler darzustellen, die aus probabilistischen und hybriden Systemen synthetisiert wurden. Das Lernen von Entscheidungsbäumen kann interne Strukturen dieser Regler ausnutzen, um Gruppen von Zuständen mit derselben Kontrollaktion zu vereinen. Wir erweitern die Standardalgorithmen zum Lernen von Entscheidungsbäumen um die Fähigkeit, Regler mit Korrekheitsgarantien darzustellen, ohne dabei diese Garantien zu verlieren. Auf dieser Grundlage entwickeln wir zwei Programme. Erstens erweitern wir das dem Stand der Technik entsprechende Regelungssynthese-Programm `Uppaal Stratego` um sichere, nahezu optimale und kleine Regler für hybride Markov-Entscheidungsprozesse bereitzustellen. Wir führen eine neue Technik namens *sicheres Stutzen* ein, die es erlaubt, das Spannungsfeld zwischen Größe und Optimalität zu erforschen, ohne die Korrektheitsgarantie des ursprünglichen Reglers zu verlieren. Als nächstes stellen wir das Programm `dtControl` vor, das jeden nicht randomisierten, gedächtnislosen Controller, der in Form einer Nachschlagetabelle verfügbar ist, in einen Entscheidungsbaum umwandeln kann. Außerdem stellen wir eine neuartige Bestimmungsstrategie vor, mit der Teile eines nicht deterministischen Reglers lokal bestimmt werden können, um die Größe des resultierenden Entscheidungsbaums deutlich zu reduzieren. Wir demonstrieren unsere Ergebnisse an Reglern, die mit dem Programm für hybride Systeme `SCOTS` sowie `Uppaal Stratego` gewonnen wurden.

Acknowledgements

First and foremost, I would like to thank my doctoral advisor, Jan, for all the support and guidance he has provided me. I am incredibly grateful for the belief and trust he has placed in me that has encouraged me to take up roles of responsibility. It has always been a pleasure to discuss with him, both work and life.

I would like to express my sincere gratitude to Prof. Kim G. Larsen, for hosting me at Aalborg University for two wonderful research stays. I consider his infectious enthusiasm and excitement entirely responsible for driving my work on decision trees. I always look forward to meeting and brainstorming with Kim.

Further, I would like to thank Prof. Tomás Brázdil for inviting me to Brno for my first research stay abroad. It was delightful working with you and Ondrej, not to forget the great lunches!

I am thankful to all my co-authors, Adrien Le Coënt, Christoph H. Lampert, Holger Hermanns, Jakob Haahr Taankvist, Krishnendu Chatterjee, Majid Zamani, Mathias Jackermeier, Przemyslaw Daca, Pushpak Jagtap, Stefanie Mohr, Tobias Winkler, Vahid Hashemi, Viktor Toman, and Yuliya Butkova for the great discussions and fruitful collaborations.

My doctoral studies and my stay in Germany wouldn't have been possible without the generous scholarship provided by the IGSSE Project "10.06 PARSEC" and employment funded by DFG Project "Statistical Unbounded Verification". I would like to thank the whole IGSSE team, especially Marco Barden and Jo-Anna Küster, who have been amazingly supportive during the far too many times I approached them. I will never forget the immense support provided by Agnieszka Slota of TUM Graduate School during the deportation scare I had.

I would like to thank Javier Esparza, our chair, whom I look up to for his patience and ability to calmly deal with challenging situations. Further, I would like to thank all my I-7 colleagues for the fun Mensa lunches and discussions. I cannot sufficiently express my words of gratitude towards Maxi, colleague and friend, for being such a great work partner and going out of his way to help me every time I got myself into trouble. I really enjoyed collaborating with him, both professionally and personally.

I am thankful to Bala, Christoph, Kush, Maxi, Shyamlal, and Tobi for proofreading various parts of this thesis and for their valuable feedback. Special thanks to my sister Cookies for pointing out all those bugs in my use of English, and not to forget, those odd Oxford commas.

My gratitude also goes to two crucial people, Prof. Muralikrishnan and Nandu, without whom I would not have chosen this path. I am grateful to Sri whose QATH course at CMI led me to the beautiful area of quantitative verification. I also thank all my teachers and mentors for making me what I am today.

I cannot forget the role my friends have played in my life when living so far away from home. Thanks to the JGM gang for making Munich a home away from home. Thanks to Anjali, Honey, Jap, Pai, Sachin, Shaju and Vivek for being the cheerleaders they are. Finally, my greatest thanks to my grandparents, Amma, Appa, and Cookies, for being so understanding, encouraging, and empathizing.

Contents

List of Figures

List of Tables

Acronyms

ADD	Algebraic Decision Diagram.
BDD	Binary Decision Diagram.
BMCTS	Bounded Monte Carlo Tree Search (MCTS).
BRTDP	Bounded Real-time Dynamic Programming.
BVI	Bounded Value Iteration.
CPS	Cyber-Physical System.
CTL	Computation Tree Logic.
CTMC	Continuous-time Markov Chain.
CTMDP	Continuous-time Markov Decision Process.
DT	Decision Tree.
LP	Linear Programming.
LTL	Linear Temporal Logic.
MC	Markov chain.
MCTS	Monte Carlo Tree Search.
MDP	Markov Decision Process.
ML	Machine Learning.
RL	Reinforcement Learning.
TBR	Time-bounded Reachability.
UCB1	Upper Confidence Bound.
VI	Value Iteration.

1 Introduction

As long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem.

Edsger W. Dijkstra

With the advent of the digital age, sophisticated systems known as Cyber-Physical System (CPS) containing hardware-software interactions have emerged. Robots, smart grids, intelligent signalling systems, IoT devices, digital communication networks, climate control systems, medical devices such as pacemakers, braking systems, and many more, are prime examples of systems where mechanical/analog components work hand-in-hand with digital components. Even daily-use home appliances such as refrigerators and coffee machines are now controlled by embedded controllers running some 'smart' programs. Malfunction of safety-critical systems, a subclass of CPS, may be specially catastrophic, leading to fatal accidents, loss of capital, or risk thereof. Unfortunately, Dijkstra's grand vision outlined in his 1972 Turing Award lecture [Dij72] has been far from true:

> ... well before the seventies have run to completion, we shall be able to design and implement the kind of systems that are now straining our programming ability, at the expense of only a few percent in man-years of what they cost us now, and that besides that, these systems will be virtually free of bugs.

Bugs in safety-critical systems have caused numerous disasters over the past decades. Concurrency bugs in the controller of the Therac-25 radiation therapy machine caused at least six accidents including three fatalities between 1985 and 1987 where cancer patients were exposed to massive amounts of radiation [LT93]. An integer overflow malfunction in the controller software of the Ariane 5 space launch vehicle caused it to self-destruct 37 seconds into launch, resulting in a loss of $500 million [Baa18]. More recently, a flaw in the Manoeuvring Characteristics Augmentation System (MCAS) of the Boeing 737 MAX caused the crashes of Lion Air Flight 610 [Wik20b] and Ethiopian Airlines Flight 302 [Wik20a] with 189 and 157 fatalities respectively. An intriguing list of 107 software horror stories can be found in [Der].

One may hence argue that it is highly advantageous or even necessary that the controllers governing such safety-critical systems are proved to be correct. Programmers

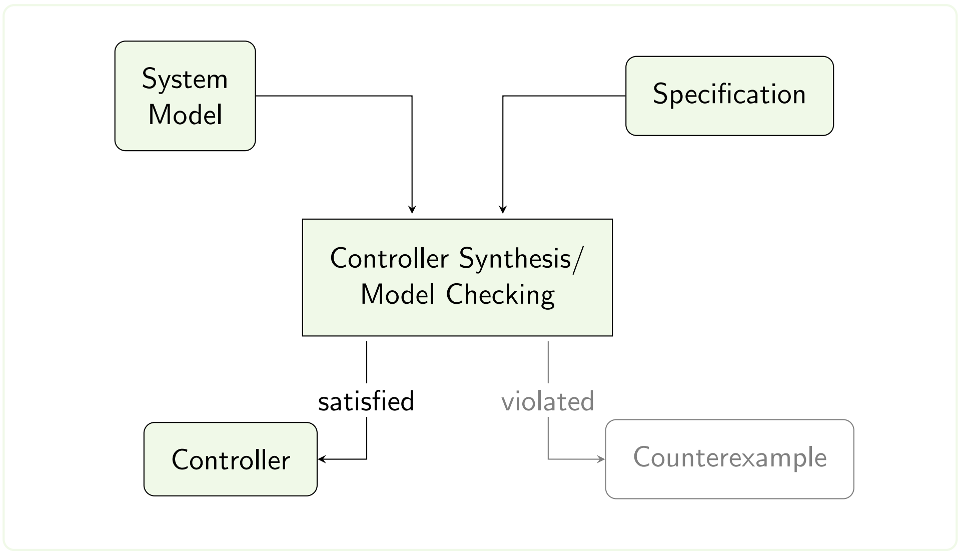


Figure 1.1: A schematic depicting the automatic synthesis of controllers from a system model and specification. In this thesis, we focus on controller synthesis as well as representation of the resulting controller.

in the ideal world painted by Dijkstra would hand-write proofs of correctness. It might be possible to prove by hand that there exists no bugs for simple and small finite state systems. However this task becomes intractable very quickly. Unlike the expectations of 1970s and 80s, people no longer try to prove correctness of programs by hand using Floyd-Hoare [Hoa69] logic. Instead, they turn to automatic methods offered by computer-aided verification.

1.1 Computer-aided Verification

There are two fundamental questions relevant in the study of safety-critical systems. Given a system and a specification:

1. (Verification) Does the system satisfy a specification?
2. (Controller Synthesis) How must the system be controlled in order to satisfy a specification?

Typically, these problems are equivalent and solving one easily gives the solution to the other. In 1970s and 80s, multiple lines of research started to emerge to automate verification of hardware and software. One line of research that focused on model-based verification techniques, i.e., using mathematical models describing the behaviour of the system, came to be known as *model checking*.

Figure 1.1 gives an overview of the model checking process. The system under analysis is first modelled either automatically (for example, from source code) or by hand. Next, the requirements of the system are formalized into logical specifications. The model checker or a controller synthesis tool takes as input the model and specification and computes whether the system satisfies the specification. If it does, typically a witness in the form of a controller may be produced. Informally, the controller is a set of rules describing the actions which the system may take in each state, so that the specification is satisfied. If the specification is violated, then a counterexample may be returned.

Model checking of finite-state discrete systems One of the first works along this line involved the reachability analysis of finite-state protocols modelled using finite-state machines [Boc78; BZ83]. The advent of temporal logics [Pnu77; GPS+80; Pnu81] to specify and verify temporal properties of programs ushered a new era in verification. Subsequently, model checking was developed independently by Clarke and Emerson [CE81] as well as Queille and Sifakis [QS82]. Then, in the mid-80s, came work on automata-theoretic approach to program verification. Vardi and Wolper [VW86] introduced the paradigm that is now ubiquitous in every model checking graduate course. Model checking has been an active area of research ever since. For a comprehensive history of model checking, an interested reader may refer to [BK08, p. 16] and [GV08].

Model checking of quantitative systems Soon, however, there was theoretical as well as practical interest, e.g. from the side of cyber-physical systems, to explore richer quantitative modelling formalisms. Two distinct sub-fields emerged, namely, probabilistic model checking and hybrid system control. As one may observe, most physical processes are stochastic in nature, displaying uncertainty and randomness, and hence it was important to use modelling formalisms expressive enough to model uncertainty. In his seminal paper [Var85], Vardi used an automata theoretic approach to check Linear Temporal Logic (LTL) specifications on Markov chain (MC) and proposed concurrent Markov chains (MCs) as a better modelling formalism for probabilistic systems. Concurrent MCs turned out to be a guise [Var99] of Markov Decision Processes (MDPs) [How60; Put94], a widely studied modelling formalism in various fields such as reinforcement learning and operations research. A more detailed historical account on the development of probabilistic model checking may be found in [BK08, p. 896].

Alternately, verification and control of CPS, displaying a unique combination of digital and analog components, necessitated a cooperation between the fields of formal verification and control theory. This led to the study of hybrid systems [Tab09], infinite-state systems displaying a mixture of discrete and continuous transitions defined by differential equations. The concept of bisimulation, introduced by [Par81] and [Mil89], which is helpful in showing equivalence between systems, has played a key role in verification of hybrid systems. The foundations for the analysis of hybrid systems were laid in the seminal paper of Alur and Dill [AD90], where they introduced timed automata and showed that a bisimulation relation between the infinite-state automata and a finite symbolic model could be established. Since then, many techniques have emerged for timed [LPY95; Yov96; DT98] as well as hybrid system verification [Tab09; BYG17]. [AHL+00] show that various classes of hybrid systems can be abstracted to purely discrete systems, following which standard model checking can be applied. More recently, work on approximate bisimulations [GP07] have brought more classes of hybrid systems under the purview of formal verification. See [Tab09; BYG17] for more information on Hybrid systems, suitable for both formal methods and control theory practitioners.

1.2 Challenges

In this dissertation, we are primarily concerned with two challenges faced by the field of quantitative model checking and controller synthesis.

State-space explosion Model checking, be it on discrete, stochastic, or hybrid systems, comes with the inherent challenge of state-space explosion [CKN+12], which refers to the size of the state space growing exponentially when the number of state variables increase.

Large and incomprehensible automatically synthesized controllers Even if we successfully tame the state-space explosion problem and synthesize controllers automatically, their large size and lack of interpretability makes them disadvantageous in multiple ways. Firstly, a controller presented as a huge list of state-action pairs in the form of lookup tables might not fit on the memory of embedded devices controlling cyber-physical systems [ZVJ18]. Secondly, since synthesis tools do not optimize the controller for interpretability, a relatively simple control sequence might be represented in an overly complicated way, and the controller would have to be treated as a black-box. This is potentially disadvantageous as it makes it impossible for domain experts to catch potential issues in the controller, caused possibly by bugs in the model or the controller synthesis tool.

1.3 Solutions to the Challenges

In this thesis, we make a humble attempt at solving these two challenges with learning-based techniques.

1.3.1 Address state-space explosion with partial exploration

While the classical algorithms were developed early on (see [BK08]), a lot of research in model checking algorithms may be attributed to the state-space explosion problem [CKN+12]. For non-quantitative systems, various techniques such as symbolic model checking [BCM+90], bounded model checking [BCC+99], partial-order reduction [Val89; God90; Pel93], and counterexample guided abstraction-refinement (CEGAR) [CGJ+00] have emerged.

In the probabilistic world, similar advances have been made to tackle the state-space explosion problem [BHK19]. Symbolic techniques making use of efficient data structures like Binary Decision Diagram (BDD) are typically heavily used in probabilistic model checking tools [KNP11; HJK+20]. There are minimization techniques based on probabilistic bisimulation [LS91] adapted to probabilistic systems in [SL95]. Various abstraction techniques, where even non-bisimilar states are grouped together, have also been developed [DJJ+02; DN04; HHW+10; KKN+10]. See [DGV+12] for a comprehensive survey of abstraction based techniques.

A dual approach to abstraction is to consider only restricted parts of the state space, which we refer to as *partial exploration*. This idea has been explored in reinforcement learning since 1990s as a means to tackle models with huge state spaces [BBS95; Ber05, Ch. 6]. However, they do not come with strong guarantees/error bounds on the values they compute, but are best-effort algorithms. Consequently, until recently, these techniques were not directly usable for the analysis of safety-critical systems.

Guaranteed reachability model checking using partial exploration One of the first specifications used in model-based verification that continues to hold relevance today is reachability, considered as early as in the late 1970s [Boc78]. In the context of Markov Decision Process (MDP) verification, the reachability specification asks if the probability of reaching some set of goal states maximized over all possible choices of actions is greater than a threshold. If the specification holds, a witness in the form of a policy or a controller is obtained that determines the action to be chosen in each state so that the chances of reaching the target is maximized. The first partial exploration based algorithm came from Brazdil et al. [BCC+14], who adapted the simulation-based algorithm of [MLG05] to give convergence results with error bounds, i.e., the probability of satisfying the specification is computed with an error of at most ϵ. This technique used asynchronous value iteration, novel in the context of probabilistic verification, to come out with a guaranteed reachability probability value. This approach turned out to be a proof-of-concept and has led to many other works based on the partial exploration idea [ACD+17; KKK+18; KM19; AKW19].

Contribution

Our contribution towards solving the state-space explosion problem is two-fold:

- For MDP, we introduce a novel asynchronous bounded value iteration algorithm directed by Monte Carlo Tree Search (MCTS) [Cou06] for computing the reachability value up to ϵ-precision. We propose three different variants interleaving MCTS with Bounded Real-time Dynamic Programming (BRTDP) [BCC+14] to various degrees. These algorithms learn to explore and analyse the model partially, sufficient to answer the reachability problem with ϵ-guarantees.
- For the continuous-time extension of MDP, the Continuous-time Markov Decision Process (CTMDP), we introduce an algorithm that speeds up computing the (ϵ-precise) time-bounded reachability value based on partial exploration. The idea, like in MDP, is to find a core subsystem of the original system and only analyse this subsystem. Our technique shows potential to speed up existing time-bounded reachability algorithms for CTMDPs.

1.3.2 Taming automatically synthesized controllers with better representations

There are not many works dealing with controller representation in the verification and cyber-physical systems literature. Traditionally, controllers have been represented as explicit state-action maps or lookup tables which are extremely large. Some control synthesis tools, such as SCOTS [RZ16] or PESSOA [JDT10], use BDDs [Bry86] to represent controllers. However, the size of the BDD is extremely sensitive to the variable ordering used and finding the optimal variable ordering is NP-complete [BW96]. In [BCC+15], Brazdil et al. introduce an approach where Decision Trees (DTs) [LL14] were used to learn explainable counterexamples resulting from MDP model checking. In order to tame automatically synthesized controllers, we propose the use of decision trees to represent not only ϵ-optimal controllers from probabilistic systems, but also guaranteed optimal controllers arising from control synthesis of non-probabilistic systems such as Hybrid systems.

Our contribution

Here, our contribution is four-fold:

- We present the first results of using DT representations for controllers of CPS and specifically, Hybrid systems, where a state comprises of multiple quantitative variables. The work differs from traditional controller representations in the sense that it produces explainable and meaningful representations, exploiting the inherent structures of these controllers.
- We demonstrate the effectiveness of the representation by integrating DTs into `Uppaal Stratego`. We explore the trade-off between size and optimality of the safe and near-optimal controllers synthesized by `Uppaal Stratego`, all the while preserving the safety guarantees.
- Following up on that, we develop an open-source toolkit `dtControl` for constructing and exporting decision tree representations of controllers. The toolkit is highly extensible, allowing users to develop their own decision tree learning algorithms easily, as well as building pipelines to their own controller synthesis tools.
- Finally, we introduce a novel determinization algorithm called *MaxFreq*, which can locally determinize subsets of control rules in permissive controllers, producing even more concise trees.

1.4 Publication Summary

This is a publication-based dissertation containing 4 papers — two on controller synthesis and two on controller representation. The original publications may be found in the appendix. Part I of the Appendix includes:

A Pranav Ashok, Tomáš Brázdil, Jan Křetínský and Ondřej Slámečka. Monte carlo tree search for verifying reachability in Markov decision processes. ISoLA 2018. [ABK+18]

B Pranav Ashok, Yuliya Butkova, Holger Hermanns, and Jan Křetínský. Continuous-time Markov decisions based on partial exploration. ATVA 2018. [ABH+18]

Part II of the Appendix contains:

C Pranav Ashok, Jan Křetínský, Kim Guldstrand Larsen, Adrien Le Coënt, Jakob Haahr Taankvist, and Maximilian Weininger. SOS: Safe, Optimal and Small strategies for hybrid Markov decision processes. QEST 2019. [AKL+19]

D Pranav Ashok, Mathias Jackermeier, Pushpak Jagtap, Jan Křetínský, Maximilian Weininger, and Majid Zamani. dtControl: Decision Tree Learning Algorithms for Controller Representation. HSCC 2020. [AJJ+20]

Each paper has appeared in peer-reviewed conference proceedings. The papers are prefaced with a summary page containing the full citation of the original publication, a short text summarizing the paper, and a summary of the contributions of the thesis author.

Other co-authored papers of the author

Further, the thesis author has co-authored 5 peer-reviewed conference publications not included in this thesis:

1. Pranav Ashok and Krishnendu Chatterjee and Przemyslaw Daca and Jan Kretínský and Tobias Meggendorfer. Value Iteration for Long-Run Average Reward in Markov Decision Processes. CAV 2017. [ACD+17]

2. Pranav Ashok and Tomás Brázdil and Krishnendu Chatterjee and Jan Kretínský and Christoph H. Lampert and Viktor Toman. Strategy Representation by Decision Trees with Linear Classifiers. QEST 2018. [ABC+19]

3. Pranav Ashok and Jan Kretínský and Maximilian Weininger. PAC Statistical Model Checking for Markov Decision Processes and Stochastic Games. CAV 2019. [AKW19]

4. Pranav Ashok and Krishnendu Chatterjee and Jan Kretínský and Maximilian Weininger and Tobias Winkler. Approximating Values of Generalized-Reachability Stochastic Games. LICS 2020. [ACK+20]

5. Pranav Ashok and Vahid Hashemi and Jan Kretínský and Stefanie Mohr. DeepAbstract: Neural Network Abstraction for Accelerating Verification. ATVA 2020. [AHK+20]

1.5 Outline of the Thesis

This publication-based dissertation is divided into four chapters followed by an appendix. In Chapter 1, we give a brief history to model checking, discuss two important challenges and give a very brief summary of the main contributions. Chapter 2 lays down foundations and preliminaries necessary to understand the content of the thesis. In Chapter 3, we discuss the state-of-the-art in model checking of MDPs and CTMDPs, and summarize how we tackle the state-space explosion problem through partial exploration. This chapter is based on the papers A and B. Chapter 4, based on papers C and D, discusses the problems in traditional controller representations and expounds on the idea of learning decision trees to represent guaranteed controllers. In Chapter 5, we conclude the thesis by summarizing the story so far and identifying some future directions. The Appendix contains four publications, papers A and B on controller synthesis, as well as papers C and D on controller representation, for which the author is the primary contributor.

2 Preliminaries

In this chapter, we familiarize the reader with some background essential to the rest of this dissertation.

2.1 Basic Notation

Let $\mathbb{R}$ be the set of real numbers with $\mathbb{R}_{\geq 0}$ denoting the set of non-negative real numbers. We denote the set of Booleans $\{true, false\}$ using $\mathbb{B}$. For any given set S, we use the notation 2^S to denote the power set (set of all subsets) of S.

Given a sequence ρ of elements from some set S, prefix(ρ) is used to denote the set of all finite prefixes of ρ. Given a set of infinite sequences Q, Q^* denotes the set of finite prefixes defined by $Q^* = \bigcup_{\rho \in Q} \text{prefix}(\rho)$.

Given a totally ordered set S, and two elements $a, b \in S$, we use the notation $[a, b]$ to denote the subset $X = \{s \in S \mid a \leq s \leq b\}$. For any value $v \in \mathbb{R}$, we call $u \in \mathbb{R}$ an ϵ-precise approximation of v if $|v - u| < \epsilon$ for some $\epsilon \in \mathbb{R}$. We say that u is ϵ-optimal if it is an ϵ-precise approximation of a desired optimal value v.

2.2 Basic Probability Theory

We assume a basic understanding of the concepts of a probability measure, probability space, and random variables. The reader may refer to a standard book on probability such as [Ros19] for more details. A *discrete probability distribution* over a finite set S is a mapping $d : S \to [0, 1]$ such that $\sum_{s \in S} d(s) = 1$. $Dist(S)$ denotes the set of all probability distributions on S. For some continuous random variable X with a probability density function $f(x)$, we define the *cumulative distribution* function as $F_X(a) = \mathbb{P}(X \leq a) = \int_{-\infty}^{a} f(x)dx$. An *exponential distribution* is a continuous distribution with the cumulative distribution function given by $F_X(x) := 1 - e^{\lambda x}$ for $x \geq 0$ and $F_X(x) := 0$ otherwise, where $\lambda > 0$ is called the rate parameter. We write $X \sim exp(\lambda)$ to denote that the random variable X is distributed exponentially.

2.3 Notion of Controllers

We define a *system* as a tuple $S = (C, A, \hookrightarrow)$ where C is a set of configurations of the system, A is a set of actions, and $\hookrightarrow$ is some transition relation. For non-probabilistic systems, $\hookrightarrow$ is a relation $\hookrightarrow \subseteq C \times A \times C$ and for probabilistic systems, $\hookrightarrow$ is a distribution function $\hookrightarrow : C \times A \times C \to [0, 1]$ such that for all $(c, a) \in C \times A$, $\sum_{c' \in C} \hookrightarrow (c, a, c')$ is either 0 or 1. A finite path ρ in the system is denoted by $\rho = c_1 \overset{a_1}{\hookrightarrow} c_2 \overset{a_2}{\hookrightarrow} \dots c_n$ where for every $1 \leq i < n$, $c_i \overset{a_i}{\hookrightarrow} c_{i+1}$ is a valid transition, i.e., $(c_i, a_i, c_{i+1}) \in \hookrightarrow$ for non-probabilistic systems and $\hookrightarrow (c_i, a_i, c_{i+1}) > 0$ for probabilistic systems. The set of all finite paths is denoted by Paths^*.

In its most general form, a controller may be defined as follows

Definition 2.3.1 (Controller)**.** A *history-dependent randomized permissive controller* is a function $\pi : \mathit{Paths}^* \to \mathit{Dist}(A)$ that maps finite paths of the system to a distribution over actions.

Controllers are important objects called by different names in different fields of research. The word "strategy" is used frequently in the game-theoretic setting to denote objects that select best actions for each of the players in each controllable state. In control theory and hybrid systems, the term "controller" is preferred for the object that takes in feedback from the system to decide the next control input. In timed systems, a "scheduler" chooses both an action and a time point at which the action is to be performed. In non-timed systems with a single player, e.g. Markov Decision Processes, the term "policy" is used to denote the object that resolves the non-deterministic choice in each state. Throughout this dissertation, we use the terms *controller*, *strategy*, *policy*, and *scheduler* interchangeably to refer to the same mathematical object.

There are multiple types of controllers commonly arising in literature. Some common aspects that are used for characterizing controllers are:

History dependence Whether the decision taken by the controller depends on the current path ($\pi : \mathbf{Paths}^* \to A$), called *history dependent*, or whether it depends only on the current configuration ($\pi : \mathbf{C} \to A$), called *memoryless.*

Randomization Whether the controller randomizes over the available actions ($\pi : C \to \mathbf{Dist}(\mathbf{A})$), called *randomized*, or chooses actions without randomizing ($\pi : C \to \mathbf{A}$), called *pure.*

Permissiveness Whether the controller allows multiple actions ($\pi : C \to \mathbf{2^A}$), called *permissive*, or if it allows only a single action ($\pi : C \to \mathbf{A}$), called *deterministic.*

Time dependence Whether the decision depends on time in addition to the current path/state ($\pi : C \times \mathbb{R}_{\geq \mathbf{0}} \to \mathit{Dist}(A)$).

3 Controller Synthesis through Partial Exploration

While dealing with the infinite state spaces typical of hybrid systems is a hard problem, we can gain a lot of insight by solving simpler formalisms. As is standard in mathematics, we choose to focus on simpler subclasses of the problem and leave the generalization for future work. In this chapter, we look at two formalisms that can be used to model systems that exhibit non-determinism and stochasticity and see how controllers can be synthesized for them using partial exploration.

The first formalism is that of Markov Decision Processes (MDPs) [How60]. MDPs have been around since 1950s and are a popular tool in many fields such as operations research [Put94], reinforcement learning [SB98], and model checking [BHK19]. MDPs are used to model discrete-time systems in which the evolution of the system is partly stochastic and partly controlled by a decision making entity, typically called a policy or a controller. They can also be seen as a game between the system, which tries to achieve a certain objective, and a probabilistic environment that randomizes the outcomes.

The second formalism is that of Continuous-time Markov Decision Processes (CTMDPs) [Sen09; Put94]. CTMDPs allow for modelling systems where the decisions are taken not at discrete decision epochs like in MDPs, but in continuous-time. Typical areas of application include queuing theory [Sen09], power management and scheduling [QQP01], distributed systems [GGL03; HHK00], epidemic and population processes [Lef81] amongst others.

In this chapter, and subsequently in Papers A and B, we consider the problem of verifying the (time-bounded) reachability specification (also known as objective) on the two formalisms. We focus on the state-space explosion challenge and we present two algorithms, one for MDP and one for CTMDP that uses partial exploration via simulations in order to solve the reachability objective. This chapter is divided into two sections. In Section 3.1, we formally introduce MDPs, describe some of the basic reachability algorithms and conclude with our new algorithm based on partial exploration. In Section 3.2, we introduce the CTMDP formalism, give a brief overview of the existing Time-bounded Reachability (TBR) algorithms and conclude with our simulation-based framework that may be used to speed up existing TBR algorithms. All the algorithms described in this chapter produce an ϵ-optimal controller that describes how to govern the system in order to achieve the reachability objective in question.

3.1 Markov Decision Process (MDP)

MDPs are commonly used in modelling systems that exhibit some random behaviour, observe the Markov property, and are controllable.

Example 3.1.1. A robot, Crawl-E, is placed into a grid world as shown in Figure 3.1. The actuators of the robot are damaged causing unintended movements. If the robot tries to move east, it moves east 90% of the time, but ends up moving north 10% of the time. On the other hand if the robot tries to move north, then it succeeds 80% of the time, but moves west 20% of the time. If it encounters a wall in its intended direction of movement, then the robot stays in the same cell and has to try again. The objective of the robot is to avoid the impending explosion in cell $(2, 1)$ and reach the target in cell $(3, 2)$. Design a controller that maximizes the probability of the robot reaching the target.

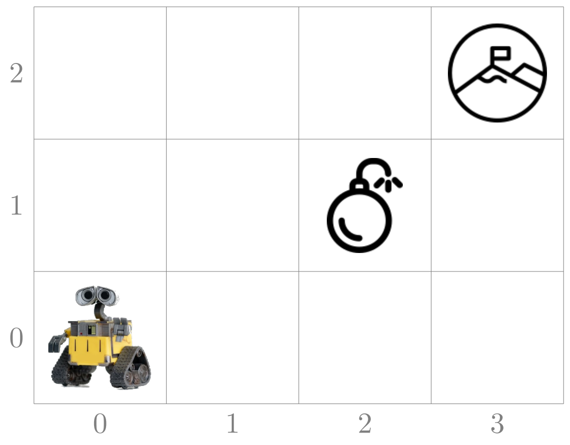


Figure 3.1: A *grid world* with a robot in cell $(0, 0)$, a impending explosion in cell $(2, 1)$, and the target in cell $(3, 2)$.

The situation presented in this fictitious example is an extremely simplified version of a discrete control problem. The reader may be able to quickly come up with a controller or a strategy to reach the target as a quick mental exercise, however, if the grid is larger, or if there are more complex actions or outcomes, the problem quickly becomes difficult. However, this question can easily be solved using automated algorithms by modelling the system as an MDP.

Formally, an MDP can be defined as follows.

Definition 3.1.1 (Markov Decision Process)**.** A Markov Decision Process (MDP) is a tuple $M = (S, s_{init}, A, Av, \Delta, \mathtt{1}, \mathtt{0})$ where

- S is a finite set of states
- s_{init} is the initial state

- A is a finite set of actions
- $Av : S \to 2^A$ assigns to every state a set of actions available in that state
- $\Delta : S \times A \to Dist(S)$ assigns to every state and action, a distribution over successor states
- $\mathtt{1} \subseteq S$ and $\mathtt{0} \subseteq S$ are mutually exclusive sets representing the *target* (also called the *goal*) and the *sink* respectively. From any sink state, it is not possible to transition into a non-sink state.

While various specifications can be checked against an MDP, in this thesis we are concerned only with the reachability specifications. Intuitively, we are interested in finding a set of decisions to be taken in each state that would maximize the probability of reaching the target set $\mathtt{1}$. In order to formalize this notion, we first define controllers.

Definition 3.1.2 (Controller)**.** A controller, also known as a policy in the context of MDPs, is defined as in Definition 2.3.1, instantiating configurations to the set of states, i.e. $C := S$. A policy is hence a map $\pi : (S \times A)^* \times S \to Dist(A)$.

For an MDP M, fixing a controller π and an initial state s yields a unique probability measure $\mathbb{P}^{\pi}_{M,s}$ which, for every set of paths, assigns a value in $[0, 1]$ [BK08, p. 757].

Recall Example 3.1.1. A controller π that can maximize the probability of reaching the target without entering the cell with the impending explosion may be defined as follows:

$$\pi((x, y)) = \begin{cases} \text{“move north”} & \text{if } 0 \leq x \leq 1 \text{ and } 0 \leq y \leq 1 \\ \text{“move east”} & \text{otherwise} \end{cases}$$

where (x, y) gives the coordinate of the cell. Notice that this controller does not depend on the path taken to reach the state from which an action is chosen. Moreover, it does not randomize over the available actions. We shall handle such controllers in the next section, where we discuss model checking algorithms for the reachability objective.

3.1.1 Model checking the reachability objective for MDPs

One of the classic objectives, the reachability objective, is highly studied for all the modelling formalisms we have discussed so far. It was first described and used for the analysis of communication protocols in [Boc78]. In the classical non-probabilistic setting, the objective asks whether it is possible to eventually reach a set of states $\mathtt{1}$ from some initial state s_{init}. This is naturally extended to the purely stochastic world, e.g. discrete-time Markov chains, where the question becomes: is it possible to eventually reach the target set $\mathtt{1}$ from the initial state s_{init} with a probability greater than p?.

In the setting that allows both non-deterministic and stochastic outcomes, the property gets an additional twist. Now, we can no longer ask for the “probability of eventually reaching $\mathtt{1}$ from s_{init}”, since it depends on the controller, i.e., the policy followed by decision maker to choose from the multiple available choices in each state.

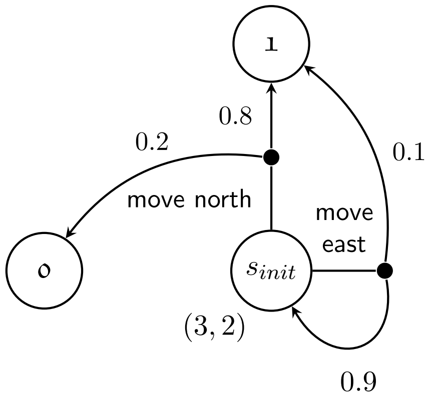


Figure 3.2: Illustrating the reachability objective on systems with both non-determinism and probabilities, using a part of the MDP modelling Example 3.1.1.

For example, consider the MDP shown in Figure 3.2. It models the decisions our robot would have to make in cell $(3, 2)$ of the grid world. If the decision maker chooses the action move north in $(3, 2)$, then the probability of reaching $\mathbb{1}$ is 0.8. However, if the action chosen is move east, then the probability becomes 1, since repeatedly trying this action would eventually take the robot to the target.

Therefore, the question for such systems becomes: "is the maximal (or minimal) probability of eventually reaching $\mathbb{1}$ from s_{init} greater than p?", where the max (or min) operator quantifies over the set of controllers of the model. If a controller is fixed, the system turns purely probabilistic (in this case, a Markov chain) on which the notion of "probability of eventually reaching $\mathbb{1}$ from s_{init}" is meaningful. In Figure 3.2, the maximal probability of reaching the target is therefore 1 and the minimal probability of reaching the target is 0.6.

In order to formally define the reachability objective, we need a few preliminaries. If the set of paths starting in the initial state s_0 and ending in some state of the target set $\mathbb{1}$ is defined as

$$\Diamond\mathbb{1} := \{\rho \mid \rho = s_0 a_0 s_1 a_1 \dots s_n \wedge s_n \in \mathbb{1}\}$$

then, we can define the value of a state s under the controller π to be

$$\mathbb{V}^{\pi}_{s}(\Diamond\mathbb{1}) := \mathbb{P}^{\pi}_{M,s}(\Diamond\mathbb{1})$$

Intuitively, for each state, $\mathbb{V}^{\pi}$ gives the probability of reaching the target from that state following the controller π.

Definition 3.1.3 (Optimal Reachability Value)**.** Given a model M, an initial state s, and a set of target states $\mathbb{1}$, the optimal reachability value is defined as

$$\mathbb{V}^{\text{opt}}_{s}(\Diamond\mathbb{1}) = \operatorname*{opt}_{\pi\in\Pi} \mathbb{V}^{\pi}_{s}(\Diamond\mathbb{1})$$

where opt $\in \{\min, \max\}$ and Π is the set of all controllers of the model.

Now we can define the problem of model checking an MDP against a reachability specification. We call this *quantitative reachability* problem, as opposed to the simpler qualitative reachability problem (whether the optimal reachability value is exactly 0 or 1) that is sometimes considered in literature.

Problem 1 (Quantitative Reachability)**.** Given an MDP M, an initial state s, a set of target states $\mathtt{1}$, and a threshold $p \in [0, 1]$, is the optimal reachability value greater than a threshold p? More precisely, is $\mathbb{V}_s^{\mathrm{opt}}(\Diamond \mathtt{1}) > p$?

A common variant of the reachability objective discussed above is the *step-bounded reachability objective*, which asks "what is the maximal (or minimal) probability of reaching $\mathtt{1}$ from s_{init} within n steps?". While we do not consider this for MDP, for CTMDPs we consider its continuous-time analogue.

The witness to the quantitative reachability problem is a controller that achieves the threshold. Alternatively, we can also phrase the optimal controller problem as follows.

Problem 2 (Optimal Controller)**.** Given an MDP M, initial state s_{init} and set of goal states $\mathtt{1}$, what is the controller π^{opt} that achieves the optimal reachability value? Formally, compute

$$\pi^{\mathrm{opt}} = \arg \operatorname*{opt}_{\pi \in \Pi} \mathbb{V}_s^{\pi}(\Diamond \mathtt{1})$$

Remark 1. An important property of the reachability specification that has spurred the development of a multitude of algorithms is that the optimal value can be achieved by a pure (non-randomized) memoryless deterministic controller [Put94]. These are controllers of the form $\pi : S \to A$.

State of the art

Quantitative reachability on MDPs can be solved in polynomial time using Linear Programming (LP) [dEp63]. However, existing LP algorithms with provable polynomial time performance perform poorly on practical MDPs of non-trivial size [LDK95]. For more information on the complexity results for infinite horizon MDPs, see [PT87; LDK95]. Various other algorithms more practical than LP have been developed and are commonly used in model checking. While traditional algorithms such as Value Iteration (VI) and Policy Iteration [Put94] are still popular, the last decade has shown numerous new algorithms for solving the reachability objective with guarantees. They include Bounded Real-time Dynamic Programming (BRTDP) [BCC+14], interval iteration [HM18; BKL+17; KKK+18], sound value iteration [QK18], and optimistic value iteration [HK20]. These algorithms return an ϵ-optimal result for an arbitrary $\epsilon > 0$.

Now we discuss some basic MDP verification algorithms that are essential to place our contributions in context.

Value iteration

Value Iteration (VI) [Bel57] is a fixpoint iteration method in which a value improvement step is repeatedly applied from a starting guess in order to approximate the optimal reachability value. Chatterjee and Henzinger [CH08] show that the optimal reachability value is not finitely reachable using VI, however it is finitely computable. In other words, it cannot be guaranteed that VI can find the optimal solution in a finite number of steps, however, it is possible to find an ϵ such that an ϵ-precise solution can be obtained in a finite number of steps, which is exponential in the number of states. See [CH08] for a survey of VI techniques for multiple models (deterministic, probabilistic, or game graphs) and objectives, or [BKN+19] for a recent complexity result on VI for finite horizon MDP.

The Bellman equations lie at the heart of VI. For every state s in the target set $\mathtt{1}$, $V^0(s)$ is initialized to 1, and for every other state s, $V^0(s)$ is initialized to 0. The next iteration is performed by applying the following equation

$$V^{n+1}(s) = \begin{cases} 1 & \text{if } s \in \mathtt{1} \\ \max\limits_{a \in A} \sum\limits_{s' \in S} \Delta(s, a, s') V^n(s') & \text{otherwise} \end{cases} \tag{3.1}$$

Intuitively, the “value” of a state s at step i, $V^i(s)$, gives the i-step probability of reaching a goal state from s. We denote the value of the state s in the limit as $V^*(s) = \lim_{n\to\infty} V^n(s)$. VI guarantees that the sequence $(V^i(s))_{i=0}^{\infty}$ converges in the limit, i.e., $V^*(s) = \mathbb{V}_s^{\text{opt}}(\Diamond \mathtt{1})$. Additionally, the iteration initialized as in Equation 3.1 also satisfies the monotone property, i.e., $V^n(s) \leq V^{n+1}(s) \leq \mathbb{V}_s^{\text{opt}}(\Diamond \mathtt{1})$, and is hence referred to as “value iteration from below”. While the advantage of VI is that it is fast in practice at obtaining a good approximation of the reachability value, it is not clear when to stop the iteration. [HM14; BKL+17] show that for certain MDPs, the commonly used stopping criteria, “stop when the difference between two consecutive approximations is less than some small ϵ”, can give arbitrarily wrong approximations.

Bounded value iteration

Several approaches have been proposed in the last decade to address the lack of a practical stopping criterion in VI. A class of these approaches [HM14; BKL+17; KKK+18], which we refer to as Bounded Value Iteration (BVI) algorithms, are based on similar ideas. These algorithms typically terminate giving an interval in which the optimal reachability value is guaranteed to lie.

The basic scheme used by BVI algorithms is to perform value iteration from below as well as above, as given by the following equations. For all states s in the target set $\mathtt{1}$, the lower value is initialized to 1, i.e. $L^0(s) = 1$. For all other states s, the lower bounds are initialized to 0, i.e. $L^0(s) = 0$. For $n > 0$, we have

$$L^{n+1}(s) = \begin{cases} 1 & \text{if } s \in \mathtt{1} \\ \max_{a \in A} \sum_{s' \in S} \Delta(s,a,s') L^n(s') & \text{otherwise} \end{cases} \tag{3.2}$$

For the iteration from above, we initialize $U^0(s) = 0$ for all sink states $s \in \mathtt{0}$, and $U^0(s) = 1$ for all other states. For $n > 0$, we have

$$U^{n+1}(s) = \begin{cases} 0 & \text{if } s \in \mathtt{0} \\ \max_{a \in A} \sum_{s' \in S} \Delta(s,a,s') U^n(s') & \text{otherwise} \end{cases} \tag{3.3}$$

L^n and U^n denote the n-step lower and upper bound functions respectively. While, due to standard VI, we know that the iteration from below converges, this is not necessarily true for the iteration from above. The trouble comes from end-components [De 97, p. 44] present in the MDP. Intuitively, end-components are a set of states and actions for which there exists a policy that can keep the MDP from exiting the component. In order to solve this problem, end-components need special treatment[1], for which an interested reader may refer to the original literature [BCC+14; HM14]. Once the end-components are handled, both the lower and upper bounds converge to the real value. For any $\epsilon > 0$, BVI gives a natural stopping criterion to compute an ϵ-optimal result: stop when for some n and the initial state s, $U^n(s) - L^n(s) \leq \epsilon$. Moreover, BVI guarantees that the optimal reachability value of every state s will be contained between $L^n(s)$ and $U^n(s)$ for all $n \in \mathbb{N}$. Hence, it can be stopped at any time with the optimal reachability value of reaching $\mathtt{1}$ from s lying between $L^k(s)$ and $U^k(s)$, where k denotes the iteration at which BVI was stopped.

Asynchronous bounded value iteration and BRTDP

Now, instead of synchronously updating the values of every state at once in VI, we can choose to do it asynchronously [BBS95]. As long as the algorithm is designed in such a way that the values of every state is infinitely often updated and the end-components are handled properly, the values would converge to the optimal [BCC+14; KKK+18]. It is therefore beneficial to run state updates in a different order, typically guided by simulations, as done by algorithms such as Real-Time Dynamic Programming (RTDP) [BBS95] or even its bounded version Bounded Real-time Dynamic Programming (BRTDP) [MLG05; BCC+14] where a lower and upper bound à la BVI are maintained and updated for each seen state.

While the RTDP algorithm suffers the same issues as VI, such as the lack of a good stopping criterion, a version of BRTDP with guarantees [BCC+14] emerged at the same time as BVI [HM14]. It is important to understand the algorithm of [BCC+14], which

[1]Two recent works, [PTH+20] and [HK20] propose algorithms that are able to provide ϵ-guaranteed results like BVI without having to explicitly handle end-components.

Algorithm 1 Bounded Real-time Dynamic Programming (based on [BCC+14])

```
procedure BRTDP($M, \epsilon$)                              ▷ $M = (S, s_{init}, A, Av, \Delta, \mathtt{1}, \mathtt{0})$
    $\forall s \in S \cdot L = 0, U = 1$
    $\forall s \in \mathtt{1} \cdot L(s) \leftarrow 1$
    $\forall s \in \mathtt{0} \cdot U(s) \leftarrow 0$
    while $U(s_{init}) - L(s_{init}) > \epsilon$ do          ▷ $L$ and $U$ are more than $\epsilon$ apart
        $\rho \leftarrow s_{init}$, $s \leftarrow s_{init}$
        while $s \notin \mathtt{1}$ and $s$ is not part of a maximal end-component do
            $a \leftarrow \arg\max_{a \in Av(s)} U(s)$
            $s' \leftarrow$ SAMPLE($s, a, L, U$)
            $\rho \leftarrow \rho \cdot s$
        end while
        Process maximal end-components
        for each $s \in \rho$ in reverse order do
            for each $a \in Av(s)$ do
                $L(s,a) \leftarrow \sum_{s' \in S} \Delta(s,a,s')L(s')$
                $U(s,a) \leftarrow \sum_{s' \in S} \Delta(s,a,s')U(s')$
            end for
            $L(s) \leftarrow \max_{a \in Av(s)} L(s,a)$
            $U(s) \leftarrow \max_{a \in Av(s)} U(s,a)$
        end for
    end while
    return $L(s_{init})$
end procedure
```

from now on we refer to as BRTDP, since it forms the base of our contribution in Section 3.1.3.

The idea of BRTDP, given in Algorithm 1, is as follows. BRTDP guides state space exploration based on the amount of "knowledge" we have about every state. A path is sampled starting from the initial state, in which the action with the greatest upper bound is chosen (line 8). Next, the successor state is sampled using the `SAMPLE()` (line 9), which may be instantiated in various ways (see [BCC+14, Remark 3]). Once the path either reaches a target state, a maximal end-component, or a sink state, the back-propagation phase starts, in which the Bellman update (Equations 3.2 and 3.3) is applied on each state along the path, to propagate the information that this specific path has gathered. Note that end-components are also processed whenever they are encountered, details of which may be found in [BCC+14, Sec. 4]. After back-propagation, the next path is sampled and the algorithm continues until the difference between the lower and upper bounds in the initial state becomes less than ϵ. This results in BRTDP returning an ϵ-precise approximation of the optimal reachability value.

Algorithm 2 The MCTS scheme (adapted from [ABK+18])

```
1: Create a root $x_0$ of the tree, labelled by $s_{init}$
2: while within computational budget do
3:     $x_{\text{parent}} \leftarrow$ SelectNode($x_0$)                ▷ select a node following the tree policy
4:     $x_{\text{child}} \leftarrow$ ExpandAndPickRolloutNode($x_{\text{parent}}$)
5:     $outcome \leftarrow$ Rollout($x_{\text{child}}$)                ▷ simulate following the rollout policy
6:     BackupOnTree($x_{\text{child}}$, $outcome$)
7: end while
8: return InducedPolicy($t_0$)                ▷ action with the best estimated value
```

3.1.2 Monte Carlo tree search: the preliminaries

In the last section, we discussed algorithms for computing optimal (or ϵ-optimal) reachability in MDPs. While these algorithms give guarantees on their result, they typically don't scale beyond MDPs with more than 10^{10} states. But what can be done when the underlying graphs are so huge that they don't fit in memory? This problem has been considered by various communities in computer science including game theory, artificial intelligence, and formal verification. One of the most successful techniques that has emerged in tackling extremely large games is that of Monte Carlo Tree Search (MCTS), first introduced for the game of Go[2] in [Cou06]. A variant of MCTS was used in the now famous AlphaGo program [SHM+16], which beat the top human Go player Lee Sedol. See [BPW+12] for a detailed survey into the first 5 years of MCTS.

The overarching idea in MCTS is to use random simulations to quickly estimate the quality of different actions available at a given state in the game graph. This is done in a systematic way by building a search tree that helps in identifying the states from which further simulations are to be run. At any point of time, the MCTS tree is a partial unfolding of the source MDP with a few added statistics. In particular, with each tree node (corresponding to a state of the MDP), we store the number of simulations (or rollouts) that have started from the node as well as the cumulative reward obtained by these simulations. MCTS tries to balance exploration, looking into the unknown regions of the state space, and exploitation, looking at those regions of the state space that seem promising. The phases of exploration and exploitation are illustrated in Figure 3.3. In the illustration on the left, many simulations are run from equally unexplored leaf nodes in order to obtain some quick estimates of the rewards available from that state. In this example, the right most leaf node turns out to be most successful. Consequently, in the next round, MCTS picks the right most leaf node for further exploitation, as depicted in the right illustration of Figure 3.3.

Algorithm 2 gives the pseudocode of a basic MCTS algorithm. MCTS proceeds in four stages.

1. In the first stage (line 3), a known policy called the *tree policy* is used to select the next node (corresponding to some state of the MDP) to explore. This policy

[2]The game of Go on a 19x19 board contains around 10^{170} legal configurations! [TF06]

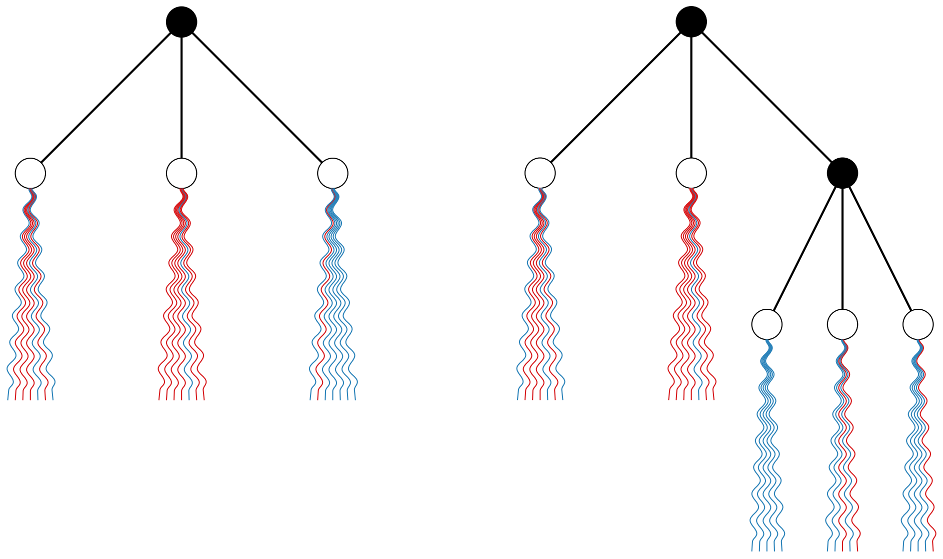


Figure 3.3: Two important phases of MCTS: exploration (left) and exploitation (right). In the exploration phase, simulations are run to estimate the unknown. In the exploitation phase, computational effort is focused on highly rewarding parts of the graph.

tries to pick the best state based on the Upper Confidence Bound (UCB1) metric [ACF02].

2. Once a node is selected, an action is picked and the tree is expanded by adding the successor states corresponding to the action. Following this, one of the newly added nodes is selected to start the simulation (line 4).
3. Next, a *rollout policy* is used to run a simulation (line 5). Typically, the rollout policy uniformly at random picks the next action. The outcome of the rollout is aggregated in x_{child}.
4. Finally, this information is propagated up the MCTS tree (line 6) so that the tree policy is updated.

Balancing exploration and exploitation We now give some technical details of the UCB1 metric that drives the tree policy, which balances exploration and exploitation. For a given node x, if n gives the total number of simulations so far, n_x gives the number of simulations starting from x, and $\frac{r_x}{n_x}$ gives the average reward over these simulations, then UCB1 is given as follows:

$$UCB1(x) = \frac{r_x}{n_x} + \sqrt{\frac{2\ln n}{n_x}}$$

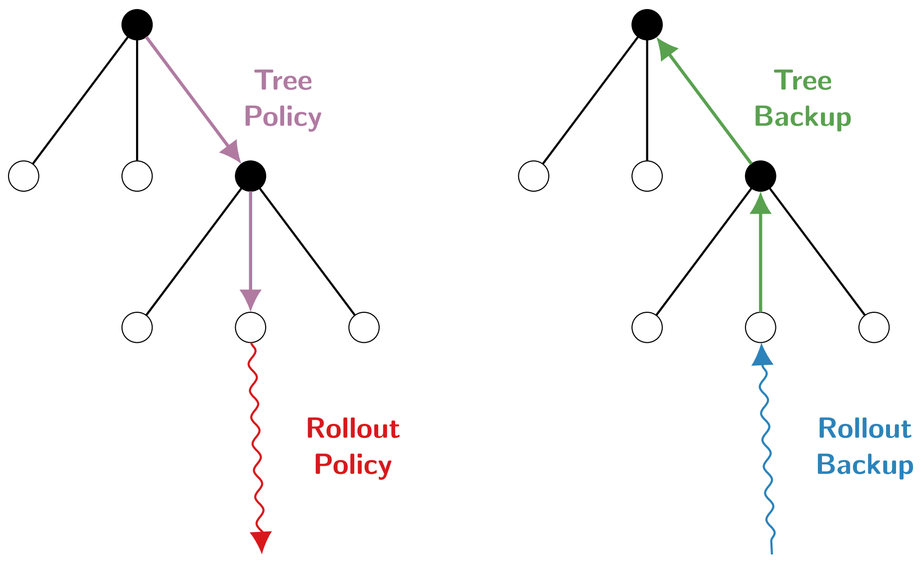


Figure 3.4: Figure illustrating various phases of exploration and update in the broader MCTS framework.

In particular, the tree policy maps every node t to a successor node having the greatest UCB1 value.

$$\textsc{TreePolicy}(x) = \underset{x' \in succ(x)}{\arg\max}\, UCB1(x')$$

The UCB1 metric balances exploration and exploitation. Since the first term of the expression, $\frac{r_x}{n_x}$, gives the average reward, successor nodes that have had a higher proportion of successful simulations would be preferred by the tree policy. On the other hand, the second term is inversely proportional to the square root of the proportion of simulations from the node x. This means that nodes from which fewer simulations have been executed get preference. In particular, nodes that have not had any simulations from them ($n_x = 0$) have $UCB1(x) = \infty$ and are hence picked uniformly at random by the tree policy. An interested reader may refer to the original work introducing UCB1 [ACF02] for further details and analysis.

3.1.3 Contribution: MCTS + BRTDP

The results from BRTDP show us that partial exploration is a viable technique for solving the optimal reachability problem with ϵ-optimality guarantees. However, the success of BRTDP for a particular model depends heavily on there being a small set of states that are important in the model [KM19]. Further, BRTDP also performs badly when there exists an action that almost surely leads to a target but requires a large number of tries (e.g., by the virtue of requiring many rare events to occur). In such cases, BRTDP needs a large number of simulations to find the low-probability paths and even more backpropagation steps to convey the information back. That said, the key advantage of BRTDP is its ability to give ϵ-guarantees while remaining a simulation-based algorithm.

On the other hand, the exploitation and exploration style of value computation, as done in MCTS, has also displayed its tremendous scalability in practice [SHM+16]. However, it does not come with the type of hard guarantees seen in BRTDP. Consequently, we attempt to direct the BRTDP search using ideas from MCTS to obtain the best of both worlds. Firstly, it would give us the search strategy of MCTS. Secondly, it would also give us the guarantees given by BRTDP. Hence, in Paper A, we present three algorithms forming a hybrid between BRTDP and MCTS. The common feature among these algorithms is that they still provide ϵ-optimal results and are immediately extensible to models in which the transition probabilities are not known, following the ideas of [BCC+14, Section 4.2]. To the best of our knowledge, this work is the first to explore the use of MCTS in MDP model checking while providing ϵ-guarantees.

In Paper A, we introduce three different techniques lying on the spectrum between pure MCTS (which does not give guarantees) and BRTDP. Each of our techniques return a guaranteed interval within which the actual optimal probability would lie. Now we describe the different variants of our algorithm. For the sake of completeness, the list also includes the MCTS algorithm without ϵ-optimal guarantees.

MCTS In standard MCTS, the tree policy is determined by the UCB1 values of each state, and the rollout policy uniformly at random chooses the next state. The reward encountered by the simulation r_t is accumulated at the node t from which simulation began in the rollout backup step. In the tree backup phase, the r and n values of each node from t to the root are updated.

Bounded MCTS (BMCTS) In the BMCTS approach, we augment the above mentioned standard MCTS with lower and upper bounds like in BRTDP. In addition to the updates described above MCTS, we maintain additional attributes – L and U – for every state and update them in both the rollout backup as well as the tree backup phases using the Bellman operator. Since we are concerned with the reachability objective, the reward of a rollout is set to 1 if the simulation ends in a goal state, or 0 if the simulation ends either in a sink state or in a end-component not containing any goal states.

MCTS-BRTDP This approach is similar to BMCTS, except that the rollout policy now chooses the action with the best upper bound, like in BRTDP. With this, we get the meaningful exploration strategy of BRTDP while still balancing exploration and exploitation like in MCTS.

BRTDP-UCB In this approach, we replace the BRTDP policy of picking the action with the best upper bound with a policy that picks the action with the best UCB1 bound.

Table 3.1 summarizes the algorithms described above. UCB1 refers to the tree policy selecting the next node based on the UCB1 metric discussed earlier. The rollout policy BRTDP refers to the BRTDP style of simulations by always choosing the action with the best upper bound.

Table 3.1: Spectrum of algorithms ranging from pure MCTS to pure BRTDP (Adapted from Paper A). L and U refer to the lower and upper bounds of the states as computed in BRTDP (Algorithm 1), while r and n are respectively the reward collected through the rollouts and the number of rollouts.

Algorithm	MCTS	BMCTS	MCTS-BRTDP	BRTDP-UCB	BRTDP
Tree Policy	UCB1	UCB1	UCB1	UCB1	BRTDP
Rollout Policy	Uniform	Uniform	BRTDP		
Rollout Backup	—	L, U	L, U	$r, n; L, U$	L, U
Tree Backup	r, n	$r, n; L, U$	$r, n; L, U$		

Concluding Remarks Our experiments show that the best performing algorithm lies in the middle of the spectrum between MCTS and BRTDP (see Table 2, Paper A). Our MCTS-BRTDP approach is consistently close or better than the parent approaches. While BRTDP always starts its simulations from a single initial state, MCTS, due to its tree policy, ends up selecting different states to base the simulations from. This key advantage of MCTS allows it to handle rare-events close to the initial state.

Related Work A closely related but differently aligned area of research is that of Safe Reinforcement Learning (RL) [GF15]. As typical with RL, the base objective is to learn a policy that optimizes the expectation of discounted rewards. There are works that focus on optimizing the worst case expected reward [NE05], using a parameter to balance the reward and the risk-taking tendency [GW05], or optimizing the expected reward over the worst k-quantile [Kas07; KM18]. For a detailed survey, we refer the reader to [GF15]. Another line of work consider approaches to shield the learner from catastrophic damages, e.g. [COM+19; HAK20; JKJ+20].

3.2 Continuous-time Markov Decision Process (CTMDP)

CTMDPs [How60] are a class of Markov processes in which transitions may be both non-deterministic as well as governed by random time delays modelled exponentially. They are a popular formalism in many fields, including but not limited to Operations Research, Queuing Theory, Scheduling and Distributed Systems.

Example 3.2.1 (Motivating time-bounded reachability)**.** Figure 3.5 shows a CTMDP with 4 states $\{s_0, s_1, s_2, s_3\}$ out of which s_0 is an initial state and s_3 is a goal state. The initial state s_0 has an action a, which leads to s_2 after an exponential distributed delay with a mean of 1/2 seconds. s_0 additionally has an action b that leads to either s_1 (with probability 6/12) or s_2 (with probability 6/12) after an average delay of 1/12 seconds. If the system reaches state s_1 or s_2, then the only available actions take the CTMDP to the goal after average delays of 1 second and 1/7 seconds respectively.

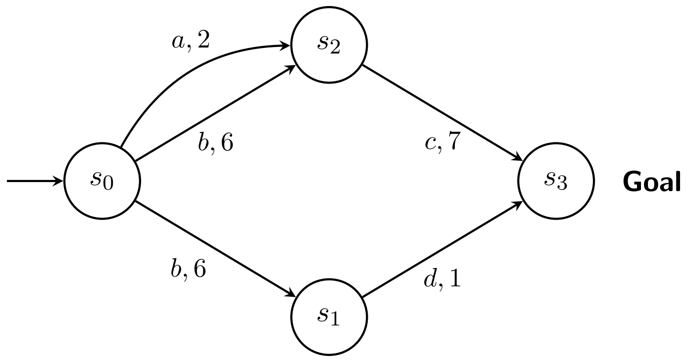


Figure 3.5: An example continuous-time Markov decision process.

Now, one may ask, what is the best strategy to adopt in order to reach the goal within 1 second? In our particular example, the strategy just involves choosing between actions a and b. Action b looks like a high risk-high reward option, since the CTMDP might move quickly to the state s_2 from where it can reach the goal on an average of 1/7 seconds. On the other hand, action a mitigates the risk of ending up in s_1 from where it takes longer on average to reach the goal.

The problem discussed in the above example can be automatically solved using a time-bounded reachability algorithm. Before we discuss the particulars of time-bounded reachability algorithms, let us define formally the notion of CTMDPs.

Definition 3.2.1 (Continuous-time Markov Decision Process)**.** A Continuous-time Markov Decision Process (CTMDP) is a tuple $C = (S, s_{init}, A, Av, \mathbf{R}, \imath)$, where

- S is a finite set of states
- s_{init} is the initial state
- A is a finite set of actions
- $\mathbf{R} : S \times A \times S \to \mathbb{R}_{\geq 0}$ is a rate matrix
- $Av : S \to A$ gives the set of actions available in a state and is given by $Av(s) = \{\alpha \in A \mid \exists s' \in S : \mathbf{R}(s, \alpha, s') > 0\}$.
- $\imath \subseteq S$ is a set of target or goal states.

During a run of the CTMDP, the time spent in each state is referred to as the *residence* or *sojourn* time. An *infinite path* in a CTMDP, $\rho = s_0 \xrightarrow{a_0 t_0} s_1 \xrightarrow{a_1 t_1} s_2 \ldots$, is identified by a sequence of tuples comprising state, action, and residence time. The set of infinite paths is denoted by *Paths*. A *finite path* is a finite prefix of some infinite path. The set of all finite paths is denoted by *Paths**. We use the notation $\Diamond^{\leq T}\imath$ to denote all infinite paths that see some state in $\imath$ within T time units, i.e,

$$\Diamond^{\leq T}\imath = \{s_0 \xrightarrow{a_0 t_0} s_1 \xrightarrow{a_1 t_1} \cdots \in Paths \mid s_0 = s_{init} \wedge \exists i \in \mathbb{N} \cdot s_i \in \imath \wedge \sum_{j=0}^{i-1} t_j \leq T)\}$$

As with MDPs, CTMDPs can also be controlled by controllers or schedulers. The schedulers in CTMDPs however choose actions not only based on the path so far, but also based on time, as their name suggests.

Definition 3.2.2 (Controller)**.** A controller, also known as a scheduler in the context of CTMDPs, is defined as in Definition 2.3.1, instantiating configurations to the Cartesian product of state and time, i.e. $C := S \times \mathbb{R}_{\geq 0}$. A scheduler is hence a map $\pi : (S \times \mathbb{R}_{\geq 0} \times A)^* \times S \times \mathbb{R}_{\geq 0} \to \mathit{Dist}(A)$.

We only consider the class of measurable schedulers as defined in [WJ06; Neu10]. Out of the many variants of schedulers studied in literature [WJ06; Neu10; RS11], this variant is called a timed history-dependent randomized scheduler. In every state, the next action is sampled from the distribution $\pi(\rho, t)$ where ρ is the path taken to arrive at this state and t is the time elapsed in the state until the decision is taken. For a CTMDP C, fixing a controller π and an initial state s yields a unique probability measure $\mathbb{P}^{\pi}_{C,s}$ which, for every set of infinite paths, assigns a value $\in [0, 1]$.

3.2.1 Model checking time-bounded reachability for CTMDPs

The problem of computing the unbounded reachability objective on CTMDPs can be reduced to the same objective on an MDP (covered in Section 3.1.1) obtained using the well-known uniformisation technique [GHP+06]. A more interesting objective for continuous-time models such as the CTMDP is that of time-bounded reachability, as we had seen in Example 3.2.1. Here, the question is "is the maximum (or minimum) probability of reaching $\mathtt{1}$ from s_{init} within t time units greater than a threshold p?".

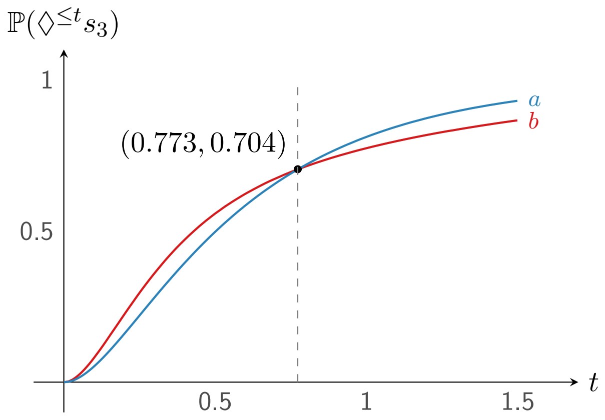


Figure 3.6: Figure denoting the probabilities of reaching the target within time t for actions a and b, for the CTMDP given in Figure 3.5. If the time remaining is less than 0.773 seconds, it is better to play action b, whereas if the deadline is more 0.773 seconds away, the option that maximizes the probability of reaching the target is action a.

The optimal controller in CTMDPs typically depends on the time remaining until the time bound is reached. For example, consider the CTMDP from Example 3.2.1 where there is a choice between actions a and b in the initial state. In Figure 3.6, we plot the probability of reaching the target against the remaining time t, for both the actions. It can be seen that the optimal action changes from b to a when the deadline is farther than 0.773 seconds.

Formally, time-bounded reachability can be defined as follows.

Definition 3.2.3 (Optimal Time-bounded Reachability Probability)**.** Given a CTMDP C, a state $s \in S$, and a deadline $T \in \mathbb{R}_{\geq 0}$, the optimal time-bounded reachability probability (or *value*) is defined as

$$\mathbb{V}_s(T) := \operatorname*{opt}_{\pi \in \Pi} \mathbb{P}^{\pi}_{C,s}(\Diamond^{\leq T} \mathbf{\imath})$$

where opt $\in \{\inf, \sup\}$ and Π is the set of general measurable schedulers.

Remark 2. [Neu10; RS11] prove that there exists a timed pure (non-randomized) memoryless controller of the form $\pi : S \times \mathbb{R}_{\geq 0} \to A$ that can achieve the optimal time-bounded reachability value achieved by the optimal timed history-dependent randomized controller.

The model checking problem on CTMDP is phrased as follows:

Problem 3 (Time-bounded Reachability (TBR))**.** Is the maximal (minimal) probability of reaching a set of goal states $\mathbf{\imath}$ from some initial state s_{init} within a deadline T greater than a threshold p? More precisely, is $\mathbb{V}^{\text{opt}}_{s_{init}}(T) > p$.

The witness to the above problem is, like in MDPs, a controller that achieves the threshold. The controller synthesis problem may independently be formulated as follows.

Problem 4 (Optimal TBR Controller)**.** Given a CTMDP C, an initial state s, a set of goal states $\mathbf{\imath}$, and a deadline T, what is the controller π^{opt} that achieves the optimal time-bounded reachability value? More formally,

$$\pi^{\text{opt}} = \arg \operatorname*{opt}_{\pi \in \Pi} \mathbb{P}^{\pi}_{C,s}(\Diamond^{\leq T} \mathbf{\imath})$$

where opt $\in \{\min, \max\}$ and Π is the set of timed pure memoryless controllers.

State of the art The decidability of the decision version of the TBR problem is still open. However, it was recently shown to be conditionally decidable subject to Schanuel's conjecture [MSS20]. For Continuous-time Markov Chain (CTMC), the well-known subclass of CTMDPs with a single action in each state, optimal TBR is decidable [ASS+00].

Starting from the seminal work of Miller [Mil68], various works have shown the existence of deterministic optimal TBR schedulers for more general classes of the model. A comprehensive review can be found in [Put94, p. 574] as well as [Neu10; RS11].

Currently, all known CTMDP algorithms, at best, compute ϵ-optimal TBR values. [BHK+05] gives the first TBR model checking algorithm for the restricted class of uniform CTMDPs, those in which the sojourn time, i.e., the distribution of time spent in a state, is same for all states. [NZ10] develops the first algorithm for a slightly less restrictive class of locally uniform CTMDPs, those in which the sojourn time in a state does not depend on the chosen action. They show that late total time memoryless deterministic schedulers suffice to resolve non-determinism in an optimal way. Classical algorithms for TBR work by discretizing the time horizon into intervals and approximating the value for each interval. However, such algorithms are typically inefficient. On this matter, [But20] comments (quoted verbatim): "As we can see from [But20, Figure 1.2], it is enough to discretize the time horizon with roughly 2 intervals (...) The discretization-based algorithms for CTMDPs and MA that are known to date use from 85 to $2 \cdot 10^7$ intervals to analyse this model with precision 10^{-6}." One of the directions towards more efficient analysis of CTMDPs is that of coming up with coarser discretizations [FRS+11; FRS+16]. Other approaches [BS11; BHH+11] divide the time-horizon into intervals of variable length, adapting to each problem. Taking an alternate approach from discretization, [BHH+15] presents an technique for approximating both early and late schedulers for arbitrary CTMDPs, building on uniformization and untimed analysis [BHK+05]. The ideas presented in Paper B have been extended to a closely related formalism of Markov automata [EHZ10] in [But20].

3.2.2 Contribution: A new time-bounded reachability framework

A common problem with the algorithms described above is that they don't scale well to large models. Again, like with MDPs, we make an attempt to solve the state-space explosion problem though partial exploration. In Paper B, we introduce a novel framework for time-bounded reachability analysis that can be used to identify the important parts of the state space using simulations. Like in Paper A, the inspiration for this work comes from BRTDP. We exploit the fact that it is not necessary to have full information about the model in order to compute the TBR value with ϵ-precision. Our framework can be instantiated with any of the existing TBR algorithms that can, for each state of the model, provide the TBR value up to ϵ-precision. Hence, if the model is amenable to partial exploration, i.e., there exists a small important subset of the model that chiefly contributes to the TBR value, our framework can be used to speed up any existing TBR algorithm.

The key component of our framework is the algorithm depicted in Algorithm 1 of Paper B, which we summarize here. The algorithm consists of 5 steps, which we now describe intuitively:

1. **Obtain the relevant subset of states.** In this step, we run simulations following a simulation policy π_{sim} that can be instantiated in various ways (see Section 3.4,

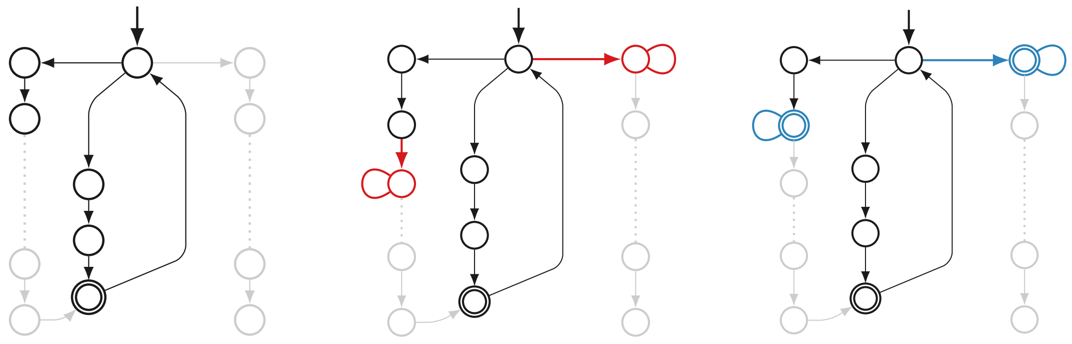


Figure 3.7: Obtaining the under-approximating (centre) and over-approximating (right) models using Step 2 from the partial model identified in Step 1 (left). Illustration adapted from Fig. 4, Paper B.

Paper B). The model is restricted to the states visited in this step to obtain a partial model M' (depicted in Figure 3.7, left).

2. **Under- and over-approximations.** Two models are constructed out of the partial model M'. In the first model $\underline{M'}$, which we call the under-approximating model, all states at the periphery of the explored states, i.e. states whose successors were not visited, are turned into $\mathsf{0}$ states with a self-loop if they are not already $\mathsf{1}$ states (depicted in Figure 3.7, centre). In the second model $\overline{M'}$, called the over-approximating model, all states at the frontier are turned into $\mathsf{1}$ states with self-loops (depicted in Figure 3.7, right).
3. **Computing TBR value.** Now, the TBR value is computed using any of the existing algorithms, e.g. [NZ10; BS11; BHH+15].
4. **Refine current partial model through more simulations using π_{sim}.** In this step, we grow our knowledge of the system by refining the partial model using further simulations. π_{sim} may additionally use the previously computed TBR values to direct the search. Steps 2, 3, and 4 are then repeated.
5. **Termination and result.** We terminate once the difference in TBR values of $\underline{M'}$ and $\overline{M'}$ is less than ϵ.

Concluding Remarks Our results show that many models have a small subset of states that our algorithm is able to identify. If the sub-CTMDP is indeed identifiable, then classical algorithms can be sped up using our framework (see Section 4, Paper B). At the same time, like in the case of MDP, our techniques don't perform too well in models where there does not exist reasonably sized sub-CTMDP suitable for computing TBR. In Section 4.1 of Paper B, we present some preliminary results to show that the models on which our technique perform poorly do not contain suitable sub-CTMDPs.

4 Controller Representation

Controllers are objects that decide how a system should behave in every state. There are various ways in which one can automatically synthesize controllers — reinforcement learning [SB98], dynamic programming [Ber05], model checking [CHV+18; BK08], and reactive synthesis [Fin16], to name a few. Controller synthesis of hybrid systems through discretization, construction of finite abstractions or showing bisimulations with symbolic models and solving them using formal verification is yet another approach to obtain controllers guaranteed to satisfy certain specifications [Tab09]. However, with all such non-trivial automatic controller synthesis approaches, the result is typically a large and incomprehensible *lookup table* mapping every valid state to the allowed action or actions. While the correctness of controllers is guaranteed by the algorithms that synthesize them, we turn our interest to data structure used to store or represent the controller function. We consider two aspects of controllers that have increasing importance in this day and age.

Size While Moore's law has held true for almost half a century, typical embedded devices still have very small amounts of memory relative to personal or enterprise-level computing systems. This meagre amount of memory is spent in, among other things, storing drivers to interface with sensors, performing wired or wireless communication, running the real-time operating system, in addition to storing controllers. Hence, it becomes extremely important that the controllers flashed onto such devices are conservative in terms of memory as well as usage of computational and communication resources.

Explainability As with most complex systems, especially those run by machine learning "blackboxes", there is a growing demand for explainability and transparency. In the fields of machine learning and artificial intelligence, explainable AI (XAI) [DSB17; GCH+18] has evolved into an important subfield. With the advent of sophisticated controllers controlling safety-critical systems, it becomes even more important to be able to explain how and why the system is controlled in a certain way, especially for auditing and certification. Human-interpretability is also useful in validating the correctness of the automatically synthesized controller. A domain expert may be able to identify oddities or bugs that may have arisen due to the model being an inaccurate representation of the system, or due to a bug in the formal synthesis tool.

An ideal controller would be that which is:

- perfectly explainable
- compact, fit in the memory of restrictive hardware

- require minimal bandwidth to actuate controllable devices
- quick and efficient at identifying the next decision

In our work on controller representation, we try to address these points. In the following sections, we give a brief introduction to existing controller representations such as lookup tables and binary decision diagrams (BDDs), and introduce the reader to the emerging line of research using decision trees (DTs) to represent controllers. We present, to the best of our knowledge, the first results on using decision trees to represent numeric controllers obtained from controller synthesis of hybrid MDPs as well as non-probabilistic Hybrid systems.

Note that we restrict ourselves to pure memoryless controllers, i.e., controllers that only depend on the current state, and do not randomize between actions, as defined in Definition 2.3.1. More precisely, we only consider controllers of the form $\pi : C \to 2^A$.

4.1 Preliminaries

We now introduce three data structures, lookup tables, Binary Decision Diagrams (BDDs) and Decision Trees (DTs), which we will be using in rest of this chapter.

4.1.1 Lookup tables

A lookup table is an explicit representation storing key-value pairs. Formally, a lookup table is a relation $T \subseteq K \times V$ where K is a set of keys and V is a set of values. Both the keys and the values may be high-dimensional, composed of multiple numeric, or non-numeric components. An example of a lookup table is given in Figure 4.1a. In this table, the first three columns together make up the key and the last column makes up the value.

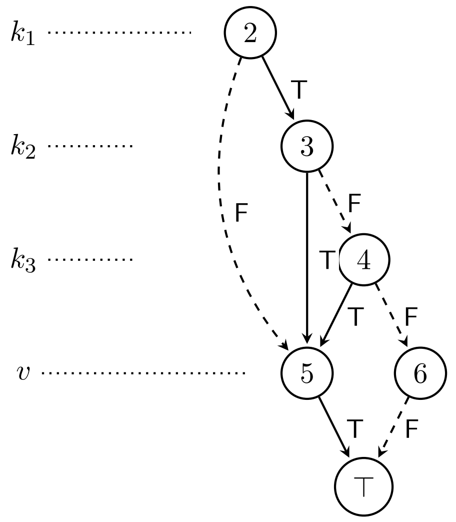
4.1.2 Binary decision diagrams

A Binary Decision Diagram (BDD) is a data structure used to represent Boolean functions, or more generally to store a set of binary strings. Hence, they can also be used to store sets and relations by encoding the variables or objects in binary. Various improvements exist such as Zero-Suppressed Decision Diagrams (ZDDs), Algebraic Decision Diagrams (ADDs) or Multi-Terminal Binary Decision Diagrams (MTBDDs), where the constants come from arbitrary finite domains. We refer the reader to [CHV+18, Ch. 7] or [And97] for a more gentle introduction to BDDs.

Example 4.1.1 (Looking up a BDD)**.** Consider a function $f : (K_1 \times K_2 \times K_3) \to V$ represented in the form of the lookup table in Figure 4.1a. Using standard (reduced ordered) BDD construction algorithms and the variable ordering $k_1 < k_2 < k_3 < v$, we get the BDD given in Figure 4.1b. Now, for instance, we can check if $f(1, 0, 1) = 1$ as follows. We start at the highest level corresponding to the variable k_1. Since we want to check the case where $k_1 = 1$, we follow the T to the node $\mathsf{2}$ in the next level. Since

k_1	k_2	k_3	v
0	0	0	1
0	0	1	1
0	1	1	1
0	1	0	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

(a) Lookup table



(b) Binary Decision Diagram

Figure 4.1: A lookup table (left) and the BDD corresponding to it (right). Variable ordering of the BDD is indicated on its left. True edges are marked with a solid line and false edges are marked with a dashed line.

we have $k_2 = 0$ and $k_3 = 1$ we take the F edge to 4 followed by the T edge to 5. Finally, since $v = 1$, we follow the T edge to $\top$ which indicates that f indeed maps $(1, 0, 1)$ to 1. Notice that in the BDD, certain nodes do not have one of the outgoing edges. This is due to the $\perp$ node being removed from the BDD for conciseness. For example, if we query the BDD for $k_1 = 1$, $k_2 = 0$, $k_3 = 1$, $v = 0$, we realize that there is no F edge at node 5, and hence conclude that f does not map $(1, 0, 1)$ to 0.

4.1.3 Decision trees

Decision Trees (DTs) are a formal decision-making tool used frequently in machine learning as models for classification and regression [LL14, Ch. 8]. Classification is the problem of mapping an input to an output or an attribute to a label. In supervised learning, a model such as a DT is *trained* using known attribute-label examples in order to predict the labels of unseen attributes. Regression is similar to classification in that regression also intends to map inputs to outputs, however, in the case of regression, the attributes are mapped to a value rather than being labelled.

For a formal definition and semantics of a decision tree, we refer the reader to Section 3 of Paper D. However, intuitively, they may be naturally interpreted as a branching program (nested if-else blocks).

Example 4.1.2 (Representing a dataset using a decision tree)**.** Let us consider a classic example of classification. Suppose that you are getting more e-mails that you can handle and you want to set up a spam filter. You have come to realize that certain e-mails are obvious spams, certain other e-mails are obvious non-spams and the rest need further

Table 4.1: Sample dataset for Example 4.1.2 showing the number of occurrences of different keywords in 5 e-mails, along with known labelling as spam, not spam, and unknown.

Number of occurrences							Label
prince	inheritance	casino	magazine	reminder	discussion	meeting	
2	3	0	0	0	0	0	spam
0	0	3	0	0	0	0	spam
0	0	0	2	1	0	0	unknown
0	0	0	0	0	1	1	not spam
0	0	0	0	1	0	1	not spam

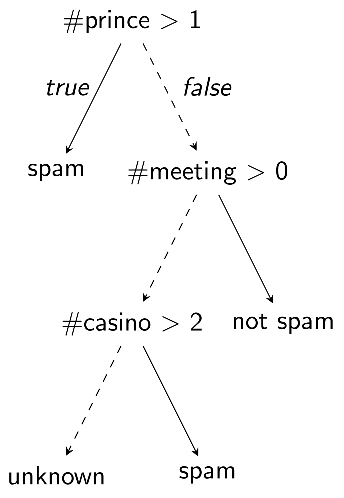


Figure 4.2: A decision tree classifying e-mails as spam, not spam, or unknown, constructed from the dataset given in Table 4.1.

investigation, based on just the number of occurrences of certain keywords. Table 4.1 summarizes your observation. From the above data, can you classify a new e-mail containing 1 occurrences each of "prince", "casino" and "discussion" as "spam", "unknown" or "not spam"?

We represent the dataset in Table 4.1 as a decision tree in Figure 4.2. Given any tuple consisting of the number of occurrences of the various keywords, a label may be found as follows. Evaluation begins from the root node (top of the tree) down the tree, ending up in a leaf node (a node without any children). The predicate in each node of the tree is evaluated based on the input tuple and either of the *true* or *false* branches is followed to the next node. The evaluation ends at a leaf node, thereby obtaining a label for the input tuple.

For every tuple of keyword occurrences in the given dataset, our decision tree is able to give the correct label. One can immediately notice that this tree does not even depend on certain variables such as the number of occurrences of the "magazine" keyword. In fact, only three of the keywords are necessary to complete this classification task. In contrast, if this table were to be represented using a BDD, then it would have 13 levels (2 bits each for the keywords "prince", "inheritance", "casino" and "magazine", 1 bit each for the remaining keywords, and 2 bits for the label).

Remark 3 (Overfitting). Note that the tree we constructed is able to map every entry in the given dataset to the correct label. In machine learning, this is termed *overfitting* and is typically encountered when a practitioner discovers that a learnt classification model fails to generalize to new data even though it gives a very high accuracy on the dataset used for training.

The above example illustrates the use of classification models in machine learning. By analysing the data presented in Example 4.1.2, we can manually come up with a set of rules, that may be concisely represented diagrammatically, in the form of a DT, as shown in Figure 4.2. However, various algorithms exist to automatically learn such trees, notably CART [BFO+84], ID3 [Qui86] and its successor C4.5 [Qui93].

Intuition behind decision tree learning algorithms We can view a decision tree learning algorithm as one that takes a lookup table as input and produces a decision tree representing the lookup table precisely as output. While traditional machine learning techniques try to produce "generalized" decision trees that are able to label unseen data, we focus on algorithms that can exactly represent the input lookup table (see Remark 3). Let the lookup table, also called the dataset in DT learning terminology, be a tuple of input vectors with their respective labels, i.e. $D = (X, Y)$. Intuitively, a decision tree algorithm tries to recursively split the high-dimensional space containing X into mutually exclusive boxes (or subsets) containing identically labelled inputs. In the following explanation, we call a box *homogeneous* if all inputs in the box are labelled identically. The algorithm follows a recursive procedure with each call containing two basic steps.

1. In each step, the algorithm tries to split the space into two boxes with the help of a predicate, which may be generated by a *predicate generator*.

2. The predicates output by the generator are evaluated using a *predicate selector*, which selects the predicate that splits the current space into two maximally homogeneous boxes. The homogeneity is measured with the help of a measure called the *impurity measure*.

These steps are recursively applied on the boxes resulting from every split, until both of the boxes are completely homogeneous.

4.2 State of the Art

Controllers can be represented using various data structures such as BDDs, ADDs, and DTs in addition to the most straightforward lookup tables. In this section, we discuss the existing state-of-the-art in controller representations.

4.2.1 Lookup tables

Lookup tables are one of the simplest and ubiquitous controller representations. Most model checking and controller synthesis tools, e.g. PRISM [KNP11], Storm [HJK+20], SCOTS [RZ16], or Uppaal Stratego [DJL+15], typically come with the ability to output their controllers as lookup tables.

The key advantage of lookup tables is the lookup speed, especially when the table is implemented on specialized hardware such as Field Programmable Gate Arrays (FPGAs). However, they are heavily disadvantaged since (i) they tend to be large due to their verbosity; and (ii) they are not explainable or interpretable unless they contain only a few lines.

4.2.2 Binary decision diagrams and their extensions

BDDs have been used in various areas such as formal verification of hardware, very large-scale integration computer-aided design (VLSI CAD) [Bry95; Min12], solving combinatorial problems [Min93], and symbolic model checking [BCM+90] among others. Inspired by their use in symbolic model checking, BDDs were first used to represent controllers (referred to as universal plans) in [CRT98]. [HSH+99; SHB00] uses Algebraic Decision Diagrams (ADDs) [BFG+97], an extension that allows non-boolean values at terminals, to represent MDPs controllers. Similar to our work, [DIL+09] proposes an automatic compression technique for numerical controllers, however using Ordered BDDs. [ZVJ18] considers the problem of determinizing controllers optimally for BDD representations.

Advantages BDDs can represent Boolean functions compactly, and are amenable fast logical manipulation [Bry92; Bry18], for e.g. operations such as union, intersection, and negation take only polynomial time. They are also quite good at compressing controllers given a good variable order and, for permissive controllers, a good determinization strategy [ZVJ18]. Moreover, the resulting BDD representation can directly be implemented in hardware.

Disadvantages That said, BDDs are incomprehensible, especially when representing sets or relations containing numbers or other non-Boolean objects such as tuples of numbers, as common in controllers. This is mainly due to the fact that each branching node in a BDD performs a comparison on a single bit of some object's binary encoding. Moreover, the size of BDDs is extremely sensitive to variable ordering.

For example, recall the lookup table and BDD of Figure 4.1. The variable ordering chosen in the figure is $s_1 < s_2 < s_3 < a_1$, which results in a BDD with 8 nodes. Figure 4.3 shows the BDD that is obtained by choosing an alternate ordering: $a_1 < s_3 < s_2 < s_1$. This BDD has eight nodes as opposed to six nodes in Figure 4.1b, which is a 33% increase in size. We computed the BDDs corresponding to all 4! variable orderings for the lookup table (Figure 4.1a). Out of the 24 BDDs, 5 have six nodes, 6 have seven nodes and 13 have eight nodes.

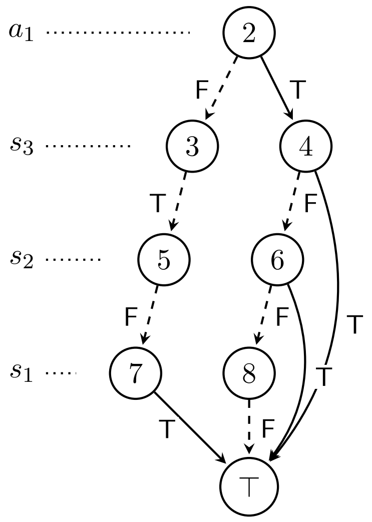


Figure 4.3: Sensitivity of BDD size to variable ordering. This figure depicts the BDD representing the lookup table in Figure 4.1a, however with a different variable ordering compared to the BDD in Figure 4.1b.

While this is a toy example, similar behaviour can be observed even on very large BDDs [ZVJ18]. The problem of finding the best variable ordering is NP-complete [BW96].

4.2.3 Decision trees

Use in Reinforcement Learning In their work on structured policy iteration, [BDG95] use DTs for exploiting structure in policy construction in MDPs. Both value functions (mappings from state to values) and policies are represented using decision trees while the MDP itself is represented using a Dynamic Bayesian Network (DBN). [BDH99] surveys the decision-theoretic planning literature covering structured/factored representation. [PH+01; Pye03] use decision trees to represent the value functions during reinforcement learning. [CDB07] uses first-order logical DT [Blo99] in relational reinforcement learning to learn a relational decision tree that predicts whether an action will be executed by the policy. [RTJ+19] is similar to the work of Pyeatt [Pye03], however, save on the size of the decision tree by using better splitting heuristics. In all of these works, the DT construction is tightly coupled with the respective (reinforcement) learning algorithms. Unlike these works, our techniques can be applied out of the box to any algorithm that produces policies or controllers in the form of lookup tables. A work quite similar to ours is [MYA12], that post-processes policies learned through reinforcement learning to obtain fuzzy decision trees [Jan98]. However, their algorithm cannot learn permissive controllers and is not tailored to produce DT controllers that preserve the guarantees of the original controllers.

Use in model checking and controller synthesis [Gir12] claims to use ADD [BFG+97] for representing a controller, however, uses DTs without explicitly giving an algorithm to learn them. [BCC+15] makes use of decision trees to learn minimal counterexamples in probabilistic verification, however, their method only produces ϵ-correct counterexamples. [BCK+18] uses decision trees to represent guaranteed strategies from reactive synthesis, however, our work differs from theirs in three ways. First, they use a binary representation of the data (similar to BDDs), which means that their predicates can only test if a Boolean combination of variables is *true* or *false*. In our work, model variables are allowed to take values from infinite but ordered sets or finite unordered sets. Second, the trees produced by [BCK+18] need to be queried with a state-action (s, a) pair to figure out if action a can be played in state s. In our trees, the leaf nodes of the trees contain playable actions so that querying a state s will immediately give the allowed action. Third, [BCK+18] requires the positive as well as negative examples to teach the DT allowed as well as disallowed actions. Consequently, their dataset has to be "complete" enumerating every possible state and action pairs. On the other hand in our work, we can directly use a synthesized controller with exclusively positive examples, without the need of enumerating all state-action pairs.

[ABC+19], which is co-authored by the author of this dissertation but not included here, builds upon the ideas presented in Paper C and introduces the use of linear predicates in the nodes of decision trees representing controllers obtained from reactive synthesis as well as model checking. The thesis [Jac20] builds upon Paper D and introduces various improvements to our tool `dtControl` such as the ability to handle categorical predicates and improved determinization strategies.

Apart from their use in representing controllers and policies, DTs are often used to increase confidence in opaque machine learning models by bringing in comprehensibility. For example, [ABT16; AA19; ZYM+19; WDH+20] use DTs to explain the decisions taken by neural networks.

4.3 Contribution: Improved Decision Tree Representations

As we saw in the previous section, BDDs are extremely sensitive to variable ordering. Even with the perfect variable order, BDDs are hardly explainable unless the domain of our controller only contains boolean variables. DTs are highly interpretable and explainable compared to BDDs. Moreover, they can handle ordered (numeric) as well as unordered (categorical) variables. While BDDs or ADDs may contain only boolean comparisons in their inner nodes, DTs may contain linear, polynomial, or even general algebraic predicates. Further, while the problem of computing the optimal DT is NP-complete, the greedy algorithms for DT construction using various splitting heuristics (called *impurity measures*) typically perform very well in practice. The rich DT learning literature also offers specialized algorithms for producing trees with more than one decision in the leaf nodes. Moreover, attribute value grouping [Qui93; Jac20, Sec. 5.2.2] allows constructing trees with multi-way splits in decision nodes. As we shall see in the following sections, these features make DTs an attractive model for representing controllers.

We now outline the contributions of Papers C and D. Both these papers tackle the problem of representing controllers concisely and explainably while also preserving the guarantees of the original controller. In Paper C, we build and present the Stratego$^+$ framework, which tightly couples with Uppaal Stratego to produce safe, small, and (nearly) optimal controllers. In Paper D, we present an open-source tool with interfaces to multiple controller synthesis tools such as Uppaal Stratego and SCOTS. Additionally, in both the papers, we present different adaptations of the DT learning algorithm CART [BFO+84] to learn exact representations of the controller while offering multiple choices to preserve or sacrifice permissiveness.

The goal of decision tree learning in machine learning is not to represent key-value pairs as we are doing here, but to learn from key-value pairs a classifier that can predict the value of unseen keys, or in other words, generalize to new data (recall Remark 3). To achieve this goal, DT learning algorithms in Machine Learning (ML) additionally feature certain peculiarities such as early stopping, pruning, and various other tricks to reduce overfitting. These techniques typically bring down the size of the DT during the learning phase itself or post-learning. In our case however, the data we want to learn come from guaranteed controllers obtained through model checking or other automated controller synthesis techniques. In order to preserve these guarantees, the controller must be represented exactly. This necessitates special care when adapting the tree construction algorithms.

To the best of our knowledge, our work is the first to use DTs to represent numeric CPS controllers obtained from controller synthesis of hybrid MDPs as well as non-probabilistic Hybrid systems, while preserving guarantees of the original controller. Naturally, the new techniques presented here may also be extended to any pure memoryless controller.

4.3.1 Stratego$^+$ framework

Our first contribution towards controller representation is the Stratego$^+$ framework. In order to grasp the idea in its entirety, it is necessary to give an overview of how Uppaal Stratego [DJL+15] learns and optimizes controllers. In Figure 4.4, we depict a simplified schematic diagram of Uppaal Stratego. In the first step, the model M given to Uppaal Stratego is abstracted into a timed game G. Uppaal Tiga [BCD+07] is used to synthesize a controller satisfying a certain specification. This controller is stored in the form of a lookup table, π. Now, Uppaal Stratego is given an optimization query alongside the model and the lookup table that ensures some specification. Uppaal Stratego uses multiple reinforcement learning algorithms on the model restricted by applying the controller produced by Uppaal Tiga on it, in order to optimize and obtain a nearly-optimal controller π_{opt} satisfying an optimization query.

The Stratego$^+$ framework integrates DTs in Uppaal Stratego in two different ways as illustrated in Fig. 2 of Paper C.

- In the first approach (part of Figure 4.4 on the right, drawn in 🟧), the controller π output by Uppaal Tiga is converted into a DT. In the reinforcement learning step, the simulations are run on the model restricted to the actions given by the

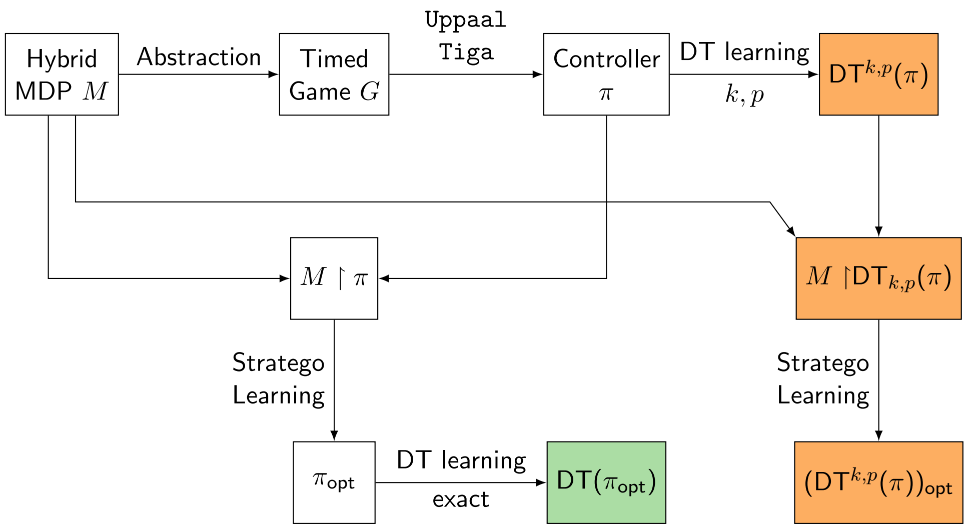


Figure 4.4: A schematic diagram showing how `Uppaal Stratego` synthesizes near-optimal strategies, along with the two new additions in `Stratego`$^+$ (drawn in 🟩 and 🟧). The parameters k and p, described in more detail in Section 3.4 of Paper C, are used to obtain a balance between size and optimality of the resulting DT-basec controller.

DT-based controller. The output of the RL step is a preference over actions in each state. We use this preference on the DT in order to obtain a deterministic, small, guaranteed and optimal controller $(\mathsf{DT}^{k,p}(\pi))_{\text{opt}}$.

- In the second approach (bottom center of Figure 4.4, drawn in 🟩), the DT learning step after `Uppaal Tiga` produces a guaranteed controller is eliminated. Instead, the RL takes place on the model restricted to the actions given by the `Uppaal Tiga` controller π. The action preference obtained from RL and the `Uppaal Tiga` controller are fed to DT learning algorithm, which then produces a deterministic tree $\mathsf{DT}(\pi_{\text{opt}})$ that is a small, guaranteed and optimal controller.

4.3.2 The `dtControl` toolbox

Our second contribution under the theme of controller representations is the tool `dtControl`, presented in Paper D. `dtControl` is a toolbox enabling researchers and practitioners to experiment with representing controllers as DTs. Currently, `dtControl` can be used to represent non-randomized memoryless controllers $C : S \to 2^A$, where S is a set of states where each state is an n-tuple of numeric variables[1], and A is a set of actions where each action is an m-tuple of numeric or non-numeric but finite variables. `dtControl` contains numerous tuneable parameters allowing users to select from a multitude of implemented algorithms.

[1]`dtControl` was recently extended [Jac20] to handle categorical variables, i.e. variables that take a finite set of unordered discrete values.

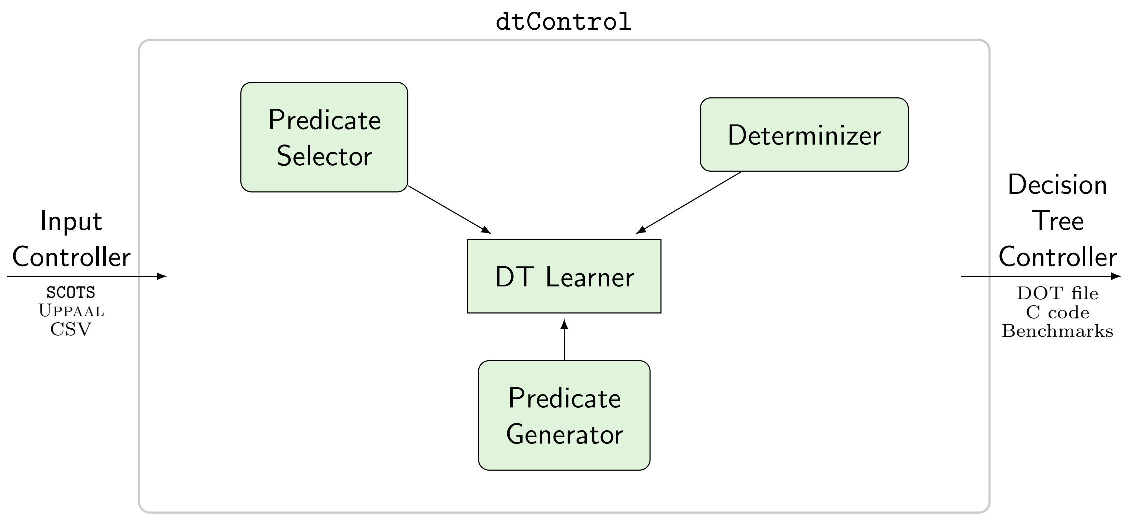


Figure 4.5: A schematic diagram of `dtControl` showing its various components and their interactions.

The tool

A schematic diagram of `dtControl` is given in Figure 4.5. Given an input controller in the form of a lookup table in one of the various supported formats, `dtControl` constructs a decision tree representing the input controller exactly (without loss of information). Additionally, if the input controller is permissive, then `dtControl` can be configured to reduce the permissiveness and typically obtain an even smaller tree. The final decision tree controller can be exported as a DOT file (Graphviz) or as a C file containing a nesting of if-else blocks.

We now discuss the key components of the `dtControl` machinery:

- **Determinizer** that can resolve the non-determinism (or reduce the permissiveness) in the controller according to a user-chosen strategy (e.g. minimize the norm of the control input, or randomly choose between the available actions).
- **Predicate generator** that generates predicates using variables and constants from a given domain, typically derived from the loaded data.
- **Predicate selector** that computes the quality or effectiveness of predicates at splitting subsets of the pre-processed data.
- **Decision tree learner** that takes in the processed input, queries the predicate generator for predicates, discriminates between predicates and chooses the best predicate with the help of the predicate selector and constructs a decision tree. The decision tree learner uses a modified version of the CART algorithm [BFO+84].

Each of the components are modular and can be easily instantiated for different needs. `dtControl` also provides a list of preset configurations that have been found to perform best in our experiments. Additionally, we also provide a user and developer manuals on the official website dtcontrol.model.in.tum.de.

Adaptations of the learning algorithms

We make the following deviations from standard DT learning algorithms used in machine learning:

- Do not perform any steps that prevent overfitting. This is achieved by splitting nodes until each of them contain homogeneously labelled data points.
- Allow arbitrary predicates at nodes, not just axis-parallel (e.g., $x > c$). For this, we introduce the idea of a predicate generator that may generate any predicate, later to be evaluated by the predicate selector. Currently supported predicate generators include logistic regression, linear support vector machines (SVMs), or OC1 [MKS+93].
- Allow determinization of the controller from within the learning algorithm. We allow determination of the controller both globally (pre-processing) and locally, just before selecting predicates for a particular node.
- Allow steps that reduce the size of the tree, but maintain guarantees. Usage of the minimum split size parameter and safe pruning (Section 3.4, Paper C) allow us to tune the amount of permissiveness in the controller while maintaining guarantees.

Additionally, we introduce a novel determinization technique called MaxFreq (Section 4.2, Paper D) that determinizes parts of the dataset locally just before splitting a particular node in order to obtain smaller trees. Experimental results (Table 1, Paper D) show that using MaxFreq typically gives tiny controllers.

Remarks on size and explainability Our results show that most (pure memoryless) controllers contain some inherent structure that decision trees can exploit in order to reduce size. The decision trees obtained using the `Stratego`$^+$ framework were found to be orders-of-magnitude smaller than the respective lookup tables (Table 1, Paper C). Additionally, we also saw the possibility of sacrificing permissiveness and thereby room for optimization in exchange for smaller trees. With `dtControl`, 5 out of 8 permissive controllers used in our benchmarks (Table 1, Paper D) could be represented with decision trees small enough to be drawn on paper. For most controllers, we also found that the DTs are smaller than a best-effort[2] BDD-based representation (Table 2, Paper D). We believe that these small trees can hence be easily converted into C code and used in embedded applications.

On the aspect of explainability, we found that it is typically easy to interpret the trees produced due to (i) size of the tree being small; as well as (ii) the predicates used in

[2]We optimized the sizes of the BDDs by applying Rudell's sifting algorithm [Rud93] to reorder variables until no better reordering could be found.

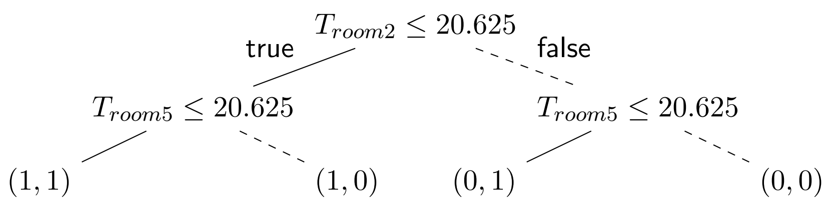


Figure 4.6: Decision tree representation of an automatic room heating system containing 26,244 entries in the lookup table.

the nodes being explainable. For example, a decision tree produced by `dtControl` for an automatic room heating system [JZ17] is depicted in Figure 4.6. While the original lookup table consisted of 26,244 state-action pairs with each state comprising sensor readings from 10 different rooms, `dtControl` discovered that only two sensors were needed to control the system while maintaining the guarantees of the original controller. This shows not just that decision trees can be explainable, but also a consequence of explainability: being able to optimize the implementation after the controller is vetted by a domain expert.

5 Conclusion & Outlook

This publication-based dissertation is a culmination of the author's doctoral work in two directions: (i) tackling the state-space explosion problem in probabilistic model checking using partial exploration; and (ii) representing controllers obtained from quantitative model checking or formal controller synthesis concisely and explainably. We summarize the contributions below and discuss some possibilities for future work.

5.1 Solving State-Space Explosion with Partial Exploration

We presented two techniques to improve existing probabilistic model checking procedures for discrete and continuous MDPs. In particular, in Paper A, we gave multiple algorithms for reachability verification of MDPs inspired by the successes of MCTS in game solving and Artificial Intelligence with the likes of AlphaGo. Our algorithms try to balance exploration and exploitation using the UCB1 heuristic commonly used in MCTS. In order to obtain ϵ-optimality, we interleave BRTDP-style simulations and back-propagations with MCTS. Our best algorithm is at least as good as pure MCTS as well as pure BRTDP, hence making it a very viable choice from among the state-of-the-art partial exploration algorithms for reachability. Its key advantage over BRTDP is that the MCTS part of our algorithm identifies good non-initial states from which the BRTDP-style simulations may start. This allows our algorithm to handle rare events better than BRTDP, as long as the MCTS tree grows to include it.

In Paper B, we introduced an algorithm to speed up a large class of traditional time-bounded reachability (TBR) algorithms, building upon the idea underlying BRTDP of using simulations to identify the "important" states. We compute two sub-CTMDPs (a lower- and an upper-bound CTMDP) based on the states we have seen through simulations and run an existing TBR algorithm on them to obtain a lower bound as well as an upper bound on the actual TBR value. These bounds are then used in order to improve our simulation strategy, which then allows us to grow the sub-CTMDPs, consequently improving our TBR estimates. The iterative algorithm hence can be seen as interleaving an existing TBR algorithm with simulations in order to speed up the said algorithm. The simulations help in identifying the states necessary to estimate the TBR value for the specific model-specification combination, leading to a reduction of the amount of computation needed.

Both the presented papers are small steps into the largely unexplored area of using partial exploration to compute reachability probabilities. While partial exploration does not work for all model topologies, we notice that for certain model-property combinations, there is typically a subset of "relevant" states [KM19] that allows these algorithms to synthesize ϵ-optimal controllers. Characterizing the classes of model-property combinations for which such relevant subsets exist could be a potential direction for future work. With such a characterization, a meta-algorithm could be designed with the ability to choose the best concrete reachability algorithm for the specific problem. A second, more obvious direction of future work could be to extend these approaches to more complex specifications such as mean-payoff (like in [ACD+17]) and LTL.

5.2 Explainable and Concise Representation of Controllers

As our second major contribution, we presented algorithms and tools for representing controllers concisely and explainably using Decision Trees in Papers C and D. In the former, we presented the `Stratego`$^+$ framework to obtain safe, near-optimal and small controllers with the help of `Uppaal Stratego`. We adapted standard DT learning algorithms with the ability to construct DTs maintaining the guarantees of the original controller. Further, we introduced some strategies to minimize the size of the DT representation even further without losing guarantees, however, at the cost of sacrificing permissiveness and consequently, optimality.

In Paper D, inspired by the promising results of representing controllers by DTs, we developed an open-source toolkit called `dtControl`. Here, we implemented various strategies to customize the DT construction, with the ability to instantiate the predicate generator, predicate selector, and determinizer in different ways. Additionally, we introduced a novel determinization strategy, MaxFreq, which can locally determinize parts of the controller during the learning process in order to obtain even smaller representations.

As opposed to representations using BDDs, we do not have to encode the domain and range of the controller in binary. Compared to previous work [BCC+15; BCK+18], our trees are also more meaningful as our predicates do not contain inequalities with unordered variables. Most importantly, we demonstrated that in many examples, we get very small DTs that can be directly read and analysed by domain experts, something that would not have been possible with BDDs and ADDs.

The potential of having explainable DTs is yet to be explored properly. In Paper C, we were able to find a bug in the model that became evident based on the DT representation of the controller. This happens when humans with sufficient domain knowledge are able to determine whether certain actions recommended by the controller are sensible. For the room heating example (`10rooms`) in Paper D, we obtained a decision tree that revealed that only two sensor readings were required to control the system — something that would not have been possible with traditional representations. In an ongoing work, we are using `dtControl` to help robot designers create better models by constructing DT representation of not just controllers, but also reachability values of states using `dtControl`. These representations identify issues such as missing recovery actions. [FZW+17] reminds us that formally verified systems may also err, due to incorrect assumptions made during modelling or formulating the specification. We believe that if controllers are small and explainable, engineers, domain experts, or certification authorities may be able to catch bugs before they materialize.

Various other directions may be explored in the future:

- Currently, we use predicates such as $x < c$ and $ax + by + cz < d$ in the decision nodes of the tree. However, can we design learning algorithms that can automatically discover the domain knowledge embedded in the controller (e.g. equations of motion in a cruise control controller)?
- In the similar vein as in [BCC+15] and the work of Boutilier et al. in reinforcement learning [SHB00; HSH+99; BDH99], can we design model checking algorithms that directly work and manipulate DTs, rather than post-processing the controller output by them?
- A decision tree typically contains duplicate nodes across multiple branches, unlike BDDs or ADDs, which are both decision diagrams without duplication. Can we deduplicate decision trees to obtain decision diagrams, or even directly learn decision diagrams?
- Could the growing field of program synthesis [GPS17] be explored for an alternate way to synthesize small and explainable controllers?

In summary, this thesis presented some ideas to help make quantitative model checking and controller synthesis more scalable, feasible and accessible. We have identified multiple areas for future work, both towards applying partial exploration techniques for probabilistic verification, and in coming up with better controller representations. We believe that these lines of research are increasingly important as the world grows ever more dependent on cyber-physical systems.

Bibliography

[AA19] Stephan Alaniz and Zeynep Akata. 'XOC: Explainable Observer-Classifier for Explainable Binary Decisions'. In: *CoRR* abs/1902.01780 (2019). arXiv: 1902.01780. URL: http://arxiv.org/abs/1902.01780.

[ABC+19] Pranav Ashok, Tomás Brázdil, Krishnendu Chatterjee, Jan Kretínský, Christoph H. Lampert and Viktor Toman. 'Strategy Representation by Decision Trees with Linear Classifiers'. In: *Quantitative Evaluation of Systems, 16th International Conference, QEST 2019, Glasgow, UK, September 10-12, 2019, Proceedings.* Ed. by David Parker and Verena Wolf. Vol. 11785. Lecture Notes in Computer Science. Springer, 2019, pp. 109–128. DOI: 10.1007/978-3-030-30281-8_7. URL: https://doi.org/10.1007/978-3-030-30281-8%5C_7.

[ABH+18] Pranav Ashok, Yuliya Butkova, Holger Hermanns and Jan Kretínský. 'Continuous-Time Markov Decisions Based on Partial Exploration'. In: *Automated Technology for Verification and Analysis - 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings.* Ed. by Shuvendu K. Lahiri and Chao Wang. Vol. 11138. Lecture Notes in Computer Science. Springer, 2018, pp. 317–334. DOI: 10.1007/978-3-030-01090-4_19. URL: https://doi.org/10.1007/978-3-030-01090-4%5C_19.

[ABK+18] Pranav Ashok, Tomás Brázdil, Jan Kretínský and Ondrej Slámecka. 'Monte Carlo Tree Search for Verifying Reachability in Markov Decision Processes'. In: *Leveraging Applications of Formal Methods, Verification and Validation. Verification - 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part II.* Ed. by Tiziana Margaria and Bernhard Steffen. Vol. 11245. Lecture Notes in Computer Science. Springer, 2018, pp. 322–335. DOI: 10.1007/978-3-030-03421-4_21. URL: https://doi.org/10.1007/978-3-030-03421-4%5C_21.

[ABT16] Karim Ahmed, Mohammad Haris Baig and Lorenzo Torresani. 'Network of Experts for Large-Scale Image Categorization'. In: *Computer Vision - ECCV 2016 - 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part VII.* Ed. by Bastian Leibe, Jiri Matas, Nicu Sebe and Max Welling. Vol. 9911. Lecture Notes in Computer Science. Springer, 2016, pp. 516–532. DOI: 10.1007/978-3-319-46478-7_32. URL: https://doi.org/10.1007/978-3-319-46478-7%5C_32.

[ACD+17] Pranav Ashok, Krishnendu Chatterjee, Przemyslaw Daca, Jan Kretínský and Tobias Meggendorfer. 'Value Iteration for Long-Run Average Reward in Markov Decision Processes'. In: *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I.* Ed. by Rupak Majumdar and Viktor Kuncak. Vol. 10426. Lecture Notes in Computer Science. Springer, 2017, pp. 201–221. DOI: `10.1007/978-3-319-63387-9\_10`. URL: `https://doi.org/10.1007/978-3-319-63387-9%5C_10`.

[ACF02] Peter Auer, Nicolò Cesa-Bianchi and Paul Fischer. 'Finite-time Analysis of the Multiarmed Bandit Problem'. In: *Mach. Learn.* 47.2-3 (2002), pp. 235–256. DOI: `10.1023/A:1013689704352`. URL: `https://doi.org/10.1023/A:1013689704352`.

[ACK+20] Pranav Ashok, Krishnendu Chatterjee, Jan Kretínský, Maximilian Weininger and Tobias Winkler. 'Approximating Values of Generalized-Reachability Stochastic Games'. In: *LICS '20: 35th Annual ACM/IEEE Symposium on Logic in Computer Science, Saarbrücken, Germany, July 8-11, 2020.* Ed. by Holger Hermanns, Lijun Zhang, Naoki Kobayashi and Dale Miller. ACM, 2020, pp. 102–115. DOI: `10.1145/3373718.3394761`. URL: `https://doi.org/10.1145/3373718.3394761`.

[AD90] Rajeev Alur and David L. Dill. 'Automata For Modeling Real-Time Systems'. In: *Automata, Languages and Programming, 17th International Colloquium, ICALP90, Warwick University, England, UK, July 16-20, 1990, Proceedings.* Ed. by Mike Paterson. Vol. 443. Lecture Notes in Computer Science. Springer, 1990, pp. 322–335. DOI: `10.1007/BFb0032042`. URL: `https://doi.org/10.1007/BFb0032042`.

[AHK+20] Pranav Ashok, Vahid Hashemi, Jan Kretínský and Stefanie Mohr. 'DeepAbstract: Neural Network Abstraction for Accelerating Verification'. In: *CoRR* abs/2006.13735 (2020). arXiv: `2006.13735`. URL: `https://arxiv.org/abs/2006.13735`.

[AHL+00] R. Alur, T. A. Henzinger, G. Lafferriere and G. J. Pappas. 'Discrete abstractions of hybrid systems'. In: *Proceedings of the IEEE* 88.7 (2000), pp. 971–984.

[AJJ+20] Pranav Ashok, Mathias Jackermeier, Pushpak Jagtap, Jan Kretínský, Maximilian Weininger and Majid Zamani. 'dtControl: decision tree learning algorithms for controller representation'. In: *HSCC '20: 23rd ACM International Conference on Hybrid Systems: Computation and Control, Sydney, New South Wales, Australia, April 21-24, 2020.* Ed. by Aaron Ames, Sanjit A. Seshia and Jyotirmoy Deshmukh. ACM, 2020, 17:1–17:7. DOI: `10.1145/3365365.3382220`. URL: `https://doi.org/10.1145/3365365.3382220`.

[AKL+19] Pranav Ashok, Jan Kretínský, Kim Guldstrand Larsen, Adrien Le Coënt, Jakob Haahr Taankvist and Maximilian Weininger. 'SOS: Safe, Optimal and Small Strategies for Hybrid Markov Decision Processes'. In: *Quantitative Evaluation of Systems, 16th International Conference, QEST 2019, Glasgow, UK, September 10-12, 2019, Proceedings.* Ed. by David Parker and Verena Wolf. Vol. 11785. Lecture Notes in Computer Science. Springer, 2019, pp. 147–164. DOI: `10.1007/978-3-030-30281-8\_9`. URL: `https://doi.org/10.1007/978-3-030-30281-8%5C_9`.

[AKW19] Pranav Ashok, Jan Kretínský and Maximilian Weininger. 'PAC Statistical Model Checking for Markov Decision Processes and Stochastic Games'. In: *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I.* Ed. by Isil Dillig and Serdar Tasiran. Vol. 11561. Lecture Notes in Computer Science. Springer, 2019, pp. 497–519. DOI: `10.1007/978-3-030-25540-4\_29`. URL: `https://doi.org/10.1007/978-3-030-25540-4%5C_29`.

[And97] Henrik Reif Andersen. *An introduction to binary decision diagrams.* Technical University of Denmark. 1997. URL: `https://web.archive.org/web/20200501000000*/http://www.cs.utexas.edu/~isil/cs389L/bdd.pdf` (visited on 20/09/2020).

[ASS+00] Adnan Aziz, Kumud Sanwal, Vigyan Singhal and Robert K. Brayton. 'Model-checking continous-time Markov chains'. In: *ACM Trans. Comput. Log.* 1.1 (2000), pp. 162–170. DOI: `10.1145/343369.343402`. URL: `https://doi.org/10.1145/343369.343402`.

[Baa18] Sara Baase. *A gift of fire : social, legal, and ethical issues for computing technology.* NY, NY: Pearson, 2018. ISBN: 978-0134615271.

[BBS95] Andrew G. Barto, Steven J. Bradtke and Satinder P. Singh. 'Learning to Act Using Real-Time Dynamic Programming'. In: *Artif. Intell.* 72.1-2 (1995), pp. 81–138. DOI: `10.1016/0004-3702(94)00011-O`. URL: `https://doi.org/10.1016/0004-3702(94)00011-O`.

[BCC+14] Tomás Brázdil, Krishnendu Chatterjee, Martin Chmelik, Vojtech Forejt, Jan Kretínský, Marta Z. Kwiatkowska, David Parker and Mateusz Ujma. 'Verification of Markov Decision Processes Using Learning Algorithms'. In: *Automated Technology for Verification and Analysis - 12th International Symposium, ATVA 2014, Sydney, NSW, Australia, November 3-7, 2014, Proceedings.* Ed. by Franck Cassez and Jean-François Raskin. Vol. 8837. Lecture Notes in Computer Science. Springer, 2014, pp. 98–114. DOI: `10.1007/978-3-319-11936-6\_8`. URL: `https://doi.org/10.1007/978-3-319-11936-6%5C_8`.

[BCC+15] Tomás Brázdil, Krishnendu Chatterjee, Martin Chmelik, Andreas Fellner and Jan Kretínský. 'Counterexample Explanation by Learning Small Strategies in Markov Decision Processes'. In: *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July*

18-24, 2015, Proceedings, Part I. Ed. by Daniel Kroening and Corina S. Pasareanu. Vol. 9206. Lecture Notes in Computer Science. Springer, 2015, pp. 158–177. DOI: `10.1007/978-3-319-21690-4\_10`. URL: `https://doi.org/10.1007/978-3-319-21690-4%5C_10`.

[BCC+99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke and Yunshan Zhu. 'Symbolic Model Checking without BDDs'. In: *Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS '99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, March 22-28, 1999, Proceedings.* Ed. by Rance Cleaveland. Vol. 1579. Lecture Notes in Computer Science. Springer, 1999, pp. 193–207. DOI: `10.1007/3-540-49059-0\_14`. URL: `https://doi.org/10.1007/3-540-49059-0%5C_14`.

[BCD+07] Gerd Behrmann, Agnès Cougnard, Alexandre David, Emmanuel Fleury, Kim Guldstrand Larsen and Didier Lime. 'UPPAAL-Tiga: Time for Playing Games!' In: *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings.* Ed. by Werner Damm and Holger Hermanns. Vol. 4590. Lecture Notes in Computer Science. Springer, 2007, pp. 121–125. DOI: `10.1007/978-3-540-73368-3\_14`. URL: `https://doi.org/10.1007/978-3-540-73368-3%5C_14`.

[BCK+18] Tomás Brázdil, Krishnendu Chatterjee, Jan Kretínský and Viktor Toman. 'Strategy Representation by Decision Trees in Reactive Synthesis'. In: *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part I.* Ed. by Dirk Beyer and Marieke Huisman. Vol. 10805. Lecture Notes in Computer Science. Springer, 2018, pp. 385–407. DOI: `10.1007/978-3-319-89960-2\_21`. URL: `https://doi.org/10.1007/978-3-319-89960-2%5C_21`.

[BCM+90] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill and L. J. Hwang. 'Symbolic Model Checking: 10^{20} States and Beyond'. In: *Proceedings of the Fifth Annual Symposium on Logic in Computer Science (LICS '90), Philadelphia, Pennsylvania, USA, June 4-7, 1990.* IEEE Computer Society, 1990, pp. 428–439. DOI: `10.1109/LICS.1990.113767`. URL: `https://doi.org/10.1109/LICS.1990.113767`.

[BDG95] Craig Boutilier, Richard Dearden and Moisés Goldszmidt. 'Exploiting Structure in Policy Construction'. In: *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, IJCAI 95, Montréal Québec, Canada, August 20-25 1995, 2 Volumes.* Morgan Kaufmann, 1995, pp. 1104–1113. URL: `http://ijcai.org/Proceedings/95-2/Papers/012.pdf`.

[BDH99] Craig Boutilier, Thomas L. Dean and Steve Hanks. 'Decision-Theoretic Planning: Structural Assumptions and Computational Leverage'. In: *J. Artif. Intell. Res.* 11 (1999), pp. 1–94. DOI: `10.1613/jair.575`. URL: `https://doi.org/10.1613/jair.575`.

[Bel57] Richard Bellman. 'A Markovian Decision Process'. In: *Indiana Univ. Math. J.* 6 (4 1957), pp. 679–684. ISSN: 0022-2518.

[Ber05] Dimitri P. Bertsekas. *Dynamic programming and optimal control, 3rd Edition.* Athena Scientific, 2005. ISBN: 1886529264. URL: `https://www.worldcat.org/oclc/314894080`.

[BFG+97] R. Iris Bahar, Erica A. Frohm, Charles M. Gaona, Gary D. Hachtel, Enrico Macii, Abelardo Pardo and Fabio Somenzi. 'Algebraic Decision Diagrams and Their Applications'. In: *Formal Methods Syst. Des.* 10.2/3 (1997), pp. 171–206. DOI: `10.1023/A:1008699807402`. URL: `https://doi.org/10.1023/A:1008699807402`.

[BFO+84] Leo Breiman, J. H. Friedman, R. A. Olshen and C. J. Stone. *Classification and Regression Trees.* Wadsworth, 1984.

[BHH+11] Peter Buchholz, Ernst Moritz Hahn, Holger Hermanns and Lijun Zhang. 'Model Checking Algorithms for CTMDPs'. In: *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings.* Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Vol. 6806. Lecture Notes in Computer Science. Springer, 2011, pp. 225–242. DOI: `10.1007/978-3-642-22110-1\_19`. URL: `https://doi.org/10.1007/978-3-642-22110-1%5C_19`.

[BHH+15] Yuliya Butkova, Hassan Hatefi, Holger Hermanns and Jan Krcál. 'Optimal Continuous Time Markov Decisions'. In: *Automated Technology for Verification and Analysis - 13th International Symposium, ATVA 2015, Shanghai, China, October 12-15, 2015, Proceedings.* Ed. by Bernd Finkbeiner, Geguang Pu and Lijun Zhang. Vol. 9364. Lecture Notes in Computer Science. Springer, 2015, pp. 166–182. DOI: `10.1007/978-3-319-24953-7\_12`. URL: `https://doi.org/10.1007/978-3-319-24953-7%5C_12`.

[BHK+05] Christel Baier, Holger Hermanns, Joost-Pieter Katoen and Boudewijn R. Haverkort. 'Efficient computation of time-bounded reachability probabilities in uniform continuous-time Markov decision processes'. In: *Theor. Comput. Sci.* 345.1 (2005), pp. 2–26. DOI: `10.1016/j.tcs.2005.07.022`. URL: `https://doi.org/10.1016/j.tcs.2005.07.022`.

[BHK19] Christel Baier, Holger Hermanns and Joost-Pieter Katoen. 'The 10,000 Facets of MDP Model Checking'. In: *Computing and Software Science - State of the Art and Perspectives.* Ed. by Bernhard Steffen and Gerhard J. Woeginger. Vol. 10000. Lecture Notes in Computer Science. Springer, 2019, pp. 420–451. DOI: `10.1007/978-3-319-91908-9\_21`. URL: `https://doi.org/10.1007/978-3-319-91908-9%5C_21`.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of model checking.* MIT Press, 2008. ISBN: 978-0-262-02649-9.

[BKL+17] 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: *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I.* Ed. by Rupak Majumdar and Viktor Kuncak. Vol. 10426. Lecture Notes in Computer Science. Springer, 2017, pp. 160–180. DOI: `10.1007/978-3-319-63387-9\_8`. URL: `https://doi.org/10.1007/978-3-319-63387-9%5C_8`.

[BKN+19] Nikhil Balaji, Stefan Kiefer, Petr Novotný, Guillermo A. Pérez and Mahsa Shirmohammadi. 'On the Complexity of Value Iteration'. In: *46th International Colloquium on Automata, Languages, and Programming, ICALP 2019, July 9-12, 2019, Patras, Greece.* Ed. by Christel Baier, Ioannis Chatzigiannakis, Paola Flocchini and Stefano Leonardi. Vol. 132. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 102:1–102:15. DOI: `10.4230/LIPIcs.ICALP.2019.102`. URL: `https://doi.org/10.4230/LIPIcs.ICALP.2019.102`.

[Blo99] Hendrik Blockeel. 'Top-Down Induction of First Order Logical Decision Trees'. In: *AI Commun.* 12.1-2 (1999), pp. 119–120. URL: `http://content.iospress.com/articles/ai-communications/aic178`.

[Boc78] Gregor von Bochmann. 'Finite State Description of Communication Protocols'. In: *Comput. Networks* 2 (1978), pp. 361–372. DOI: `10.1016/0376-5075(78)90015-6`. URL: `https://doi.org/10.1016/0376-5075(78)90015-6`.

[BPW+12] Cameron Browne, Edward Jack Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez Liebana, Spyridon Samothrakis and Simon Colton. 'A Survey of Monte Carlo Tree Search Methods'. In: *IEEE Trans. Comput. Intell. AI Games* 4.1 (2012), pp. 1–43. DOI: `10.1109/TCIAIG.2012.2186810`. URL: `https://doi.org/10.1109/TCIAIG.2012.2186810`.

[Bry18] Randal E. Bryant. 'Binary Decision Diagrams'. In: *Handbook of Model Checking.* Ed. by Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith and Roderick Bloem. Cham: Springer International Publishing, 2018, pp. 191–217. ISBN: 978-3-319-10575-8. DOI: `10.1007/978-3-319-10575-8_7`. URL: `https://doi.org/10.1007/978-3-319-10575-8_7`.

[Bry86] Randal E. Bryant. 'Graph-Based Algorithms for Boolean Function Manipulation'. In: *IEEE Trans. Computers* 35.8 (1986), pp. 677–691. DOI: `10.1109/TC.1986.1676819`. URL: `https://doi.org/10.1109/TC.1986.1676819`.

[Bry92] Randal E. Bryant. 'Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams'. In: *ACM Comput. Surv.* 24.3 (1992), pp. 293–318. DOI: 10.1145/136035.136043. URL: https://doi.org/10.1145/136035.136043.

[Bry95] Randal E. Bryant. 'Binary decision diagrams and beyond: enabling technologies for formal verification'. In: *Proceedings of the 1995 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 1995, San Jose, California, USA, November 5-9, 1995*. Ed. by Richard L. Rudell. IEEE, 1995, pp. 236–243. DOI: 10.1109/ICCAD.1995.480018. URL: https://doi.org/10.1109/ICCAD.1995.480018.

[BS11] Peter Buchholz and Ingo Schulz. 'Numerical analysis of continuous time Markov decision processes over finite horizons'. In: *Comput. Oper. Res.* 38.3 (2011), pp. 651–659. DOI: 10.1016/j.cor.2010.08.011. URL: https://doi.org/10.1016/j.cor.2010.08.011.

[But20] Yuliya Butkova. 'Towards Efficient Analysis of Markov Automata'. PhD thesis. Saarland University, 2020.

[BW96] Beate Bollig and Ingo Wegener. 'Improving the Variable Ordering of OBDDs Is NP-Complete'. In: *IEEE Trans. Computers* 45.9 (1996), pp. 993–1002. DOI: 10.1109/12.537122. URL: https://doi.org/10.1109/12.537122.

[BYG17] Calin Belta, Boyan Yordanov and Ebru Aydin Gol. *Formal Methods for Discrete-Time Dynamical Systems*. Springer International Publishing, 2017. DOI: 10.1007/978-3-319-50763-7. URL: https://doi.org/10.1007/978-3-319-50763-7.

[BZ83] Daniel Brand and Pitro Zafiropulo. 'On Communicating Finite-State Machines'. In: *J. ACM* 30.2 (1983), pp. 323–342. DOI: 10.1145/322374.322380. URL: https://doi.org/10.1145/322374.322380.

[CDB07] Tom Croonenborghs, Kurt Driessens and Maurice Bruynooghe. 'Learning Relational Options for Inductive Transfer in Relational Reinforcement Learning'. In: *Inductive Logic Programming, 17th International Conference, ILP 2007, Corvallis, OR, USA, June 19-21, 2007, Revised Selected Papers*. Ed. by Hendrik Blockeel, Jan Ramon, Jude W. Shavlik and Prasad Tadepalli. Vol. 4894. Lecture Notes in Computer Science. Springer, 2007, pp. 88–97. DOI: 10.1007/978-3-540-78469-2_12. URL: https://doi.org/10.1007/978-3-540-78469-2%5C_12.

[CE81] Edmund M. Clarke and E. Allen Emerson. 'Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic'. In: *Logics of Programs, Workshop, Yorktown Heights, New York, USA, May 1981*. Ed. by Dexter Kozen. Vol. 131. Lecture Notes in Computer Science. Springer, 1981, pp. 52–71. DOI: 10.1007/BFb0025774. URL: https://doi.org/10.1007/BFb0025774.

[CGJ+00] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu and Helmut Veith. 'Counterexample-Guided Abstraction Refinement'. In: *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*. Ed. by E. Allen Emerson and A. Prasad Sistla. Vol. 1855. Lecture Notes in Computer Science. Springer, 2000, pp. 154–169. DOI: `10.1007/10722167\_15`. URL: `https://doi.org/10.1007/10722167%5C_15`.

[CH08] Krishnendu Chatterjee and Thomas A. Henzinger. 'Value Iteration'. In: *25 Years of Model Checking - History, Achievements, Perspectives*. Ed. by Orna Grumberg and Helmut Veith. Vol. 5000. Lecture Notes in Computer Science. Springer, 2008, pp. 107–138. DOI: `10.1007/978-3-540-69850-0\_7`. URL: `https://doi.org/10.1007/978-3-540-69850-0%5C_7`.

[CHV+18] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith and Roderick Bloem, eds. *Handbook of Model Checking*. Springer, 2018. ISBN: 978-3-319-10574-1. DOI: `10.1007/978-3-319-10575-8`. URL: `https://doi.org/10.1007/978-3-319-10575-8`.

[CKN+12] Edmund M. Clarke, William Klieber, Miloš Nováček and Paolo Zuliani. 'Model Checking and the State Explosion Problem'. In: *Tools for Practical Software Verification: LASER, International Summer School 2011, Elba Island, Italy, Revised Tutorial Lectures*. Ed. by Bertrand Meyer and Martin Nordio. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 1–30. ISBN: 978-3-642-35746-6. DOI: `10.1007/978-3-642-35746-6_1`. URL: `https://doi.org/10.1007/978-3-642-35746-6_1`.

[COM+19] Richard Cheng, Gábor Orosz, Richard M. Murray and Joel W. Burdick. 'End-to-End Safe Reinforcement Learning through Barrier Functions for Safety-Critical Continuous Control Tasks'. 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*. AAAI Press, 2019, pp. 3387–3395. DOI: `10.1609/aaai.v33i01.33013387`. URL: `https://doi.org/10.1609/aaai.v33i01.33013387`.

[Cou06] Rémi Coulom. 'Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search'. In: *Computers and Games, 5th International Conference, CG 2006, Turin, Italy, May 29-31, 2006. Revised Papers*. Ed. by H. Jaap van den Herik, Paolo Ciancarini and H. H. L. M. Donkers. Vol. 4630. Lecture Notes in Computer Science. Springer, 2006, pp. 72–83. DOI: `10.1007/978-3-540-75538-8\_7`. URL: `https://doi.org/10.1007/978-3-540-75538-8%5C_7`.

[CRT98] Alessandro Cimatti, Marco Roveri and Paolo Traverso. 'Automatic OBDD-Based Generation of Universal Plans in Non-Deterministic Domains'. In: *Proceedings of the Fifteenth National Conference on Artificial Intelligence* *and Tenth Innovative Applications of Artificial Intelligence Conference, AAAI 98, IAAI 98, July 26-30, 1998, Madison, Wisconsin, USA*. Ed. by Jack Mostow and Chuck Rich. AAAI Press / The MIT Press, 1998, pp. 875–881. URL: `http://www.aaai.org/Library/AAAI/1998/aaai98-124.php`.

[De 97] Luca De Alfaro. 'Formal verification of probabilistic systems'. PhD thesis. Stanford University, 1997.

[dEp63] F. d'Epenoux. 'A Probabilistic Production and Inventory Problem'. In: *Management Science* 10.1 (1963), pp. 98–108. DOI: `10.1287/mnsc.10.1.98`. eprint: `https://doi.org/10.1287/mnsc.10.1.98`. URL: `https://doi.org/10.1287/mnsc.10.1.98`.

[Der] Nachum Dershowitz. *Software Horror Stories*. `https://web.archive.org/web/20200527131419/http://www.cs.tau.ac.il/~nachumd/verify/horror.html`. [Online; accessed 08-September-2020].

[DGV+12] Christian Dehnert, Daniel Gebler, Michele Volpato and David N. Jansen. 'On Abstraction of Probabilistic Systems'. In: *Stochastic Model Checking. Rigorous Dependability Analysis Using Model Checking Techniques for Stochastic Systems - International Autumn School, ROCKS 2012, Vahrn, Italy, October 22-26, 2012, Advanced Lectures*. Ed. by Anne Remke and Mariëlle Stoelinga. Vol. 8453. Lecture Notes in Computer Science. Springer, 2012, pp. 87–116. DOI: `10.1007/978-3-662-45489-3\_4`. URL: `https://doi.org/10.1007/978-3-662-45489-3%5C_4`.

[Dij72] Edsger W. Dijkstra. 'The Humble Programmer'. In: *Commun. ACM* 15.10 (1972), pp. 859–866. DOI: `10.1145/355604.361591`. URL: `https://doi.org/10.1145/355604.361591`.

[DIL+09] Giuseppe Della Penna, Benedetto Intrigila, Nadia Lauri and Daniele Magazzeni. 'Fast and Compact Encoding of Numerical Controllers Using OBDDs'. In: *Informatics in Control, Automation and Robotics: Selcted Papers from the International Conference on Informatics in Control, Automation and Robotics 2008*. Ed. by Juan Andrade Cetto, Jean-Louis Ferrier and Joaquim Filipe. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 75–87. ISBN: 978-3-642-00271-7. DOI: `10.1007/978-3-642-00271-7_5`. URL: `https://doi.org/10.1007/978-3-642-00271-7_5`.

[DJJ+02] Pedro R. D'Argenio, Bertrand Jeannet, Henrik Ejersbo Jensen and Kim Guldstrand Larsen. 'Reduction and Refinement Strategies for Probabilistic Analysis'. In: *Process Algebra and Probabilistic Methods, Performance Modeling and Verification, Second Joint International Workshop PAPM-PROBMIV 2002, Copenhagen, Denmark, July 25-26, 2002, Proceedings*. Ed. by Holger Hermanns and Roberto Segala. Vol. 2399. Lecture Notes in Computer Science. Springer, 2002, pp. 57–76. DOI: `10.1007/3-540-45605-8\_5`. URL: `https://doi.org/10.1007/3-540-45605-8%5C_5`.

[DJL+15] Alexandre David, Peter Gjøl Jensen, Kim Guldstrand Larsen, Marius Mikucionis and Jakob Haahr Taankvist. ‘Uppaal Stratego’. In: *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings.* Ed. by Christel Baier and Cesare Tinelli. Vol. 9035. Lecture Notes in Computer Science. Springer, 2015, pp. 206–211. DOI: `10.1007/978-3-662-46681-0\_16`. URL: `https://doi.org/10.1007/978-3-662-46681-0%5C_16`.

[DN04] Pedro R. D’Argenio and Peter Niebert. ‘Partial Order Reduction on Concurrent Probabilistic Programs’. In: *1st International Conference on Quantitative Evaluation of Systems (QEST 2004), 27-30 September 2004, Enschede, The Netherlands.* IEEE Computer Society, 2004, pp. 240–249. DOI: `10.1109/QEST.2004.1348038`. URL: `https://doi.org/10.1109/QEST.2004.1348038`.

[DSB17] Derek Doran, Sarah Schulz and Tarek R. Besold. ‘What Does Explainable AI Really Mean? A New Conceptualization of Perspectives’. In: *Proceedings of the First International Workshop on Comprehensibility and Explanation in AI and ML 2017 co-located with 16th International Conference of the Italian Association for Artificial Intelligence (AI*IA 2017), Bari, Italy, November 16th and 17th, 2017.* Ed. by Tarek R. Besold and Oliver Kutz. Vol. 2071. CEUR Workshop Proceedings. CEUR-WS.org, 2017. URL: `http://ceur-ws.org/Vol-2071/CExAIIA%5C_2017%5C_paper%5C_2.pdf`.

[DT98] Conrado Daws and Stavros Tripakis. ‘Model Checking of Real-Time Reachability Properties Using Abstractions’. In: *Tools and Algorithms for Construction and Analysis of Systems, 4th International Conference, TACAS ’98, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS’98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings.* Ed. by Bernhard Steffen. Vol. 1384. Lecture Notes in Computer Science. Springer, 1998, pp. 313–329. DOI: `10.1007/BFb0054180`. URL: `https://doi.org/10.1007/BFb0054180`.

[EHZ10] Christian Eisentraut, Holger Hermanns and Lijun Zhang. ‘On Probabilistic Automata in Continuous Time’. In: *Proceedings of the 25th Annual IEEE Symposium on Logic in Computer Science, LICS 2010, 11-14 July 2010, Edinburgh, United Kingdom.* IEEE Computer Society, 2010, pp. 342–351. DOI: `10.1109/LICS.2010.41`. URL: `https://doi.org/10.1109/LICS.2010.41`.

[Fin16] Bernd Finkbeiner. ‘Synthesis of Reactive Systems’. In: *Dependable Software Systems Engineering.* Ed. by Javier Esparza, Orna Grumberg and Salomon Sickert. Vol. 45. NATO Science for Peace and Security Series - D: Information and Communication Security. IOS Press, 2016, pp. 72–98.

DOI: 10.3233/978-1-61499-627-9-72. URL: https://doi.org/10.3233/978-1-61499-627-9-72.

[FRS+11] John Fearnley, Markus N. Rabe, Sven Schewe and Lijun Zhang. 'Efficient Approximation of Optimal Control for Continuous-Time Markov Games'. In: *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2011, December 12-14, 2011, Mumbai, India*. Ed. by Supratik Chakraborty and Amit Kumar. Vol. 13. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2011, pp. 399–410. DOI: 10.4230/LIPIcs.FSTTCS.2011.399. URL: https://doi.org/10.4230/LIPIcs.FSTTCS.2011.399.

[FRS+16] John Fearnley, Markus N. Rabe, Sven Schewe and Lijun Zhang. 'Efficient approximation of optimal control for continuous-time Markov games'. In: *Inf. Comput.* 247 (2016), pp. 106–129. DOI: 10.1016/j.ic.2015.12.002. URL: https://doi.org/10.1016/j.ic.2015.12.002.

[FZW+17] Pedro Fonseca, Kaiyuan Zhang, Xi Wang and Arvind Krishnamurthy. 'An Empirical Study on the Correctness of Formally Verified Distributed Systems'. In: *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*. Ed. by Gustavo Alonso, Ricardo Bianchini and Marko Vukolic. ACM, 2017, pp. 328–343. DOI: 10.1145/3064176.3064183. URL: https://doi.org/10.1145/3064176.3064183.

[GCH+18] Randy Goebel, Ajay Chander, Katharina Holzinger, Freddy Lécué, Zeynep Akata, Simone Stumpf, Peter Kieseberg and Andreas Holzinger. 'Explainable AI: The New 42?' In: *Machine Learning and Knowledge Extraction - Second IFIP TC 5, TC 8/WG 8.4, 8.9, TC 12/WG 12.9 International Cross-Domain Conference, CD-MAKE 2018, Hamburg, Germany, August 27-30, 2018, Proceedings*. Ed. by Andreas Holzinger, Peter Kieseberg, A Min Tjoa and Edgar R. Weippl. Vol. 11015. Lecture Notes in Computer Science. Springer, 2018, pp. 295–303. DOI: 10.1007/978-3-319-99740-7_21. URL: https://doi.org/10.1007/978-3-319-99740-7%5C_21.

[GF15] Javier García and Fernando Fernández. 'A comprehensive survey on safe reinforcement learning'. In: *J. Mach. Learn. Res.* 16 (2015), pp. 1437–1480. URL: http://dl.acm.org/citation.cfm?id=2886795.

[GGL03] Sanjay Ghemawat, Howard Gobioff and Shun-Tak Leung. 'The Google file system'. In: *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003*. Ed. by Michael L. Scott and Larry L. Peterson. ACM, 2003, pp. 29–43. DOI: 10.1145/945445.945450. URL: https://doi.org/10.1145/945445.945450.

[GHP+06] Xianping Guo, Onésimo Hernández-Lerma, Tomás Prieto-Rumeau, Xi-Ren Cao, Junyu Zhang, Qiying Hu, Mark E Lewis and Ricardo Vélez. 'A survey of recent results on continuous-time Markov decision processes'. In: *Top* 14.2 (2006), pp. 177–261.

[Gir12] Antoine Girard. 'Low-Complexity Quantized Switching Controllers using Approximate Bisimulation'. In: *CoRR* abs/1209.4576 (2012). arXiv: `1209.4576`. URL: `http://arxiv.org/abs/1209.4576`.

[God90] Patrice Godefroid. 'Using Partial Orders to Improve Automatic Verification Methods'. In: *Computer-Aided Verification, Proceedings of a DIMACS Workshop 1990, New Brunswick, New Jersey, USA, June 18-21, 1990.* Ed. by Edmund M. Clarke and Robert P. Kurshan. Vol. 3. DIMACS Series in Discrete Mathematics and Theoretical Computer Science. DIMACS/AMS, 1990, pp. 321–340. DOI: `10.1090/dimacs/003/21`. URL: `https://doi.org/10.1090/dimacs/003/21`.

[GP07] Antoine Girard and George J. Pappas. 'Approximation Metrics for Discrete and Continuous Systems'. In: *IEEE Trans. Autom. Control.* 52.5 (2007), pp. 782–798. DOI: `10.1109/TAC.2007.895849`. URL: `https://doi.org/10.1109/TAC.2007.895849`.

[GPS+80] Dov M. Gabbay, Amir Pnueli, Saharon Shelah and Jonathan Stavi. 'On the Temporal Basis of Fairness'. In: *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages, Las Vegas, Nevada, USA, January 1980.* Ed. by Paul W. Abrahams, Richard J. Lipton and Stephen R. Bourne. ACM Press, 1980, pp. 163–173. DOI: `10.1145/567446.567462`. URL: `https://doi.org/10.1145/567446.567462`.

[GPS17] Sumit Gulwani, Oleksandr Polozov and Rishabh Singh. 'Program Synthesis'. In: *Found. Trends Program. Lang.* 4.1-2 (2017), pp. 1–119. DOI: `10.1561/2500000010`. URL: `https://doi.org/10.1561/2500000010`.

[GV08] Orna Grumberg and Helmut Veith, eds. *25 Years of Model Checking - History, Achievements, Perspectives.* Vol. 5000. Lecture Notes in Computer Science. Springer, 2008. ISBN: 978-3-540-69849-4. DOI: `10.1007/978-3-540-69850-0`. URL: `https://doi.org/10.1007/978-3-540-69850-0`.

[GW05] Peter Geibel and Fritz Wysotzki. 'Risk-Sensitive Reinforcement Learning Applied to Control under Constraints'. In: *J. Artif. Intell. Res.* 24 (2005), pp. 81–108. DOI: `10.1613/jair.1666`. URL: `https://doi.org/10.1613/jair.1666`.

[HAK20] Mohammadhosein Hasanbeig, Alessandro Abate and Daniel Kroening. 'Cautious Reinforcement Learning with Logical Constraints'. In: *Proceedings of the 19th International Conference on Autonomous Agents and Multiagent Systems, AAMAS '20, Auckland, New Zealand, May 9-13, 2020.* Ed. by Amal El Fallah Seghrouchni, Gita Sukthankar, Bo An and Neil Yorke-Smith. International Foundation for Autonomous Agents and Multiagent

Systems, 2020, pp. 483–491. URL: https://dl.acm.org/doi/abs/10.5555/3398761.3398821.

[HHK00] Boudewijn R. Haverkort, Holger Hermanns and Joost-Pieter Katoen. 'On the Use of Model Checking Techniques for Dependability Evaluation'. In: *19th IEEE Symposium on Reliable Distributed Systems, SRDS'00, Nürnberg, Germany, October 16-18, 2000, Proceedings*. IEEE Computer Society, 2000, pp. 228–237. DOI: 10.1109/RELDI.2000.885410. URL: https://doi.org/10.1109/RELDI.2000.885410.

[HHW+10] Ernst Moritz Hahn, Holger Hermanns, Björn Wachter and Lijun Zhang. 'PASS: Abstraction Refinement for Infinite Probabilistic Models'. In: *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*. Ed. by Javier Esparza and Rupak Majumdar. Vol. 6015. Lecture Notes in Computer Science. Springer, 2010, pp. 353–357. DOI: 10.1007/978-3-642-12002-2_30. URL: https://doi.org/10.1007/978-3-642-12002-2%5C_30.

[HJK+20] Christian Hensel, Sebastian Junges, Joost-Pieter Katoen, Tim Quatmann and Matthias Volk. *The Probabilistic Model Checker Storm*. 2020. arXiv: 2002.07080 [cs.SE].

[HK20] Arnd Hartmanns and Benjamin Lucien Kaminski. 'Optimistic Value Iteration'. In: *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II*. Ed. by Shuvendu K. Lahiri and Chao Wang. Vol. 12225. Lecture Notes in Computer Science. Springer, 2020, pp. 488–511. DOI: 10.1007/978-3-030-53291-8_26. URL: https://doi.org/10.1007/978-3-030-53291-8%5C_26.

[HM14] Serge Haddad and Benjamin Monmege. 'Reachability in MDPs: Refining Convergence of Value Iteration'. In: *Reachability Problems - 8th International Workshop, RP 2014, Oxford, UK, September 22-24, 2014. Proceedings*. Ed. by Joël Ouaknine, Igor Potapov and James Worrell. Vol. 8762. Lecture Notes in Computer Science. Springer, 2014, pp. 125–137. DOI: 10.1007/978-3-319-11439-2_10. URL: https://doi.org/10.1007/978-3-319-11439-2%5C_10.

[HM18] Serge Haddad and Benjamin Monmege. 'Interval iteration algorithm for MDPs and IMDPs'. In: *Theor. Comput. Sci.* 735 (2018), pp. 111–131. DOI: 10.1016/j.tcs.2016.12.003. URL: https://doi.org/10.1016/j.tcs.2016.12.003.

[Hoa69] C. A. R. Hoare. 'An Axiomatic Basis for Computer Programming'. In: *Commun. ACM* 12.10 (1969), pp. 576–580. DOI: 10.1145/363235.363259. URL: https://doi.org/10.1145/363235.363259.

[How60] Ronald A Howard. 'Dynamic programming and markov processes.' In: (1960).

[HSH+99] Jesse Hoey, Robert St-Aubin, Alan J. Hu and Craig Boutilier. 'SPUDD: Stochastic Planning using Decision Diagrams'. In: *UAI '99: Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence, Stockholm, Sweden, July 30 - August 1, 1999.* Ed. by Kathryn B. Laskey and Henri Prade. Morgan Kaufmann, 1999, pp. 279–288. URL: `https://dslpitt.org/uai/displayArticleDetails.jsp?mmnu=1%5C&smnu=2%5C&article%5C_id=178%5C&proceeding%5C_id=15`.

[Jac20] Mathias Jackermeier. 'dtControl: Decision Tree Learning for Explainable Controller Representation'. Bachelorarbeit. Technische Universität München, 2020.

[Jan98] Cezary Z. Janikow. 'Fuzzy decision trees: issues and methods'. In: *IEEE Trans. Syst. Man Cybern. Part B* 28.1 (1998), pp. 1–14. DOI: `10.1109/3477.658573`. URL: `https://doi.org/10.1109/3477.658573`.

[JDT10] Manuel Mazo Jr., Anna Davitian and Paulo Tabuada. 'PESSOA: A Tool for Embedded Controller Synthesis'. In: *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings.* Ed. by Tayssir Touili, Byron Cook and Paul B. Jackson. Vol. 6174. Lecture Notes in Computer Science. Springer, 2010, pp. 566–569. DOI: `10.1007/978-3-642-14295-6\_49`. URL: `https://doi.org/10.1007/978-3-642-14295-6%5C_49`.

[JKJ+20] Nils Jansen, Bettina Könighofer, Sebastian Junges, Alex Serban and Roderick Bloem. 'Safe Reinforcement Learning Using Probabilistic Shields (Invited Paper)'. In: *31st International Conference on Concurrency Theory, CONCUR 2020, September 1-4, 2020, Vienna, Austria (Virtual Conference).* Ed. by Igor Konnov and Laura Kovács. Vol. 171. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 3:1–3:16. DOI: `10.4230/LIPIcs.CONCUR.2020.3`. URL: `https://doi.org/10.4230/LIPIcs.CONCUR.2020.3`.

[JZ17] Pushpak Jagtap and Majid Zamani. 'QUEST: A Tool for State-Space Quantization-Free Synthesis of Symbolic Controllers'. In: *Quantitative Evaluation of Systems - 14th International Conference, QEST 2017, Berlin, Germany, September 5-7, 2017, Proceedings.* Ed. by Nathalie Bertrand and Luca Bortolussi. Vol. 10503. Lecture Notes in Computer Science. Springer, 2017, pp. 309–313. DOI: `10.1007/978-3-319-66335-7\_21`. URL: `https://doi.org/10.1007/978-3-319-66335-7%5C_21`.

[Kas07] Hisashi Kashima. 'Risk-Sensitive Learning via Minimization of Empirical Conditional Value-at-Risk'. In: *IEICE Trans. Inf. Syst.* 90-D.12 (2007), pp. 2043–2052. DOI: `10.1093/ietisy/e90-d.12.2043`. URL: `https://doi.org/10.1093/ietisy/e90-d.12.2043`.

[KKK+18] Edon Kelmendi, Julia Krämer, Jan Kretínský and Maximilian Weininger. 'Value Iteration for Simple Stochastic Games: Stopping Criterion and Learning Algorithm'. In: *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I.* Ed. by Hana Chockler and Georg Weissenbacher. Vol. 10981. Lecture Notes in Computer Science. Springer, 2018, pp. 623–642. DOI: `10.1007/978-3-319-96145-3\_36`. URL: `https://doi.org/10.1007/978-3-319-96145-3%5C_36`.

[KKN+10] Mark Kattenbelt, Marta Z. Kwiatkowska, Gethin Norman and David Parker. 'A game-based abstraction-refinement framework for Markov decision processes'. In: *Formal Methods Syst. Des.* 36.3 (2010), pp. 246–280. DOI: `10.1007/s10703-010-0097-6`. URL: `https://doi.org/10.1007/s10703-010-0097-6`.

[KM18] Jan Kretínský and Tobias Meggendorfer. 'Conditional Value-at-Risk for Reachability and Mean Payoff in Markov Decision Processes'. In: *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018.* Ed. by Anuj Dawar and Erich Grädel. ACM, 2018, pp. 609–618. DOI: `10.1145/3209108.3209176`. URL: `https://doi.org/10.1145/3209108.3209176`.

[KM19] Jan Kretínský and Tobias Meggendorfer. 'Of Cores: A Partial-Exploration Framework for Markov Decision Processes'. In: *30th International Conference on Concurrency Theory, CONCUR 2019, August 27-30, 2019, Amsterdam, the Netherlands.* Ed. by Wan J. Fokkink and Rob van Glabbeek. Vol. 140. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 5:1–5:17. DOI: `10.4230/LIPIcs.CONCUR.2019.5`. URL: `https://doi.org/10.4230/LIPIcs.CONCUR.2019.5`.

[KNP11] M. Kwiatkowska, G. Norman and D. Parker. 'PRISM 4.0: Verification of Probabilistic Real-time Systems'. In: *Proc. 23rd International Conference on Computer Aided Verification (CAV'11).* Ed. by G. Gopalakrishnan and S. Qadeer. Vol. 6806. LNCS. Springer, 2011, pp. 585–591.

[LDK95] Michael L. Littman, Thomas L. Dean and Leslie Pack Kaelbling. 'On the Complexity of Solving Markov Decision Problems'. In: *UAI '95: Proceedings of the Eleventh Annual Conference on Uncertainty in Artificial Intelligence, Montreal, Quebec, Canada, August 18-20, 1995.* Ed. by Philippe Besnard and Steve Hanks. Morgan Kaufmann, 1995, pp. 394–402. URL: `https://dslpitt.org/uai/displayArticleDetails.jsp?mmnu=1%5C&smnu=2%5C&article%5C_id=457%5C&proceeding%5C_id=11`.

[Lef81] Claude Lefévre. 'Optimal control of a birth and death epidemic process'. In: *Operations Research* 29.5 (1981), pp. 971–982.

[LL14] Daniel T Larose and Chantal D Larose. *Discovering knowledge in data: an introduction to data mining.* Vol. 4. John Wiley & Sons, 2014.

[LPY95] Kim Guldstrand Larsen, Paul Pettersson and Wang Yi. 'Model-Checking for Real-Time Systems'. In: *Fundamentals of Computation Theory, 10th International Symposium, FCT '95, Dresden, Germany, August 22-25, 1995, Proceedings*. Ed. by Horst Reichel. Vol. 965. Lecture Notes in Computer Science. Springer, 1995, pp. 62–88. DOI: 10.1007/3-540-60249-6_41. URL: https://doi.org/10.1007/3-540-60249-6%5C_41.

[LS91] Kim Guldstrand Larsen and Arne Skou. 'Bisimulation through Probabilistic Testing'. In: *Inf. Comput.* 94.1 (1991), pp. 1–28. DOI: 10.1016/0890-5401(91)90030-6. URL: https://doi.org/10.1016/0890-5401(91)90030-6.

[LT93] N. G. Leveson and C. S. Turner. 'An investigation of the Therac-25 accidents'. In: *Computer* 26.7 (July 1993), pp. 18–41. ISSN: 1558-0814. DOI: 10.1109/MC.1993.274940.

[Mil68] Bruce L. Miller. 'Finite State Continuous Time Markov Decision Processes with a Finite Planning Horizon'. In: *SIAM Journal on Control* 6.2 (1968), pp. 266–280. DOI: 10.1137/0306020. eprint: https://doi.org/10.1137/0306020. URL: https://doi.org/10.1137/0306020.

[Mil89] Robin Milner. *Communication and concurrency.* PHI Series in computer science. Prentice Hall, 1989. ISBN: 978-0-13-115007-2.

[Min12] Shin-ichi Minato. *Binary decision diagrams and applications for VLSI CAD.* Vol. 342. Springer Science & Business Media, 2012.

[Min93] Shin-ichi Minato. 'Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems'. In: *Proceedings of the 30th International Design Automation Conference.* DAC '93. Dallas, Texas, USA: Association for Computing Machinery, 1993, pp. 272–277. ISBN: 0897915771. DOI: 10.1145/157485.164890. URL: https://doi.org/10.1145/157485.164890.

[MKS+93] Sreerama K. Murthy, Simon Kasif, Steven Salzberg and Richard Beigel. 'OC1: A Randomized Induction of Oblique Decision Trees'. In: *Proceedings of the 11th National Conference on Artificial Intelligence. Washington, DC, USA, July 11-15, 1993.* Ed. by Richard Fikes and Wendy G. Lehnert. AAAI Press / The MIT Press, 1993, pp. 322–327. URL: http://www.aaai.org/Library/AAAI/1993/aaai93-049.php.

[MLG05] H. Brendan McMahan, Maxim Likhachev and Geoffrey J. Gordon. 'Bounded real-time dynamic programming: RTDP with monotone upper bounds and performance guarantees'. In: *Machine Learning, Proceedings of the Twenty-Second International Conference (ICML 2005), Bonn, Germany, August 7-11, 2005.* Ed. by Luc De Raedt and Stefan Wrobel. Vol. 119. ACM International Conference Proceeding Series. ACM, 2005, pp. 569–576. DOI: 10.1145/1102351.1102423. URL: https://doi.org/10.1145/1102351.1102423.

[MSS20] Rupak Majumdar, Mahmoud Salamati and Sadegh Soudjani. 'On Decidability of Time-Bounded Reachability in CTMDPs'. In: *47th International Colloquium on Automata, Languages, and Programming, ICALP 2020, July 8-11, 2020, Saarbrücken, Germany (Virtual Conference).* Ed. by Artur Czumaj, Anuj Dawar and Emanuela Merelli. Vol. 168. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 133:1–133:19. DOI: `10.4230/LIPIcs.ICALP.2020.133`. URL: `https://doi.org/10.4230/LIPIcs.ICALP.2020.133`.

[MYA12] Min Wu, A. Yamashita and H. Asama. 'Rule abstraction and transfer in reinforcement learning by decision tree'. In: *2012 IEEE/SICE International Symposium on System Integration (SII).* 2012, pp. 529–534.

[NE05] Arnab Nilim and Laurent El Ghaoui. 'Robust Control of Markov Decision Processes with Uncertain Transition Matrices'. In: *Operations Research* 53.5 (2005), pp. 780–798. DOI: `10.1287/opre.1050.0216`. eprint: `https://doi.org/10.1287/opre.1050.0216`. URL: `https://doi.org/10.1287/opre.1050.0216`.

[Neu10] Martin R. Neuhäußer. 'Model checking nondeterministic and randomly timed systems'. PhD thesis. RWTH Aachen University, 2010. ISBN: 978-90-365-2975-4. URL: `http://darwin.bth.rwth-aachen.de/opus3/volltexte/2010/3136/`.

[NZ10] Martin R. Neuhäußer and Lijun Zhang. 'Time-Bounded Reachability Probabilities in Continuous-Time Markov Decision Processes'. In: *QEST 2010, Seventh International Conference on the Quantitative Evaluation of Systems, Williamsburg, Virginia, USA, 15-18 September 2010.* IEEE Computer Society, 2010, pp. 209–218. DOI: `10.1109/QEST.2010.47`. URL: `https://doi.org/10.1109/QEST.2010.47`.

[Par81] David Michael Ritchie Park. 'Concurrency and Automata on Infinite Sequences'. In: *Theoretical Computer Science, 5th GI-Conference, Karlsruhe, Germany, March 23-25, 1981, Proceedings.* Ed. by Peter Deussen. Vol. 104. Lecture Notes in Computer Science. Springer, 1981, pp. 167–183. DOI: `10.1007/BFb0017309`. URL: `https://doi.org/10.1007/BFb0017309`.

[Pel93] Doron A. Peled. 'All from One, One for All: on Model Checking Using Representatives'. In: *Computer Aided Verification, 5th International Conference, CAV '93, Elounda, Greece, June 28 - July 1, 1993, Proceedings.* Ed. by Costas Courcoubetis. Vol. 697. Lecture Notes in Computer Science. Springer, 1993, pp. 409–423. DOI: `10.1007/3-540-56922-7\_34`. URL: `https://doi.org/10.1007/3-540-56922-7%5C_34`.

[PH+01] Larry D Pyeatt, Adele E Howe et al. 'Decision tree function approximation in reinforcement learning'. In: *Proceedings of the third international symposium on adaptive systems: evolutionary computation and probabilistic graphical models.* Vol. 2. 1/2. Cuba. 2001, pp. 70–77.

[Pnu77] Amir Pnueli. 'The Temporal Logic of Programs'. In: *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977.* IEEE Computer Society, 1977, pp. 46–57. DOI: 10.1109/SFCS.1977.32. URL: https://doi.org/10.1109/SFCS.1977.32.

[Pnu81] Amir Pnueli. 'The Temporal Semantics of Concurrent Programs'. In: *Theor. Comput. Sci.* 13 (1981), pp. 45–60. DOI: 10.1016/0304-3975(81)90110-9. URL: https://doi.org/10.1016/0304-3975(81)90110-9.

[PT87] Christos H. Papadimitriou and John N. Tsitsiklis. 'The Complexity of Markov Decision Processes'. In: *Math. Oper. Res.* 12.3 (1987), pp. 441–450. DOI: 10.1287/moor.12.3.441. URL: https://doi.org/10.1287/moor.12.3.441.

[PTH+20] Kittiphon Phalakarn, Toru Takisaka, Thomas Haas and Ichiro Hasuo. 'Widest Paths and Global Propagation in Bounded Value Iteration for Stochastic Games'. In: *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II.* Ed. by Shuvendu K. Lahiri and Chao Wang. Vol. 12225. Lecture Notes in Computer Science. Springer, 2020, pp. 349–371. DOI: 10.1007/978-3-030-53291-8_19. URL: https://doi.org/10.1007/978-3-030-53291-8%5C_19.

[Put94] Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming.* Wiley Series in Probability and Statistics. Wiley, 1994. ISBN: 978-0-47161977-2. DOI: 10.1002/9780470316887. URL: https://doi.org/10.1002/9780470316887.

[Pye03] Larry D. Pyeatt. 'Reinforcement Learning with Decision Trees'. In: *The 21st IASTED International Multi-Conference on Applied Informatics (AI 2003), February 10-13, 2003, Innsbruck, Austria.* Ed. by M. H. Hamza. IASTED/ACTA Press, 2003, pp. 26–31.

[QK18] Tim Quatmann and Joost-Pieter Katoen. 'Sound Value Iteration'. In: *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I.* Ed. by Hana Chockler and Georg Weissenbacher. Vol. 10981. Lecture Notes in Computer Science. Springer, 2018, pp. 643–661. DOI: 10.1007/978-3-319-96145-3_37. URL: https://doi.org/10.1007/978-3-319-96145-3%5C_37.

[QQP01] Qinru Qiu, Qing Qu and Massoud Pedram. 'Stochastic modeling of a power-managed system-construction andoptimization'. In: *IEEE Trans. on CAD of Integrated Circuits and Systems* 20.10 (2001), pp. 1200–1217. DOI: 10.1109/43.952737. URL: https://doi.org/10.1109/43.952737.

[QS82] Jean-Pierre Queille and Joseph Sifakis. 'Specification and verification of concurrent systems in CESAR'. In: *International Symposium on Programming, 5th Colloquium, Torino, Italy, April 6-8, 1982, Proceedings.* Ed. by Mariangiola Dezani-Ciancaglini and Ugo Montanari. Vol. 137. Lecture Notes in Computer Science. Springer, 1982, pp. 337–351. DOI: `10.1007/3-540-11494-7\_22`. URL: `https://doi.org/10.1007/3-540-11494-7%5C_22`.

[Qui86] J. Ross Quinlan. 'Induction of Decision Trees'. In: *Mach. Learn.* 1.1 (1986), pp. 81–106.

[Qui93] J. Ross Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufmann, 1993.

[Ros19] Sheldon M. Ross. *Introduction to Probability Models (Twelfth Edition).* Academic press, 2019. ISBN: "978-0-12-814346-9". DOI: `"https://doi.org/10.1016/B978-0-12-814346-9.00009-3"`. URL: `http://www.sciencedirect.com/science/article/pii/B9780128143469000093`.

[RS11] Markus N. Rabe and Sven Schewe. 'Finite optimal control for time-bounded reachability in CTMDPs and continuous-time Markov games'. In: *Acta Informatica* 48.5-6 (2011), pp. 291–315. DOI: `10.1007/s00236-011-0140-0`. URL: `https://doi.org/10.1007/s00236-011-0140-0`.

[RTJ+19] Aaron M. Roth, Nicholay Topin, Pooyan Jamshidi and Manuela Veloso. 'Conservative Q-Improvement: Reinforcement Learning for an Interpretable Decision-Tree Policy'. In: *CoRR* abs/1907.01180 (2019). arXiv: `1907.01180`. URL: `http://arxiv.org/abs/1907.01180`.

[Rud93] Richard Rudell. 'Dynamic variable ordering for ordered binary decision diagrams'. In: *Proceedings of the 1993 IEEE/ACM International Conference on Computer-Aided Design, 1993, Santa Clara, California, USA, November 7-11, 1993.* Ed. by Michael R. Lightner and Jochen A. G. Jess. IEEE, 1993, pp. 42–47. DOI: `10.1109/ICCAD.1993.580029`. URL: `https://doi.org/10.1109/ICCAD.1993.580029`.

[RZ16] Matthias Rungger and Majid Zamani. 'SCOTS: A Tool for the Synthesis of Symbolic Controllers'. In: *Proceedings of the 19th International Conference on Hybrid Systems: Computation and Control, HSCC 2016, Vienna, Austria, April 12-14, 2016.* Ed. by Alessandro Abate and Georgios E. Fainekos. ACM, 2016, pp. 99–104. DOI: `10.1145/2883817.2883834`. URL: `https://doi.org/10.1145/2883817.2883834`.

[SB98] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning - an introduction.* Adaptive computation and machine learning. MIT Press, 1998. ISBN: 978-0-262-19398-6. URL: `https://www.worldcat.org/oclc/37293240`.

[Sen09] Linn I Sennott. *Stochastic dynamic programming and the control of queueing systems.* Vol. 504. John Wiley & Sons, 2009.

[SHB00] Robert St-Aubin, Jesse Hoey and Craig Boutilier. 'APRICODD: Approximate Policy Construction Using Decision Diagrams'. In: *Advances in Neural Information Processing Systems 13, Papers from Neural Information Processing Systems (NIPS) 2000, Denver, CO, USA*. Ed. by Todd K. Leen, Thomas G. Dietterich and Volker Tresp. MIT Press, 2000, pp. 1089–1095. URL: `http://papers.nips.cc/paper/1840-apricodd-approximate-policy-construction-using-decision-diagrams`.

[SHM+16] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Vedavyas Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy P. Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel and Demis Hassabis. 'Mastering the game of Go with deep neural networks and tree search'. In: *Nat.* 529.7587 (2016), pp. 484–489. DOI: `10.1038/nature16961`. URL: `https://doi.org/10.1038/nature16961`.

[SL95] Roberto Segala and Nancy A. Lynch. 'Probabilistic Simulations for Probabilistic Processes'. In: *Nord. J. Comput.* 2.2 (1995), pp. 250–273.

[Tab09] Paulo Tabuada. *Verification and Control of Hybrid Systems - A Symbolic Approach.* Springer, 2009. ISBN: 978-1-4419-0223-8. URL: `http://www.springer.com/mathematics/applications/book/978-1-4419-0223-8`.

[TF06] John Tromp and Gunnar Farnebäck. 'Combinatorics of Go'. In: *Computers and Games, 5th International Conference, CG 2006, Turin, Italy, May 29-31, 2006. Revised Papers.* Ed. by H. Jaap van den Herik, Paolo Ciancarini and H. H. L. M. Donkers. Vol. 4630. Lecture Notes in Computer Science. Springer, 2006, pp. 84–99. DOI: `10.1007/978-3-540-75538-8\_8`. URL: `https://doi.org/10.1007/978-3-540-75538-8%5C_8`.

[Val89] Antti Valmari. 'Stubborn sets for reduced state space generation'. In: *Advances in Petri Nets 1990 [10th International Conference on Applications and Theory of Petri Nets, Bonn, Germany, June 1989, Proceedings].* Ed. by Grzegorz Rozenberg. Vol. 483. Lecture Notes in Computer Science. Springer, 1989, pp. 491–515. DOI: `10.1007/3-540-53863-1\_36`. URL: `https://doi.org/10.1007/3-540-53863-1%5C_36`.

[Var85] Moshe Y. Vardi. 'Automatic Verification of Probabilistic Concurrent Finite-State Programs'. In: *26th Annual Symposium on Foundations of Computer Science, Portland, Oregon, USA, 21-23 October 1985.* IEEE Computer Society, 1985, pp. 327–338. DOI: `10.1109/SFCS.1985.12`. URL: `https://doi.org/10.1109/SFCS.1985.12`.

[Var99] Moshe Y. Vardi. 'Probabilistic Linear-Time Model Checking: An Overview of the Automata-Theoretic Approach'. In: *Formal Methods for Real-Time and Probabilistic Systems, 5th International AMAST Workshop, ARTS'99, Bamberg, Germany, May 26-28, 1999. Proceedings.* Ed. by Joost-Pieter Katoen. Vol. 1601. Lecture Notes in Computer Science. Springer, 1999, pp. 265–276. DOI: 10.1007/3-540-48778-6_16. URL: https://doi.org/10.1007/3-540-48778-6%5C_16.

[VW86] 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.* IEEE Computer Society, 1986, pp. 332–344.

[WDH+20] Alvin Wan, Lisa Dunlap, Daniel Ho, Jihan Yin, Scott Lee, Henry Jin, Suzanne Petryk, Sarah Adel Bargal and Joseph E. Gonzalez. ‘NBDT: Neural-Backed Decision Trees’. In: *CoRR* abs/2004.00221 (2020). arXiv: 2004.00221. URL: https://arxiv.org/abs/2004.00221.

[Wik20a] Wikipedia contributors. *Ethiopian Airlines Flight 302 — Wikipedia, The Free Encyclopedia.* https://en.wikipedia.org/w/index.php?title=Ethiopian_Airlines_Flight_302&oldid=976926129. [Online; accessed 8-September-2020]. 2020.

[Wik20b] Wikipedia contributors. *Lion Air Flight 610 — Wikipedia, The Free Encyclopedia.* https://en.wikipedia.org/w/index.php?title=Lion_Air_Flight_610&oldid=976898113. [Online; accessed 8-September-2020]. 2020.

[WJ06] Nicolás Wolovick and Sven Johr. ‘A Characterization of Meaningful Schedulers for Continuous-Time Markov Decision Processes’. In: *Formal Modeling and Analysis of Timed Systems, 4th International Conference, FORMATS 2006, Paris, France, September 25-27, 2006, Proceedings.* Ed. by Eugene Asarin and Patricia Bouyer. Vol. 4202. Lecture Notes in Computer Science. Springer, 2006, pp. 352–367. DOI: 10.1007/11867340_25. URL: https://doi.org/10.1007/11867340%5C_25.

[Yov96] Sergio Yovine. ‘Model Checking Timed Automata’. In: *Lectures on Embedded Systems, European Educational Forum, School on Embedded Systems, Veldhoven, The Netherlands, November 25-29, 1996.* Ed. by Grzegorz Rozenberg and Frits W. Vaandrager. Vol. 1494. Lecture Notes in Computer Science. Springer, 1996, pp. 114–152. DOI: 10.1007/3-540-65193-4_20. URL: https://doi.org/10.1007/3-540-65193-4%5C_20.

[ZVJ18] Ivan S. Zapreev, Cees Verdier and Manuel Mazo Jr. ‘Optimal Symbolic Controllers Determinization for BDD storage’. In: *6th IFAC Conference on Analysis and Design of Hybrid Systems, ADHS 2018, Oxford, UK, July 11-13, 2018.* Ed. by Alessandro Abate, Antoine Girard and Maurice Heemels. Vol. 51. IFAC-PapersOnLine 16. Elsevier, 2018, pp. 1–6. DOI: 10.1016/j.ifacol.2018.08.001. URL: https://doi.org/10.1016/j.ifacol.2018.08.001.

[ZYM+19] Quanshi Zhang, Yu Yang, Haotian Ma and Ying Nian Wu. 'Interpreting CNNs via Decision Trees'. In: *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019, Long Beach, CA, USA, June 16-20, 2019*. Computer Vision Foundation / IEEE, 2019, pp. 6261–6270. DOI: 10.1109/CVPR.2019.00642. URL: http://openaccess.thecvf.com/content%5C_CVPR%5C_2019/html/Zhang%5C_Interpreting%5C_CNNs%5C_via%5C_Decision%5C_Trees%5C_CVPR%5C_2019%5C_paper.html.

Appendix

Part I

Controller Synthesis

A Monte Carlo Tree Search for Verifying Reachability in Markov Decision Processes (ISoLA 2018)

This paper has been published as a **peer reviewed conference paper**.

Pranav Ashok, Tomás Brázdil, Jan Kretínský and Ondrej Slámecka. 'Monte Carlo Tree Search for Verifying Reachability in Markov Decision Processes'. In: *Leveraging Applications of Formal Methods, Verification and Validation. Verification - 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part II.* ed. by Tiziana Margaria and Bernhard Steffen. Vol. 11245. Lecture Notes in Computer Science. Springer, 2018, pp. 322–335. DOI: `10.1007/978-3-030-03421-4\_21`. URL: `https://doi.org/10.1007/978-3-030-03421-4%5C_21`

Summary

Monte Carlo Tree Search (MCTS) has been a very successful search and exploitation technique from reinforcement learning. In recent years, its effectiveness has been most notably demonstrated in AlphaGo [SHM+16], a Go playing algorithm that defeated the (human) Go world champion. On the other hand, asynchronous value iteration in the form of BRTDP has been shown to be very good at solving reachability objectives with hard guarantees on certain Markov Decision Processes with extremely large state spaces.

In this paper, we combine the best of the two worlds to provide asynchronous value iteration algorithms driven by MCTS. We implement 3 algorithms with varying degrees of mixing between MCTS and BRTDP. Results show that our algorithms perform as good as BRTDP on most examples, and in some cases, produce a result even when BRTDP times out.

Contribution

Composition and revision of the manuscript with significant role in writing sections 3, 4 and 5. Discussion and development of the ideas, implementation and evaluation with the following notable individual contributions: identification of the 3 variants of the algorithm, co-development of the implementation, evaluation.

Monte Carlo Tree Search for Verifying Reachability in Markov Decision Processes

Pranav Ashok[1], Tomáš Brázdil[2], Jan Křetínský[1]( ), and Ondřej Slámečka[2]

[1] Technical University of Munich, Munich, Germany
jan.kretinsky@tum.de
[2] Masaryk University, Brno, Czech Republic

Abstract. The maximum reachability probabilities in a Markov decision process can be computed using value iteration (VI). Recently, simulation-based heuristic extensions of VI have been introduced, such as bounded real-time dynamic programming (BRTDP), which often manage to avoid explicit analysis of the whole state space while preserving guarantees on the computed result. In this paper, we introduce a new class of such heuristics, based on Monte Carlo tree search (MCTS), a technique celebrated in various machine-learning settings. We provide a spectrum of algorithms ranging from MCTS to BRTDP. We evaluate these techniques and show that for larger examples, where VI is no more applicable, our techniques are more broadly applicable than BRTDP with only a minor additional overhead.

1 Introduction

Markov decision processes (MDP) [Put14] are a classical formalism for modelling systems with both non-deterministic and probabilistic behaviour. Although there are various analysis techniques for MDP that run in polynomial time and return precise results, such as those based on linear programming, they are rarely used. Indeed, dynamic programming techniques, such as value iteration (VI) or policy iteration, are usually preferred despite their exponential complexity. The reason is that for systems of sizes appearing in practice not even polynomial algorithms are useful and heuristics utilizing the structure of the human-designed systems become a necessity. Consequently, probabilistic model checking has adopted [BCC+14,ACD+17,KM17,KKKW18,DJKV17] techniques, which generally come with weaker guarantees on correctness and running time, but in practice perform better. These techniques originate from reinforcement learning, such as delayed Q-learning [SLW+06], or probabilistic planning, such as bounded real-time dynamic programming (BRTDP) [MLG05].

This research was supported in part by Deutsche Forschungsgemeinschaft (DFG) through the TUM International Graduate School of Science and Engineering (IGSSE) project 10.06 *PARSEC*, the Czech Science Foundation grant No. 18-11193S, and the DFG project 383882557 *Statistical Unbounded Verification*.

T. Margaria and B. Steffen (Eds.): ISoLA 2018, LNCS 11245, pp. 322–335, 2018.
https://doi.org/10.1007/978-3-030-03421-4_21

Since verification techniques are applied in safety-critical settings, the results produced by the techniques are required to be correct and optimal, or at least ε-optimal for a given precision ε. To this end, we follow the general scheme of [BCC+14] and combine the non-guaranteed heuristics with the traditional guaranteed techniques such as VI. However, while pure VI analyses the whole state space, the heuristics often allow us to focus only on a small part of it and still give precise estimates of the induced error. This approach was applied in [BCC+14], yielding a variant of BRTDP for MDP with reachability. Although BRTDP has already been shown to be quite good at avoiding unimportant parts of the state space in many cases, it struggles in many other settings, for instance where the paths to the goal are less probable or when the degree of non-determinism is high.

In this paper, we go one step further and bring a yet less guaranteed, but yet more celebrated technique of *Monte Carlo tree search* (MCTS) [AK87,KS06, Cou07,BPW+12] into verification. MCTS is a heuristic search algorithm which combines exact computation using search trees with sampling methods. To find the best actions to perform, MCTS constructs a search tree by successively unfolding the state-space of the MDP. The value of each newly added state is evaluated using simulations (also called roll-outs) and its value is backpropagated through the already existing search tree.

We show that the exact construction of the search tree in MCTS mitigates some of the pitfalls of BRTDP which relies completely on simulation. Namely, the search tree typically reaches less probable paths much sooner than a BRTDP simulation, e.g., in the example depicted in Fig. 1. We combine MCTS with BRTDP in various ways, obtaining thus a spectrum of algorithms ranging from pure BRTDP to pure MCTS along with a few hybrids in between. The aim is to overcome the weaknesses of BRTDP, while at the same time allowing to tackle large state spaces, which VI is unable to handle, with guaranteed (ε)-optimality of the solution.

While usually performing comparable to BRTDP, we are able to provide reasonable examples which can be tackled using neither BRTDP nor VI, but with our MCTS-BRTDP hybrid algorithms. Consequently, we obtain a technique applicable to larger systems, unlike VI, which is more broadly applicable than BRTDP with not much additional overhead.

Our contribution can be summarized as follows:

- We provide several ways of integrating MCTS into verification approaches so that the resulting technique is an anytime algorithm, returning the maximum reachability probability in MDP together with the respective error bound.
- We evaluate the new techniques and compare them to the state-of-the-art implementations based on VI and BRTDP. We conclude that for larger systems, where VI is not applicable, MCTS-based techniques are more robust than BRTDP.

1.1 Related Work

The correctness of the error bounds in our approach is guaranteed through the computation of the lower and upper bounds on the actual value. Such a computation has been established for MDP in [BCC+14,HM17] and the technique is based on the classical notion of the MEC quotient [dA97].

Statistical Model Checking (SMC). [YS02] is a collection of simulation-based techniques for verification where confidence intervals and probably approximately correct results (PAC) are sufficient. While on Markov chain it is essentially the Monte Carlo technique, on MDP it is more complex and its use is limited [Kre16], resulting in either slow [BCC+14] or non-guaranteed [HMZ+12,DLST15] methods. In contrast, MCTS combines Monte Carlo evaluation with explicit analysis of parts of the state space and thus opens new ways of integrating simulations into MDP verification.

Monte Carlo Tree Search (MCTS). There is a huge amount of literature on various versions of MCTS and applications. See [BPW+12] for an extensive survey. MCTS has been spectacularly successful in several domains, notably in playing classical board games such as Go [SHM+16] and chess [SHS+17].

Many variants of MCTS can be distinguished based on concrete implementations of its four phases: Selection and expansion, where a search tree is extended, roll-out, where simulations are used to evaluate newly added nodes, and back-up, where the result of roll-out is propagated through the search tree. In the selection phase, actions can be chosen based on various heuristics such as the most common UCT [KS06] and its extensions such as FPU and AMAF [GW06]. The evaluation phase has also been approached from many directions. To improve upon purely random simulation, domain specific rules have been employed to guide the choice of actions [ST09] and (deep) reinforcement learning (RL) techniques have been used to learn smart simulation strategies [SHM+16]. On the other hand, empirical evidence shows that it is often more beneficial to make simple random roll-outs as opposed to complex simulations strategies [JKR17]. RL has also been integrated with MCTS to generalize UCT by temporal difference (TD) backups [SSM12,VSS17].

2 Preliminaries

2.1 Markov Decision Processes

A *probability distribution* on a finite set X is a mapping $d : X \mapsto [0,1]$, such that $\sum_{x \in X} d(x) = 1$. We denote by $\mathcal{D}(X)$ the set of all probability distributions on X. Further, the *support* of a probability distribution ρ is denoted by $\mathrm{supp}(d) = \{x \in X \mid d(x) > 0\}$.

Definition 1 (MDP). *A* Markov decision processes (MDP) *is a tuple of the form* $\mathcal{M} = (S, s_{init}, Act, \mathsf{Av}, \Delta, \mathtt{1}, \mathtt{0})$*, where* S *is a finite set of* states*,* $s_{init}, \mathtt{1}, \mathtt{0} \in S$ *is the* initial *state,* goal *state, and* sink *state, respectively, Act is a finite set of* actions, $\mathsf{Av} : S \to 2^{Act}$ *assigns to every state a set of* available *actions, and* $\Delta : S \times Act \to \mathcal{D}(S)$ *is a* transition function *that given a state s and an action $a \in \mathsf{Av}(s)$ yields a probability distribution over successor states.*

An *infinite path* ρ in an MDP is an infinite word $\rho = s_0a_0s_1a_1 \cdots \in (S \times Act)^\omega$, such that for every $i \in \mathbb{N}$, $a_i \in \mathsf{Av}(s_i)$ and $\Delta(s_i, a_i, s_{i+1}) > 0$. A *finite path* $w = s_0a_0s_1a_1 \ldots s_n \in (S \times Act)^* \times S$ is a finite prefix of an infinite path. A *policy* on an MDP is a function $\pi : (S \times Act)^* \times S \to \mathcal{D}(Act)$, which given a finite path $w = s_0a_0s_1a_1 \ldots s_n$ yields a probability distribution $\pi(w) \in \mathcal{D}(\mathsf{Av}(s_n))$ on the actions to be taken next. We denote the set of all strategies of an MDP by Π. Fixing a policy π and an initial state s on an MDP $\mathcal{M}$ yields a unique probability measure $\mathbb{P}^{\pi}_{\mathcal{M},s}$ over infinite paths [Put14, Sect. 2.1.6] and thus the probability $V^\pi(s) := \mathbb{P}^{\pi}_{\mathcal{M},s}[\{\rho \mid \exists i \in \mathbb{N} : \rho(i) = \mathfrak{1}\}]$ to reach the goal state when following π.

Definition 2. *Given a state s, the* maximum reachability probability *is $V(s) = \sup_{\pi\in\Pi} V^\pi(s)$. A policy σ is ε-optimal if $V(s_{init}) - V^\sigma(s_{init}) \leq \varepsilon$.*

Note that there always exists a 0-optimal policy of the form $\pi : S \to Act$ [Put14].

A pair (T, A), where $\emptyset \neq T \subseteq S$ and $\emptyset \neq A \subseteq \bigcup_{s\in T} \mathsf{Av}(s)$, is an *end component* of an MDP $\mathcal{M}$ if

- for all $s \in T, a \in A \cap \mathsf{Av}(s)$ we have $\mathrm{supp}(\Delta(s, a)) \subseteq T$, and
- for all $s, s' \in T$ there is a finite path $w = sa_0 \ldots a_ns' \in (T \times A)^* \times T$, i.e. w starts in s, ends in s', stays inside T and only uses actions in A.

An end component (T, A) is a *maximal end component (MEC)* if there is no other end component (T', A') such that $T \subseteq T'$ and $A \subseteq A'$. The set of MECs of an MDP $\mathcal{M}$ is denoted by $\mathsf{MEC}(\mathcal{M})$. An MDP can be turned into its *MEC quotient* [dA97] as follows. Each MEC is merged into a single state, all the outgoing actions are preserved, except for a possible self-loop (when all its successors are in this MEC). Moreover, if there are no available actions left then it is merged with the sink state $\mathfrak{0}$. For a formal definition, see [ABKS18, Appendix A]. The techniques we discuss in this paper require the MDP be turned (gradually or at once) into its MEC-quotient in order to converge to V, as described below.

2.2 Value Iteration

Value iteration (VI) is a dynamic programming algorithm first described by Bellman [Bel57]. The maximum reachability probability is characterized as the least fixpoint of the following equation system for $s \in S$:

$$V(s) = \begin{cases} 1 & \text{if } s = \mathfrak{1} \\ 0 & \text{if } s = \mathfrak{0} \\ \max\limits_{a\in Av(s)} \sum\limits_{s'\in S} \Delta(s,a)(s') \cdot V(s') & \text{otherwise} \end{cases}$$

In order to compute the least fixpoint, an iterative method can be used as follows. We initialize $V^0(s) = 0$ for all s except for $V^0(\mathfrak{1}) = 1$. A successive iteration V^{i+1} is computed by evaluating the right-hand side of the equation system with V^i. The optimal value is achieved in the limit, i.e. $\lim_{n\to\infty} V^n = V$.

In order to obtain bounds on the imprecision of V^n, one can employ a *bounded* variant of VI [MLG05,BCC+14] (also called *interval iteration* [HM17]). Here one computes not only the *lower bounds* on V via the least-fixpoint approximants V^n (onwards denoted by L^n), but also *upper bounds* via the greatest-fixpoint approximants, called U^n. The greatest-fixpoint can be approximated in a dual way, by initializing the values to 1 except for $\mathfrak{0}$ where it is 0. On MDPs without MECs (except for $\mathfrak{1}$ and $\mathfrak{0}$), we have both $\lim_{n\to\infty} L^n = V = \lim_{n\to\infty} U^n$, giving us an anytime algorithm with the guarantee that $V \in [L^n, U^n]$. However, on general MDPs, $\lim_{n\to\infty} U^n$ can be larger than V, often yielding only a trivial bound of 1, see [ABKS18, Appendix B]. The solution suggested in [BCC+14, HM17] is to consider the MEC quotient instead, where the guarantee is recovered. The MEC quotient can be pre-computed [HM17] or computed gradually on-the-fly only for a part of the state space [BCC+14].

2.3 Bounded Real-Time Dynamic Programming (BRTDP)

BRTDP [MLG05] is a heuristic built on top of (bounded) VI, which has been adapted to the reachability objective in [BCC+14]. It belongs to a class of *asynchronous* VI algorithms. In other words, it differs from VI in that in each iteration it does not update the values for all states, but only a few. The states to be updated are chosen as those that appear in a simulation run performed in that iteration. This way we focus on states visited with high probability and thus having higher impact on the value.

BRTDP thus proceeds as follows. It maintains the current lower and upper bounds L and U on the value, like bounded VI. In each iteration, a simulation run is generated by choosing in each state s

– the action maximizing U, i.e. $\arg\max_a \sum_{s'\in S} \Delta(s,a)(s') \cdot U(s')$,
– the successor s' of a randomly with weight $\Delta(s,a)(s')$ or (more efficiently) in the variant of [BCC+14] with weight $\Delta(s,a)(s') \cdot (U(s') - L(s'))$.

Then the value of each state on the simulation run is updated by the equation system for V. This happens preferably in the backward order [BCC+14] from the last state ($\mathfrak{1}$ or $\mathfrak{0}$) towards s_{init}, thus efficiently propagating the value information. These iterations are performed repeatedly until $U(s_{\text{init}}) - L(s_{\text{init}}) < \varepsilon$ for some predefined precision ε, yielding the interval $[L(s_{\text{init}}), U(s_{\text{init}})]$ containing the value $V(s_{\text{init}})$.

If the only MECs are formed by $\mathfrak{1}$ and $\mathfrak{0}$ then the algorithm (depicted in [ABKS18, Appendix C, Algorithm 2] terminates almost surely. But if there are other MECs, then this is not necessarily the case. In order to ensure correct termination, [BCC+14] modifies the algorithm in a way, which we adopt also for our algorithms: Periodically, after a certain number of iterations, we compute the

MEC quotient, not necessarily of the whole MDP, but only of the part formed by the states visited so far by the algorithm.

3 Monte-Carlo Tree Search

Motivation. One of the main challenges in the application of BTRDP are the presence of events that happen after a long time (but are not necessarily rare). For example, consider the simple MDP in Fig. 1, modelling a hypothetical communication protocol. Let s_3 be the goal state, representing a failure state. From each of the states s_0, s_1 and s_2, the MDP tries to send a message. If the message sending fails, which happens with a very low probability, the MDP moves to the next state, otherwise returning to the initial state s_0. BRTDP repeatedly samples paths and propagates ("backs-up") the encountered values through the paths. Even though the goal is reached in almost every simulation, it can take very long time to finish each simulation.

The idea of MCTS is to "grow" a tree into the model rooted at the initial state while starting new simulations from the leaves of the tree. In this example, the tree soon expands to s_1 or s_2 and from then on, simulations are started there and thus are more targeted and the back-propagation of the values is faster.

Generic Algorithm. MCTS can be visualized (see Fig. 2) as gradually building up a tree, which describes several steps of unfolding of the system, collecting more

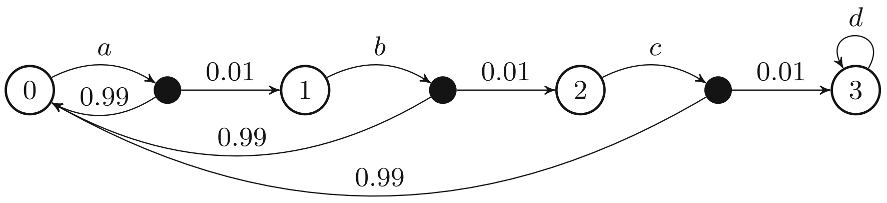


Fig. 1. A difficult case for BRTDP

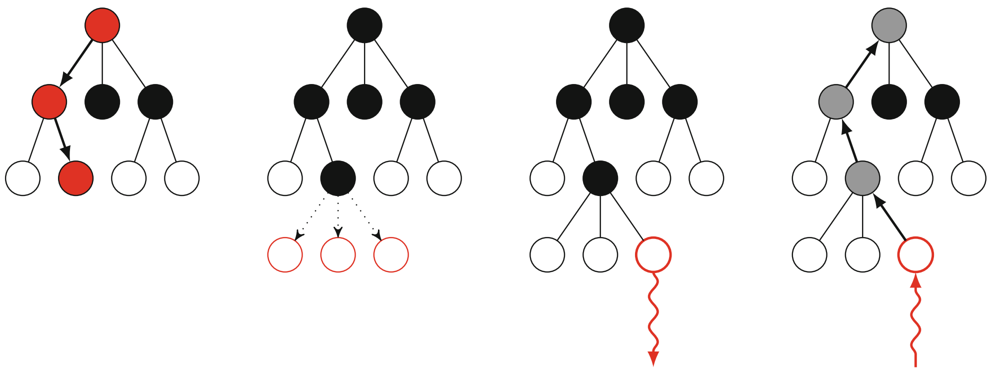


Fig. 2. The four stages of MCTS: *selection*, *expansion*, *roll-out* and *back-propagation*

Algorithm 1. The (enriched) MCTS scheme

```
1: Create a root $t_0$ of the tree, labelled by $s_{\text{init}}$
2: while within computational budget do
3:     $t_{\text{parent}} \leftarrow$ SelectNode($t_0$)
4:     $t_{\text{child}} \leftarrow$ ExpandAndPickroll-outNode($t_{\text{parent}}$)
5:     $outcome \leftarrow$ roll-out($t_{\text{child}}$)
6:     BackupOnRoll-out($outcome$)                    ▷ Not in the standard MCTS
7:     BackupOnTree($t_{\text{child}}, outcome$)
8: return InducedPolicy($t_0$)              ▷ action with the best estimated value
```

and more information. Note that nodes of the tree correspond to states of the MDP.

The MCTS procedure (see Algorithm 1) proceeds in rounds, each consisting of several stages:

First, in the **selection** stage (line 3), a *tree policy* is followed leading from the root of the tree to the currently most "interesting" leaf.

Second, in the **expansion** stage (line 4), one or more of the successors are added to the tree. In our setting with required guarantees, it turns out sensible to expand all successors, i.e. $\bigcup_{a \in \mathsf{Av}(t_{\text{parent}})} \text{supp}(\Delta(s_{\text{parent}}, a))$. One of the successors is picked, according to the tree policy.

Third, in the **roll-out** stage (line 5) a simulation run is generated, adhering to a *roll-out policy*. In the classical setting, the simulation run receives a certain reward. In our setting, the simulation run receives a reward of 1 if it encounters the goal state, otherwise it receives 0.

Finally, in the **back-propagation** (or **backup**) stage, the information whether a goal was reached during the roll-out is propagated up the tree (line 7). The information received through such simulations helps MCTS to decide the direction in which the tree must be grown in the next step. On top of the standard MCTS, we also consider possible back-propagation of this information along the generated simulations run (line 6).

Next, we discuss a particularly popular way to implement the MCTS scheme.

UCT Implementation. Each stage of the MCTS scheme can be instantiated in various ways. We now describe one of the most common and successful implementations, called UCT [KS06] (abbreviation of "UCB applied to Trees"). Each node t in the tree keeps track of two numbers: n_t is the number of times t has been visited when selecting leaves; v_t is the number of times it has been visited *and* the roll-out from the descendant leaf hit the goal. These numbers can be easily updated during the back-propagation: we simply increase all n's on the way from the leaf to the root, similarly for the v's whenever the roll-out hit the goal. This allows us to define the last missing piece, namely the tree policy based on these values, which is called *Upper Confidence Bound* (UCB1 or simply UCB). The UCB1 value of a node t in the MCTS tree is defined as follows:

$$\mathit{UCB1}(t) = \frac{v_t}{n_t} + C\sqrt{\frac{\ln n_{\mathrm{parent}(t)}}{n_t}}$$

where parent(t) is the parent of t in the tree and C is a fixed *exploration-exploitation constant*. The tree policy choosing nodes with maximum UCB1 bound is called the *UCB1 policy*.

Intuitively, the first term of the expression describes the ratio of "successful" roll-outs (i.e., hitting the goal) started from a descendant of t. In other words, this *exploitation* term tries to approximate the probability to reach the goal from t. The higher the value, the higher the chances that the tree policy would pick t. In contrast, the second term decreases when the node t is chosen too often. This compensates for the first term, thereby allowing for the siblings of t to be explored. This is the *exploration* term and the effect of the two terms is balanced by the constant C. The appropriate values of C vary among domains and, in practice, are determined experimentally.

It has been proved in [KS06] that for finite-horizon MDP (or MDP with discounted rewards), the UCT with an appropriately selected C converges in the following sense: After n iterations of the MCTS algorithm, the difference between the expectation of the value estimate and the true value of a given state is bounded by $\mathcal{O}(\log(n)/n)$. Moreover, the probability that an action with the best upper confidence bound in the root is not optimal converges to zero at a polynomial rate with $n \to \infty$.

4 Augmenting MCTS with Guarantees

In this section, we present several algorithms based on MCTS. Note that the typical uses of MCTS in practice only require the algorithm to guess a good but not necessarily the optimal action for the current state. When adapting learning algorithms to the verification setting, an important requirement is an ability to give precise guarantees on the sought value or to produce an ε-optimal scheduler. Consequently, in order to obtain the guarantees, our algorithms combine MCTS with BRTDP, which comes with guarantees, to various degrees. The spectrum is depicted in Table 1 and described in more detail below.

Table 1. Spectrum of algorithms ranging from pure MCTS to pure BRTDP

Algorithm	MCTS	BMCTS	MCTS-BRTDP	BRTDP-UCB	BRTDP
SelectNode (Tree Policy)	UCB1	UCB1	UCB1	UCB1	BRTDP
Roll-out	Uniform	Uniform	BRTDP		
BackpupOnRoll-out	—	L, U	L, U	$v, n,\ L, U$	L, U
BackupOnTree	v, n	$v, n,\ L, U$	$v, n,\ L, U$		

1. Pure MCTS (the UCT variant): The tree is constructed using the UCB1 policy while roll-outs are performed using a uniform random policy, i.e. actions are chosen uniformly, successors according to their transition probabilities. The roll-outs are either successful or not, and this information is back-propagated up the tree, i.e. the v- and n-values of the nodes in the tree are updated.
2. Bounded MCTS (BMCTS): In addition to all the steps of pure MCTS, here we also update the L- and U-values, using the transition probabilities and the old L- and U-values. This update takes place both on all states in the tree on the path from the root to the current leaf as well as all states of visited during the roll-out. The updates happen backwards, starting from the last state of the simulation run, towards the leaf and then towards the root.
3. MCTS-BRTDP: This algorithm is essentially BMCTS with the only difference that the roll-out is performed using the BRTDP policy, i.e. action is chosen as $\arg\max_{a \in \mathsf{Av}(s)} \sum_{s'} \Delta(s,a)(s') \cdot U(s')$, and the successor of a randomly with the weight $\Delta(s,a)(s') \cdot (U(s') - L(s'))$.
4. BRTDP-UCB: As we move towards the BRTDP side of the spectrum, there is no difference between whether a state is captured in the tree or not: back-propagation works the same and the policy to select the node is the same as for the roll-out. In this method, we use the UCB1 policy to choose actions on the whole path from the initial state all the way to the goal or the sink, and back-propagate all information. Note that as opposed to BRTDP, the exploitation is interleaved with some additional exploration, due to UCB1.
5. Pure BRTDP: Finally, we also consider the pure BRTDP algorithm as presented earlier in the paper. This works the same as BRTDP-UCB, but the (selection and roll-out) policy is that of BRTDP.

While MCTS does not provide exact guarantees on the current error, all the other methods keep track of the lower and the upper bound, yielding the respective guarantee on the value. Note that MCTS-BRTDP is a variant of MCTS, where BRTDP is used not only to provide these bounds, but also to provide a more informed roll-out. Such a policy is expected to be more efficient compared to just using a uniform policy or UCB. Since some studies [JKR17] have counter-intuitively shown that more informed roll-outs do not necessarily improve the runtime, we also include BMCTS and BRTDP-UCB, where the path generation is derived from the traditional MCTS approach; the former applies it in the MCTS setting, the latter in the BRTDP setting.

MDPs with MECs. In MDPs where the only MECs are formed by $\mathfrak{1}$ and $\mathfrak{0}$, the roll-outs almost surely reach one of the two states and then stop. Since in MDPs with MECs this is not necessarily the case, we have to collapse the MECs, like discussed in Sect. 2.3. Therefore, a roll-out $w = s_0 s_1 s_2 \dots s_n$ is stopped if $s_n \in \{\mathfrak{1}, \mathfrak{0}\}$ or $s_n = s_k$, for some $0 \leq k < n$. In the latter case, there is a chance that an end component has been discovered. Hence we compute the MEC quotient and only then continue the process. When MECs are collapsed this way, both the lower and the upper bound converge to the value and the resulting methods terminate in finite time almost surely, due to reasoning of [BCC+14] and the exploration term being positive.

Example 1. We revisit the example of Fig. 1 to see how the MCTS-based algorithms presented above tend to work. We focus on MCTS-BRTDP. Once the algorithm starts it is easy to see that in 3 iterations of MCTS-BRTDP, the target state s_3 will belong to the MCTS tree. From the next iteration onwards, the selection step will always choose s_3 to start the roll-out from. But since s_3 is already a target, the algorithm just proceeds to update the information up the tree to the root state. Hence, with just 3 iterations more than what value iteration would need, the algorithm converges to the same result.

While this example is quite trivial, we show experimentally in the next section that the MCTS-based algorithms not only run roughly as fast as BRTDP, but in certain large models that exhibit behaviour similar to the example of Fig. 1, it is clearly the better option.

5 Experimental Evaluation

We implemented all the algorithms outlined in Table 1 in PRISM Model Checker [KNP11]. Table 2 presents the run-times obtained on twelve different models. Six of the models used (`coin, leader, mer, firewire, zeroconf` and `wlan`) were chosen from the PRISM Benchmark Suite [KNP12] and the remaining six were constructed by modifying `firewire`, `zeroconf` and `wlan` in order to demonstrate the strengths and weaknesses of VI, BRTDP, and our algorithms. Recall the motivational example of Fig. 1 in Sect. 3, which is hard for BRTDP, yet easy for VI. We refer to this MDP as `brtdp-adversary`. This hard case is combined with one of benchmarks, `firewire`, `zeroconf` and `wlan`, in two ways as follows.

Branch Models. In the first construction, we allow a branching decision at the initial state. The resulting model is shown in Fig. 3. If action a is chosen, then we enter the `brtdp-adversary` branch. If action b is chosen, then we enter the standard `zeroconf` model. Using this schema, we create three models: `branch-zeroconf`, `branch-firewire` and `branch-wlan`.

Composition Models. In the second construction, the models are constructed by the parallel (asynchronous) composition of the `brtdp-adversary` and one of `firewire`, `zeroconf` and `wlan`. The resulting models are named `comp-firewire`, `comp-zeroconf` and `comp-wlan`.

We run the experiments on an Intel Xeon server with sufficient memory (default PRISM settings). The implementation is single threaded and each experiment is run 15 times to alleviate the effect of the probabilistic nature of the algorithms. The median run-time for each model configuration is reported in Table 2. Note that the measured time is the wall-clock time needed to perform the model checking. We do not include the time needed to start PRISM and to read and parse the input file. An execution finishes successfully once the value of

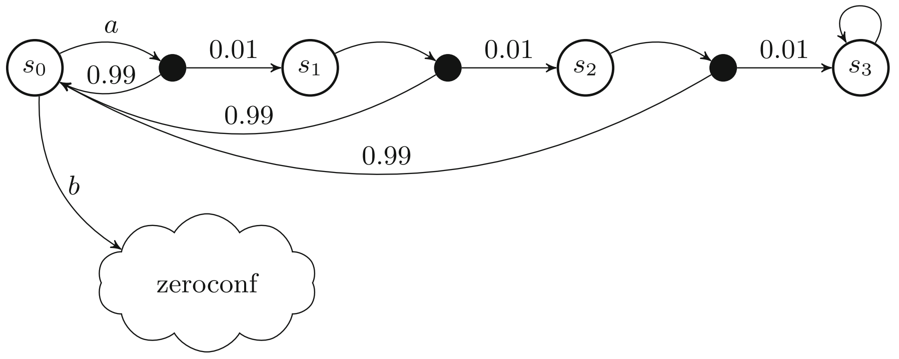


Fig. 3. Combining zeroconf model with the BRTDP adversary in a branching manner.

Table 2. Comparison of the various algorithms. All run-times are in seconds. Cells with '–' denote either running out of memory (in the case of VI) or running out of time.

Benchmark	BMCTS	MCTS-BRTDP	BRTDP-UCB	BRTDP	VI
consensus	5.55	6.48	7.47	6.15	1.13
leader	18.67	15.79	16.33	15.06	8.94
mer	–	4.79	–	3.63	–
firewire	0.07	0.08	0.09	0.09	6.99
wlan	0.09	0.07	0.08	0.08	–
zeroconf	0.93	0.20	0.59	0.20	–
comp-firewire	9.36	9.55	–	–	20.77
comp-wlan	2.51	2.25	–	–	–
comp-zeroconf	–	29.55	–	–	–
branch-firewire	0.09	0.09	0.02	0.09	9.33
branch-wlan	0.10	0.08	0.09	0.07	–
branch-zeroconf	25.90	30.78	35.67	38.14	–

the queried maximal reachability property is known with a guaranteed absolute precision of 10^{-6}, except for `comp-zeroconf` and `branch-zeroconf` where it stops once the value is known with a precision of 10^{-2}. Timeout is set to 10 min.

Table 2 shows that in the cases where BRTDP performs well, our algorithms perform very similarly, irrespective of the performance of VI. However, when BRTDP struggles due to the hard case, our MCTS-based algorithms still perform well, even in cases where VI also times out. In particular, MCTS-BRTDP shows a consistently good performance. The less informed variants BMCTS and BRTDP-UCB of MCTS-BRTDP and BRTDP, respectively, perform consistently at most as well their respective more informed variants. This only confirms the intuitive expectation which, however, has been shown invalid in other settings.

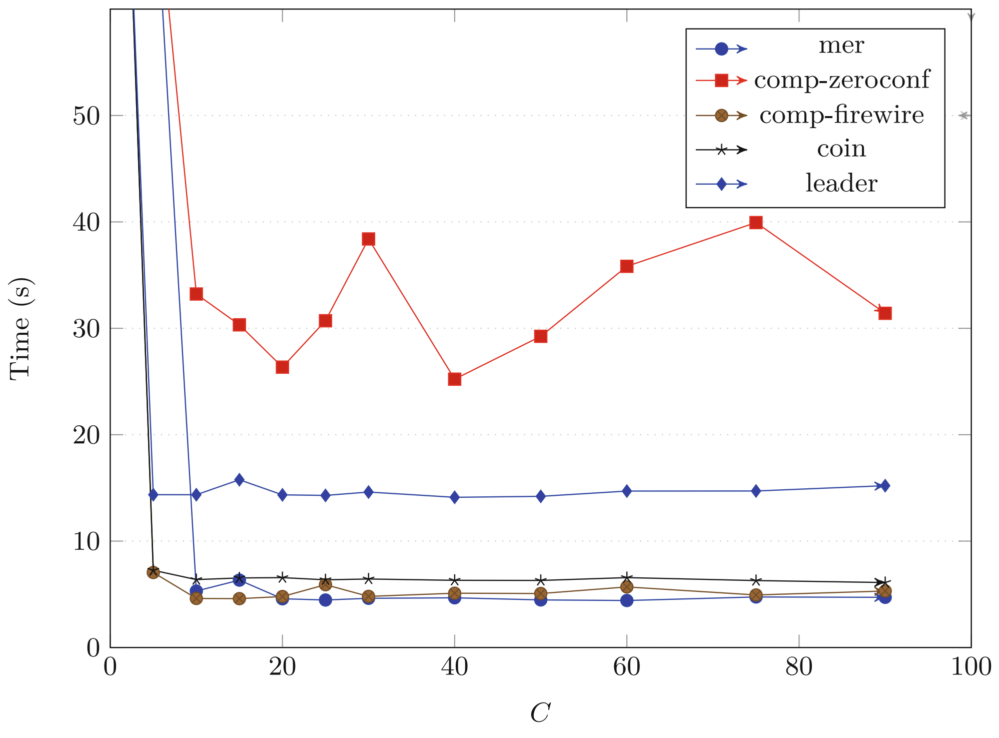


Fig. 4. Dependence of the run-time of MCTS-BRTDP on the exploration-exploitation constant, C

In the experiments, we have used the exploration-exploitation constant $C = 25$, which is rather large compared to the typical usage in literature [KSW06, BPW+12]. This significantly supports exploration. The effect of the constant is illustrated in Fig. 4. We can see that for lower values of C, the algorithms perform worse, due to insufficient incentive to explore.

In conclusion, we see that exploration, which is present in MCTS but not explicitly in BRTDP, leads to a small overhead which, however, pays off dramatically in more complex cases. For the total number of states explored by VI, BRTDP, and our MCTS-based algorithms, we refer the reader to [ABKS18, Appendix D, Table 3].

6 Conclusion

We have introduced Monte Carlo tree search into verification of Markov decision processes, ensuring hard guarantees of the resulting methods. We presented several algorithms, covering the spectrum between MCTS and (in this context more traditional) MDP-planning heuristic BRTDP. Our experiments suggest that the overhead caused by additional exploration is outweighed by the ability of the technique to handle the cases, which are more complicated for BRTDP. Further, similarly to BRTDP, the techniques can handle larger state spaces, where the traditional value iteration fails to deliver any result. Consequently, the method is more robust and applicable to a larger spectrum of examples, at the cost of a negligible overhead.

References

[ABKS18] Ashok, P., Brázdil, T., Křetínský, J., Slámečka, O.: Monte Carlo tree search for verifying reachability in Markov decision processes. CoRR abs/1809.03299 (2018)

[ACD+17] Ashok, P., Chatterjee, K., Daca, P., Křetínský, J., Meggendorfer, T.: Value iteration for long-run average reward in Markov decision processes. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 201–221. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_10

[AK87] Abramson, B., Korf, R.E.: A model of two-player evaluation functions. In: Proceedings of the 6th National Conference on Artificial Intelligence (AAAI 1987), pp. 90–94. Morgan Kaufmann (1987)

[BCC+14] Brázdil, T., et al.: Verification of Markov decision processes using learning algorithms. In: Cassez, F., Raskin, J.-F. (eds.) ATVA 2014. LNCS, vol. 8837, pp. 98–114. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11936-6_8

[Bel57] Bellman, R.: A Markovian decision process. 6:15 (1957)

[BPW+12] Browne, C., et al.: A survey of monte carlo tree search methods. IEEE Trans. Comput. Intell. AI Games **4**, 1–43 (2012)

[Cou07] Coulom, R.: Efficient selectivity and backup operators in Monte-Carlo tree search. In: van den Herik, H.J., Ciancarini, P., Donkers, H.H.L.M.J. (eds.) CG 2006. LNCS, vol. 4630, pp. 72–83. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-75538-8_7

[dA97] de Alfaro, L.: Formal verification of probabilistic systems. Ph.D. thesis, Stanford University (1997)

[DJKV17] Dehnert, C., Junges, S., Katoen, J.P., Volk, M.: A storm is coming: a modern probabilistic model checker. In: Majumdar, R., Kunčak, V. (eds.) Computer Aided Verification. CAV 2017. Lecture Notes in Computer Science, vol. 10427, pp. 592–600. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63390-9_31

[DLST15] D'Argenio, P., Legay, A., Sedwards, S., Traonouez, L.-M.: Smart sampling for lightweight verification of Markov decision processes. Int. J. Softw. Tools Technol. Transf. **17**(4), 469–484 (2015)

[GW06] Gelly, S., Wang, Y.: Exploration exploitation in Go: UCT for Monte-Carlo Go. In: NIPS: Neural Information Processing Systems Conference On-line Trading of Exploration and Exploitation Workshop, Canada, December 2006

[HM17] Haddad, S., Monmege, B.: Interval iteration algorithm for MDPs and IMDPs. Theor. Comput. Sci. **735**, 111–131 (2018)

[HMZ+12] Henriques, D., Martins, J., Zuliani, P., Platzer, A., Clarke, E.M.: Statistical model checking for Markov decision processes. In: QEST, pp. 84–93 (2012)

[JKR17] James, S., Konidaris, G., Rosman, B.: An analysis of Monte Carlo tree search. In: Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, 4–9 February 2017, San Francisco, California, USA, pp. 3576–3582 (2017)

[KKKW18] Kelmedi, E., Krämer, J., Kretínský, J., Weininger, M.: Value iteration for simple stochastic games: stopping criterion and learning algorithm. In: CAV (1), pp. 623–642. Springer (2018)

[KM17] Křetínský, J., Meggendorfer, T.: Efficient strategy iteration for mean payoff in Markov decision processes. In: ATVA, pp. 380–399 (2017)

[KNP11] Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 585–591. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_47

[KNP12] Kwiatkowska, M., Norman, G., Parker, D.: The PRISM benchmark suite. In: Proceedings of 9th International Conference on Quantitative Evaluation of SysTems (QEST 2012), pp. 203–204. IEEE CS Press (2012)

[Kre16] Křetínský, J.: Survey of statistical verification of linear unbounded properties: model checking and distances. In: Margaria, T., Steffen, B. (eds.) ISoLA 2016. LNCS, vol. 9952, pp. 27–45. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47166-2_3

[KS06] Kocsis, L., Szepesvári, C.: Bandit based Monte-Carlo planning. In: Fürnkranz, J., Scheffer, T., Spiliopoulou, M. (eds.) ECML 2006. LNCS (LNAI), vol. 4212, pp. 282–293. Springer, Heidelberg (2006). https://doi.org/10.1007/11871842_29

[KSW06] Kocsis, L., Szepesvári, C., Willemson, J.: Improved Monte-Carlo search (2006)

[MLG05] Brendan McMahan, H., Likhachev, M., Gordon, G.J.: Bounded real-time dynamic programming: RTDP with monotone upper bounds and performance guarantees. In: ICML 2005 (2005)

[Put14] Puterman, M.L.: Markov Decision Processes: Discrete Stochastic Dynamic Programming. Wiley, New York (2014)

[SHM+16] Silver, D., et al.: Mastering the game of go with deep neural networks and tree search. Nature **529**(7587), 484–489 (2016)

[SHS+17] Silver, D., et al.: Mastering chess and Shogi by self-play with a general reinforcement learning algorithm. CoRR, abs/1712.01815 (2017)

[SLW+06] Strehl, A.L., Li, L., Wiewiora, E., Langford, J., Littman, M.L.: PAC model-free reinforcement learning. In: ICML, pp. 881–888 (2006)

[SSM12] Silver, D., Sutton, R.S., Mueller, M.: Temporal-difference search in computer Go. Mach. Learn. **87**, 183–219 (2012)

[ST09] Silver, D., Tesauro, G.: Monte-Carlo simulation balancing. In: Proceedings of the 26th Annual International Conference on Machine Learning (ICML 2009), pp. 945–952. ACM (2009)

[VSS17] Vodopivec, T., Samothrakis, S., Ster, B.: On Monte Carlo tree search and reinforcement learning. J. Artif. Intell. Res. **60**, 881–936 (2017)

[YS02] Younes, H.L.S., Simmons, R.G.: Probabilistic verification of discrete event systems using acceptance sampling. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 223–235. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45657-0_17

B Continuous-time Markov Decisions Based on Partial Exploration (ATVA 2018)

This paper has been published as a **peer reviewed conference paper**.

Pranav Ashok, Yuliya Butkova, Holger Hermanns and Jan Kretínský. 'Continuous-Time Markov Decisions Based on Partial Exploration'. In: *Automated Technology for Verification and Analysis - 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings.* Ed. by Shuvendu K. Lahiri and Chao Wang. Vol. 11138. Lecture Notes in Computer Science. Springer, 2018, pp. 317–334. DOI: 10.1007/978-3-030-01090-4_19. URL: https://doi.org/10.1007/978-3-030-01090-4%5C_19

Summary

Continuous-time Markov Decision Process (CTMDP) are a continuous-time extension of MDPs in which an exponentially distributed delay is incurred in every state before a transition is executed. An important problem on CTMDP, called time-bounded reachability, is to synthesize a scheduler (or a controller) that optimizes the probability of reaching a set of goal states within a given deadline. While it is known that there exists a timed non-randomized memoryless scheduler that can achieve the optimum, it is however not known how to compute it. Hence, a large body of research focuses on obtaining ϵ-optimal schedulers.

In this paper, we present a framework to speed up existing time-bounded reachability algorithms via simulations, at the same time maintaining ϵ-guarantees. We achieve this by iteratively identifying sub-CTMDPs relevant for the given objective, constructing an under- and an over-approximation of each and use existing time-bounded reachability algorithms on it. This gives us, for every sub-CTMDP, a lower and upper bound on the optimal time-bounded reachability value. We stop when the difference between these bounds is ϵ. Our results show that for many CTMDPs in our benchmark suite, it is sufficient to explore only a small part of the state space, which in turn speeds up existing algorithms.

Contribution

Composition and revision of the manuscript with significant contributions to Sections 3 and 4. Discussion and development of the ideas, experimentation and evaluation. Co-lead role in the design and implementation of the presented tool and its integration with the PRISM Model Checker.

Continuous-Time Markov Decisions Based on Partial Exploration

Pranav Ashok[1], Yuliya Butkova[2], Holger Hermanns[2], and Jan Křetínský[1]( )

[1] Technical University of Munich, Munich, Germany
{ashok,jan.kretinsky}@in.tum.de
[2] Saarland University, Saarbrücken, Germany
{butkova,hermanns}@cs.uni-saarland.de

Abstract. We provide a framework for speeding up algorithms for time-bounded reachability analysis of continuous-time Markov decision processes. The principle is to find a small, but almost equivalent *subsystem* of the original system and only analyse the subsystem. Candidates for the subsystem are identified through simulations and iteratively enlarged until runs are represented in the subsystem with high enough probability. The framework is thus dual to that of abstraction refinement. We instantiate the framework in several ways with several traditional algorithms and experimentally confirm orders-of-magnitude speed ups in many cases.

1 Introduction

Continuous-time Markov decision processes (CTMDP) [Ber95, Sen99, Fei04] are the natural real-time extension of (discrete-time) Markov decision processes (MDP). They can likewise be viewed as non-deterministic extensions of continuous-time Markov chains (CTMC). As such, CTMDP feature probabilistic and non-deterministic behaviour as well as random time delays governed by exponential probability distributions. Prominent application areas of CTMDP include operations research [BDF81, Fei04], power management and scheduling [QQP01], networked, distributed systems [HHK00, GGL03], as well as epidemic and population processes [Lef81]. Moreover, CTMDPs are the core semantic model underlying formalisms such as generalised stochastic Petri nets, Markovian stochastic activity networks, and interactive Markov chains [EHKZ13].

A large variety of properties can be expressed using logics such as CSL [ASSB96]. Apart from classical techniques from the MDP context, the analysis of such properties relies fundamentally on the problem of time-bounded

This research was supported in part by Deutsche Forschungsgemeinschaft (DFG) through the TUM International Graduate School of Science and Engineering (IGSSE) project 10.06 *PARSEC*, the ERC through Advanced Grant 695614 (POWVER), the Czech Science Foundation grant No. 18-11193S, and the DFG project 383882557 *Statistical Unbounded Verification*.

S. K. Lahiri and C. Wang (Eds.): ATVA 2018, LNCS 11138, pp. 317–334, 2018.
https://doi.org/10.1007/978-3-030-01090-4_19

reachability (TBR), i.e. *what is the maximal/minimal probability to reach a given state within a given time bound*. Since this is the cornerstone of the analysis, a manifold of algorithms have been proposed for TBR [BHHK04, BFK+09, NZ10, FRSZ11, BS11, HH13, BHHK15]. While the algorithmic approaches are diverse, relying on uniformisation and various forms of discretization, they are mostly back-propagating the values computed, i.e. in the form of value iteration.

Not surprisingly, all these algorithms naturally process the state space of the CTMDP in its entirety. In this work we instead suggest a framework that enables TBR analysis with guaranteed precision while often exploring only a small, property-dependent part of the state space. Similar ideas have appeared for (discrete-time) MDPs and unbounded reachability [BCC+14] or mean payoff [ACD+17]. These techniques are based on *asynchronous* value-iteration approaches, originally proposed in the probabilistic planning world, such as bounded real-time dynamic programming (BRTDP) [MLG05]. Intuitively, the back-propagation of values (value iteration steps) are not performed on all states in each iteration (synchronously), but always only the "interesting" ones are considered (asynchronously); in order to bound the error in this approach, one needs to compute both an under- and an over-approximation of the actual value.

In other words, the main idea is to keep track of (under- and over-)approximation of the value when accepting that we have no information about the values attained in certain states. Yet if we can determine that these states are reached with very low probability, their effect on the actual value is provably negligible and thus the lack of knowledge only slightly increases the difference between the under- and over-approximations. To achieve this effect, the algorithm of [BCC+14] alternates between two steps: (i) simulating a run of the MDP using a (hopefully good) scheduler, and (ii) performing the standard value iteration steps on the states visited by this run.

It turns out that this idea cannot be transferred to the continuous-time setting easily. In technical terms, the main issue is that the value iteration in this context takes the form of synchronous back-propagation, which when implemented in an asynchronous fashion results in memory requirements that tend to dominate the memory savings expectable due to partial exploration.

Therefore, we twist the above approach and present a yet simpler algorithmic strategy in this paper. Namely, our approach alternates between several simulation steps, and a subsequent run of TBR analysis only focussed on the already explored subsystem, instead of the entire state space. If the distance between under- and over-approximating values is small enough, we can terminate; otherwise, running more simulations extends the considered state subspace, thereby improving the precision in the next round. If the underlying TBR analysis provides an optimal scheduler along with the value of time-bounded reachability, then our solution as well provides the optimal scheduler for the TBR problem on the given CTMDP.

There are thus two largely independent components to the framework, namely (i) a heuristic how to explore the system via simulation, and (ii) an algorithm to solve time-bounded reachability on CTMDP. The latter is here instantiated with some of the classic algorithms mentioned above, namely the first discretization-based algorithm [NZ10] and the two most competitive improvements over it [BS11,BHHK15], based on uniformisation and untimed analysis. The former basically boils down to constructing a scheduler resolving the nondeterminism effectively. We instantiate this exploration heuristics in two ways. Firstly, we consider a scheduler returned by the most recent run of the respective TBR algorithm, assuming this to yield a close-to-optimal scheduler, so as to visit the most important parts of the state space, relative to the property in question. Secondly, since this scheduler may not be available when working with TBR algorithms that return only the value, we also employ a scheduler resolving choices uniformly. Although the latter may look very straightforward, it turns out to already speed up the original algorithm considerably in many cases. This is rooted in the fact that that scheduler best represents the available knowledge, since the uniform distribution is the one with maximimal entropy.

Depending on the model and the property under study, different ratios of the state space entirety need to be explored to achieve the desired precision. Furthermore, our approach is able to exploit that the reachability objective is of certain forms, in stark contrast to the classic algorithm that needs to perform the same computation irrespective of the concrete set of target states. Still, the approach we propose will naturally profit from future improvements in effectiveness of classic TBR analysis.

We summarize our contribution as follows:

– We introduce a framework to speed up TBR algorithms for CTMDP and instantiate it in several ways. It is based on a partial, simulation-based exploration of the state space spanned by a model.
– We demonstrate its effectiveness in combination with several classic algorithms, obtaining orders of magnitude speed ups in many experiments. We also illustrate the limitations of this approach on cases where the state space needs to be explored almost in its entirety.
– We conclude that our framework is a generic add-on to arbitrary TBR algorithms, often saving considerably more work than introduced by its overhead.

2 Preliminaries

In this section, we introduce some central notions. A *probability distribution* on a finite set X is a mapping $\rho : X \to [0,1]$, such that $\sum_{x \in X} \rho(x) = 1$. $\mathcal{D}(X)$ denotes the set of all probability distributions on X.

Definition 1. *A* continuous-time Markov decision process *(CTMDP) is a tuple* $\mathcal{M} = (s_{init}, S, Act, \mathbf{R}, G)$ *where S is a finite set of* states, *s_{init} is the initial state, Act is a finite set of* actions, *$\mathbf{R} : S \times Act \times S \to \mathbb{R}_{\geq 0}$ is a rate matrix and $G \subseteq S$ is a set of* goal *states.*

For a state $s \in S$ we define the set of *enabled actions* $\mathrm{Act}(s)$ as follows: $\mathrm{Act}(s) = \{\alpha \in \mathrm{Act} \mid \exists s' \in S : \ \mathbf{R}(s, \alpha, s') > 0\}$. States s' for which $\mathbf{R}(s, \alpha, s') > 0$

form the set of *successor states of s via α*, denoted as $\mathrm{Succ}(s, \alpha)$. W. l. o. g. we require that all sets $\mathrm{Act}(s)$ and $\mathrm{Succ}(s, \alpha)$ are non-empty. A state s, s. t. $\forall \alpha \in \mathrm{Act}(s): \mathrm{Succ}(s, \alpha) = \{s\}$ is called *absorbing*.

For a given state s and action $\alpha \in \mathrm{Act}(s)$, we denote by $\lambda(s, \alpha) = \sum_{s'} \mathbf{R}(s, \alpha, s')$ the *exit rate* of α in s and $\Delta(s, \alpha, s') = \mathbf{R}(s, \alpha, s')/\lambda(s, \alpha)$.

An example CTMDP is depicted in Fig. 1a. Here states are depicted in circles and are labelled with numbers from 0 to 5. The goal state G is marked with a double circle. Dashed transitions represent available actions, e.g. state 1 has two enabled actions α and β. A solid transition labelled with a number denotes the rate, e.g. $\mathbf{R}(1, \beta, G) = 1.1$, therefore there is a solid transition from state 1 via action β to state G with rate 1.1. If there is only one enabled action for a state, we only show the rates of the transition via this action and omit the action itself. For example, state 0 has only 1 enabled action (lets say α) and therefore it only has outgoing solid transition with rate $1.1 = \mathbf{R}(0, \alpha, 1)$.

The system starts in the initial state $s_0 = s_{\mathrm{init}}$. While being in a state s_0, the system picks an action $\alpha_0 \in \mathrm{Act}(s)$. When an action is picked the CTMDP resides in s_0 for the amount of time t_0 which is sampled from exponential distribution with parameter $\lambda(s_0, \alpha_0)$. Later in this paper we refer to this as *residence time* in a state. After t_0 time units the system transitions into one of the successor states $s_1 \in \mathrm{Succ}(s_0, \alpha_0)$ selected randomly with distribution $\Delta(s_0, \alpha_0, \cdot)$. After this transition the process is repeated from state s_1 forming an *infinite path* $\rho = s_0 \xrightarrow{\alpha_0, t_0} s_1 \xrightarrow{\alpha_1, t_1} s_2 \ldots$. A finite prefix of an infinite path is called a *(finite) path*. We will use $\rho{\downarrow}$ to denote the last state of a finite path ρ. We will denote the set of all finite paths in a CTMDP with *Paths**, and the set of all infinite paths with *Paths*.

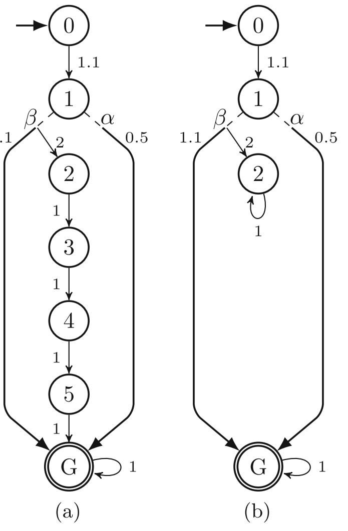


Fig. 1. Example CTMDPs.

CTMDPs pick actions with the help of *schedulers*. A scheduler is a measurable[1] function $\pi : Paths^* \times \mathbb{R}_{\geqslant 0} \to \mathcal{D}(\mathrm{Act})$ such that $\pi(\rho, t) \in \mathrm{Act}(\rho{\downarrow})$. Being in a state s at time point t the CTMDP samples an action from $\pi(\rho, t)$, where ρ is the path that the system took to arrive in s. We denote the set of all schedulers with Π.

Fixing a scheduler π in a CTMDP $\mathcal{M}$, the unique probability measure $\mathrm{Pr}_\pi^{\mathcal{M}}$ over the space of all infinite paths can be obtained [Neu10], denoted also by Pr_π when $\mathcal{M}$ is clear from context.

Optimal Time-Bounded Reachability

Let $\mathcal{M} = (s_{\mathrm{init}}, S, \mathrm{Act}, \mathbf{R}, G)$ be a CTMDP, $s \in S$, $T \in \mathbb{R}_{\geqslant 0}$ a time bound, and $\mathrm{opt} \in \{\sup, \inf\}$. The *optimal (time-bounded) reachability probability* (or *value*) of state s in $\mathcal{M}$ is defined as follows:

$$\mathrm{val}_{\mathcal{M}}^{s}(T) := \mathrm{opt}_{\pi \in \Pi} \, \mathrm{Pr}_\pi^{\mathcal{M}} \left[\Diamond^{\leqslant T} G \right],$$

[1] Measurable with respect to the standard σ-algebra on the set of paths [NZ10].

where $\Diamond^{\leqslant T} G = \{s_0 \xrightarrow{\alpha_0,t_0} s_1 \xrightarrow{\alpha_1,t_1} s_2 \ldots \mid s_0 = s \wedge \exists i : s_i \in G \wedge \sum_{j=0}^{i-1} t_j \leq T\}$ is the set of paths starting from s and reaching G before T.

The *optimal (time-bounded) reachability probability* (or *value*) of $\mathcal{M}$ is defined as $\mathrm{val}_{\mathcal{M}}(T) = \mathrm{val}_{\mathcal{M}}^{s_{\mathrm{init}}}(T)$. A scheduler that achieves optimum for $\mathrm{val}_{\mathcal{M}}(T)$ is the *optimal scheduler.* A scheduler that achieves value v, such that $||v - \mathrm{val}_{\mathcal{M}}(T)||_\infty < \varepsilon$ is called *ε-optimal.*

3 Algorithm

In this work we target CTMDPs that have large state spaces, but only a small subset of those states is actually contributing significantly to the reachability probability.

Consider, for example, the *polling system* represented schematically in Fig. 2. Here two stations store continuously arriving tasks in a queue. Tasks are to be processed by a server. If the task is processed successfully it is removed from the queue, otherwise it is returned back into the queue. State space of the CTMDP $\mathcal{M}$ modelling this polling system is a tuple (q_1, q_2, s), where q_i is the amount of tasks in queue i and s is a state of the server (could be e. g. *processing task, awaiting task*, etc.).

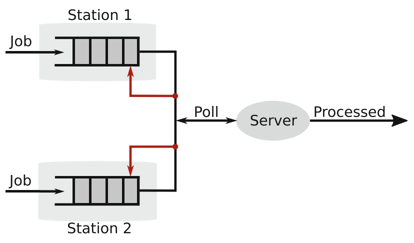


Fig. 2. Schematic representation of polling system

One of the possible questions could be, for example, *what is the maximum probability of both queues to be full after a certain time point.* This corresponds to goal states being of the form (N, N, s), where N is the maximal queue capacity and s – any state of the server. Given that both queues are initially empty, all the paths reaching goal states have to visit states $(q_1, q_2, \cdot)$, where $q_i = [0..N]$. However, for similar questions, for example, *what is the maximum probability of the first queue to be full after a certain time point*, the situation changes. Here goal states are of the form (N, q_2, s), where $q_2 = 0..N$ and s – any state of the server. The scheduler that only extracts tasks from the second queue is the fastest to fill the first one and is therefore the optimal one. The set of states that are most likely visited when following this scheduler are those states where the size of the second queue is small. This naturally depends on the rates of task arrival and processing. Assuming that the size of the queue rarely exceeds 2 tasks, all the states $(\cdot, q_2, \cdot)$, where $q_2 = 3..N$ do not affect the reachability probability too much.

As a more concrete example, consider the CTMDP of Fig. 1a. Here all the states in the center have exit rate 1 and form a long chain. Due to the length of this chain the probability to reach the goal state via these states within time 2 is very small. In fact, the maximum probability to reach the target state within 2 time units in the CTMDP on the left and the one on the right are exactly the same and equal 0.4584. Thus, on this CTMDP, 40% of the state space can be reduced without any effect on the reachability value.

Classical model checking algorithms do not take into account any information about the property and perform exhaustive state-space exploration. Given that only a subset of states is relevant to the reachability value, these algorithms may perform many unnecessary computations.

Our Solution

Throughout this section we work with a CTMDP $\mathcal{M} = (s_{\text{init}}, S, \text{Act}, \mathbf{R}, G)$ and a time bound $T \in \mathbb{R}_{\geqslant 0}$.

The main contribution of this paper is a simple framework for solving the time-bounded reachability objective in CTMDPs without considering their whole state-space. This framework in presented in Algorithm 1. The algorithm involves the following major steps:

Algorithm 1 SubspaceTBR

```
    Input: CTMDP $\mathcal{M} = (s_{\text{init}}, S, \text{Act}, \mathbf{R}, G)$, time bound $T$, precision $\varepsilon$
    Output: $(\ell, u) \in [0,1]^2$ such that $\ell \leqslant \text{val}(T) \leqslant u$ and $u - \ell < \varepsilon$ and
            $\varepsilon-$ optimal scheduler $\pi$ for $\text{val}_{\mathcal{M}}(T)$
 1: if $s_{\text{init}} \in G$ then return $(1,1)$, and an arbitrary scheduler $\pi \in \Pi$
 2: $\ell = 0, u = 1$
 3: $\pi_{\texttt{sim}} = \pi_{\texttt{uniform}}$
 4: $S' = \{s_{\text{init}}\}$
 5: while $u - \ell \geqslant \varepsilon$ do
 6:     $S' = S' \cup$ getRelevantSubset$(\mathcal{M}, T, \pi_{\texttt{sim}})$
 7:     $\underline{\mathcal{M}} = \mathsf{lower}(\mathcal{M}, S')$, $\overline{\mathcal{M}} = \mathsf{upper}(\mathcal{M}, S')$
 8:     $\ell = \text{val}_{\underline{\mathcal{M}}}(T)$, $u = \text{val}_{\overline{\mathcal{M}}}(T)$
 9:     $\overline{\pi}_{\texttt{opt}} \leftarrow$ optimal scheduler for $\text{val}_{\overline{\mathcal{M}}}(T)$, $\underline{\pi}_{\texttt{opt}} \leftarrow$ optimal scheduler for $\text{val}_{\underline{\mathcal{M}}}(T)$
10:     $\pi_{\texttt{sim}} =$ ChooseScheduler$(\pi_{\texttt{uniform}}, \overline{\pi}_{\texttt{opt}})$ // choose a scheduler for simulations
11: $\forall t \in [0,T], \forall s \in S' : \pi(s,t) = \underline{\pi}_{\texttt{opt}}(s,t)$
12: $\forall t \in [0,T], \forall s \in S \setminus S' : \pi(s,t) \leftarrow$ any $\alpha \in \text{Act}(s)$ // extend optimal scheduler to $S$
13: return $(\ell, u), \pi$
```

Step 1 A "relevant subset" of the state-space $S' \subseteq S$ is computed (line 6).

Step 2 Using this subset, CTMDPs $\underline{\mathcal{M}}$ and $\overline{\mathcal{M}}$ are constructed (line 7). We define functions $\mathsf{upper}(\mathcal{M}, S')$ and $\mathsf{lower}(\mathcal{M}, S')$ later in this section.

Step 3 The reachability values of $\underline{\mathcal{M}}$ and $\overline{\mathcal{M}}$ are under- and over-approximations of the reachability value $\text{val}_{\mathcal{M}}(T)$. The values are computed in line 8 along with the optimal schedulers in line 9.

Step 4 Scheduler $\pi_{\texttt{sim}}$, used for obtaining the relevant subset, is selected at line 10.

Step 5 If the two approximations are sufficiently close, i.e. $\text{val}_{\overline{\mathcal{M}}}(T) - \text{val}_{\underline{\mathcal{M}}}(T) < \varepsilon$, $\left[\text{val}_{\underline{\mathcal{M}}}(T), \text{val}_{\overline{\mathcal{M}}}(T)\right]$ is the interval in which the actual reachability value lies. The algorithm is stopped and this interval along with the ε-optimal scheduler are returned. If not, the algorithm repeats from line 6, growing the relevant subset in each iteration.

Algorithm 2 GETRELEVANTSUBSET($\mathcal{M}, T, \pi_{\mathtt{sim}}$)

```
    Input: CTMDP $\mathcal{M} = (s_{\text{init}}, S, \text{Act}, \mathbf{R}, G)$, time bound $T$, a scheduler $\pi_{\mathtt{sim}}$
    Parameters: $\text{n}_{\text{sim}} \in \mathbb{N}$
    Output: $S' \subseteq S$
1: for $(i = 0;\ i < \text{n}_{\text{sim}};\ i = i + 1)$ do
2:     $\rho = s_{\text{init}}$, $t = 0$
3:     while $t < T$ and $\rho{\downarrow} \notin G$ do
4:         $s = \rho{\downarrow}$
5:         Sample action $\alpha$ from distribution $\mathcal{D}(\text{Act}(s)) = \pi_{\mathtt{sim}}(\rho, 0)$
6:         Sample $t'$ from exponential distribution with parameter $\lambda(s, \alpha)$
7:         Sample a successor $s'$ of $s$ with distribution $\Delta(s, \alpha, \cdot)$
8:         $\rho = \rho \xrightarrow{t'} s'$, $t = t + t'$
9:     add all states of $\rho$ to $S'$
```

In the following section, we elucidate these steps and discuss several instantiations and variations of this framework.

3.1 Step 1: Obtaining the Relevant Subset

The main challenge of the approach is to extract a relatively small *representative* set $S' \subseteq S$, for which $\text{val}_{\overline{\mathcal{M}}}(T)$ and $\text{val}_{\underline{\mathcal{M}}}(T)$ are close to the value $\text{val}_{\mathcal{M}}(T)$ of the original model. If this is possible, then instead of computing the probability of reaching goal in $\mathcal{M}$, we can compute the same in $\overline{\mathcal{M}}$ and $\underline{\mathcal{M}}$ to get an ε-width interval in which the actual value is guaranteed to lie. If the sizes of $\overline{\mathcal{M}}$ and $\underline{\mathcal{M}}$ are relatively small, then the computation is generally much faster.

In this work we propose a heuristics for selecting the relevant subset based on simulations. Simulation of continuous-time Markov chains (CTMDPs with singleton set $\text{Act}(s)$ for all states) is a widely used approach that performs very well in many practical cases. It is based on sampling a path of the model according to its probability space. Namely, upon entering a state s the residence time is sampled from the exponential distribution and then the successor state s' is sampled randomly from the distribution $\Delta(s, \alpha, s')$. Here α is the only action available in state s. The process is repeated from state s' until a goal state is reached or the cumulative time over this path exceeds the time-bound.

However this approach only works for fully stochastic processes, which is not the case for arbitrary CTMDPs due to the presence of multiple available actions. In order to make the process fully stochastic one has to fix a scheduler that decides which actions are to be selected during the run of a CTMDP.

Our heuristic is presented in Algorithm 2. It takes as input the CTMDP, time bound and a scheduler $\pi_{\mathtt{sim}}$. The algorithms performs n_{sim} simulations and outputs all the states visited during the execution. Here $\text{n}_{\text{sim}} \in \mathbb{N}$ is a parameter of the algorithm. Each simulation run starts in the initial state. At first an action is sampled from $\mathcal{D}(\text{Act}(s)) = \pi_{\mathtt{sim}}(\rho, 0)$ and then the simulation proceeds in the same way as described above for CTMCs by sampling residence times and successor states. Notice that even though time-point 0 is used for the scheduler, this does not affect the correctness of the approach, since it is only used as a heuristic to sample the subspace. In fact, one could instantiate GETRELEVANTSUBSET with an arbitrary heuristic (e. g. from artificial intelligence domain, or one that is more targeted towards a specific model). Correctness of the lower and upper bounds will not be affected by this. However, termination of the algorithm cannot be ensured for any arbitrary heuristic. Indeed, one has to make sure that the bounds will eventually converge to the value.

Example 1. Consider the CTMDP from Fig. 3a. Figure 3b, c show two possible sampled paths. The path in 3c reaches the target within the given time-bound and the path in 3b times out before reaching the goal state. The relevant subset is thus all the states visited during the two simulations.

3.2 Step 2: Under- and Over-Approximating CTMDP

We will now explain line 7 of Algorithm 1. Here we obtain two CTMDPs, such that the value of $\underline{\mathcal{M}}$ is a guaranteed lower bound, and the value of $\overline{\mathcal{M}}$ is a guaranteed upper bound on the value of $\mathcal{M}$.

Let $S' \subseteq S$ be the subset of states obtained in line 6. We are interested in extracting some information regarding the reachability value of $\mathcal{M}$ from this subset. In order to do this, we consider two cases. (i) A pessimistic case, where all the unexplored states are non-goal states and absorbing (or *sink states*); and (ii) an optimistic case, where all the unexplored states are indeed goals. It is easy to see that the "pessimistic" CTMDP $\underline{\mathcal{M}}$ will have a smaller (or equal) value than the original CTMDP, which in turn will have a value smaller (or equal) than the "optimistic" CTMDP $\overline{\mathcal{M}}$. Notice that for the reachability value the goal states can also be made absorbing and this will not change the value[2]. Before we define the two CTMDPs formally, we illustrate the construction on an example. Note that the fringe "one-step outside" of the relevant subset is still a part of the considered sub-CTMDPs.

Example 2. Let S' be the state space of the CTMDP from Fig. 3a explored in Example 1. Figure 4a depicts the sub-CTMDP obtained by restricting the state space of the original model to S'. Figure 4b, c demonstrate how the "pessimistic" and "optimistic" CTMDPs can be obtained. All the states that are not part of S' are made absorbing for the "pessimistic" CTMDP Fig. 4b and are made goal states for the "optimistic" CTMDP Fig. 4c.

Formally, we define methods $\mathsf{lower}(\mathcal{M}, S')$ and $\mathsf{upper}(\mathcal{M}, S')$ that return the pessimistic and optimistic CTMDP, respectively. The $\mathsf{lower}(\mathcal{M}, S')$ method returns a CTMDP $\underline{\mathcal{M}} = (s_{\text{init}}, \widetilde{S}, \text{Act}, \widetilde{\mathbf{R}}, G)$, where $\widetilde{S} = S' \cup \text{Succ}(S')$, and $\forall s', s'' \in \widetilde{S}$:

$$\widetilde{\mathbf{R}}[s', \alpha, s''] = \begin{cases} \mathbf{R}[s', \alpha, s''] & \text{if } s' \in S' \\ \overline{\lambda} & \text{if } s' \notin S', s'' = s' \\ 0 & \text{otherwise,} \end{cases}$$

[2] This is due to the fact that for the reachability value, only what happens before the first arrival to the goal matters, and everything that happens afterwards is irrelevant.

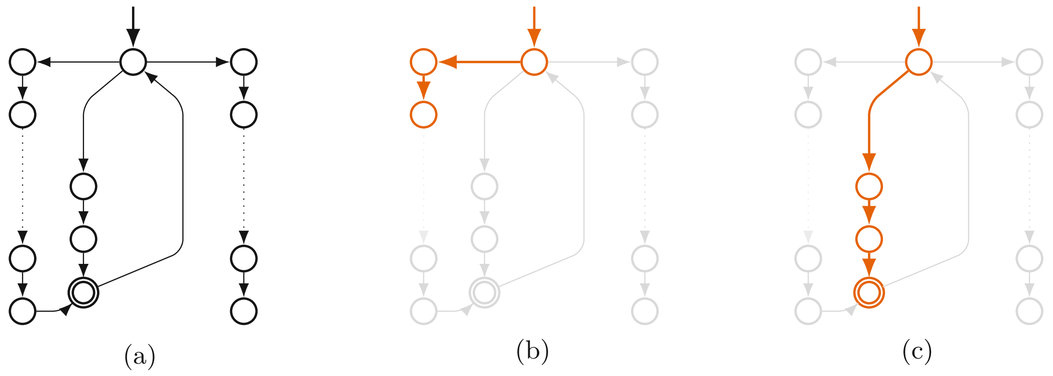


Fig. 3. A simple CTMDP is presented in (a) with rates and action labels ignored. (b) shows a sampled run which ends on running out of time while exploring the left-most branch. (c) shows a simulation which ends on discovering a target state.

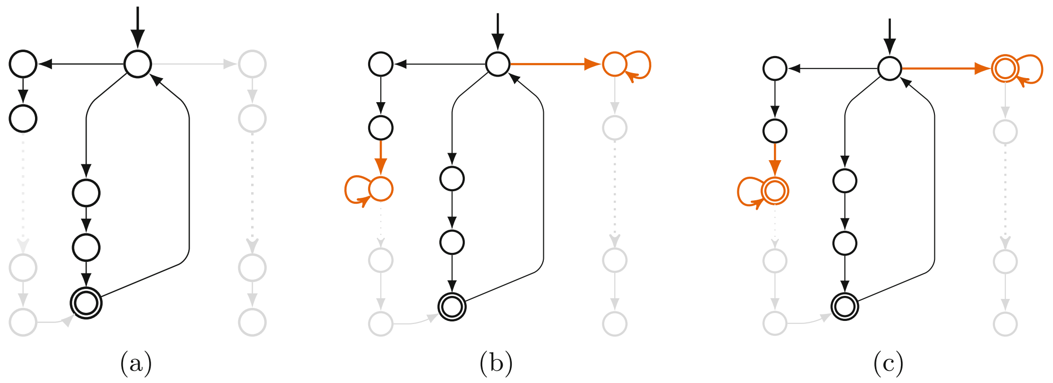


Fig. 4. (a) depicts the relevant subset obtained at line 6 of Algorithm 1. (b, c) show the addition of successors (in highlight) of the states at the fringe. In (b), the appended states are made absorbing by adding a self-loop of rate $\overline{\lambda}$. Meanwhile in (c), the newly added states are made goals.

where $\overline{\lambda}$ is the maximum exit rate in $\mathcal{M}$. And the method $\mathsf{upper}(\mathcal{M}, S')$ returns CTMDP $\overline{\mathcal{M}} = (s_{\text{init}}, \widetilde{S}, \text{Act}, \widetilde{\mathbf{R}}, \overline{G})$, where $\overline{G} = G \cup (\widetilde{S} \setminus S')$, and state space $\widetilde{S}$ and the rate matrix $\widetilde{\mathbf{R}}$ are the same as for $\mathsf{lower}(\mathcal{M}, S')$.

Since many states are absorbing now large parts of the state space may become unreachable, namely all the states that are not in $\widetilde{S}$.

Lemma 1. $\text{val}_{\underline{\mathcal{M}}}(T) \leqslant \text{val}_{\mathcal{M}}(T) \leqslant \text{val}_{\overline{\mathcal{M}}}(T)$

3.3 Step 3: Computing the Reachability Value

Algorithm 1 requires computing the reachability values for CTMDPs $\underline{\mathcal{M}}$ and $\overline{\mathcal{M}}$ (line 9). This can be done by any algorithm for reachability analysis, e. g. [BHHK15,NZ10,HH13,BS11,FRSZ11,BHHK04] which approximate the value up to an arbitrary precision ε. These algorithms usually also compute the ε-optimal scheduler along with the approximation of the reachability value. In the following we will use interchangeably the notions of the value and its ε-approximation, as well as an optimal scheduler and an ε-optimal scheduler.

Notice that some of the algorithms mentioned above compute optimal reachability value only w. r. t. a subclass of schedulers, rather than the full class Π. In this case the result of Algorithm 1 will be the optimal reachability value with respect to this subclass and not class Π.

3.4 Step 4: The Choice of Scheduler $\pi_{\mathtt{sim}}$

At line 10 of Algorithm 1 the scheduler $\pi_{\mathtt{sim}}$ is selected that is used in the subsequent iteration for refining the relevant subset of states. We propose two ways of instantiating the function ChooseScheduler($\pi_{\mathtt{uniform}}, \overline{\pi}_{\mathtt{opt}}$), one with the uniform scheduler $\pi_{\mathtt{uniform}}$, and another with the scheduler $\overline{\pi}_{\mathtt{opt}}$. Depending on the model, its goal states and the time bound one of the options may deliver smaller relevant subset than another:

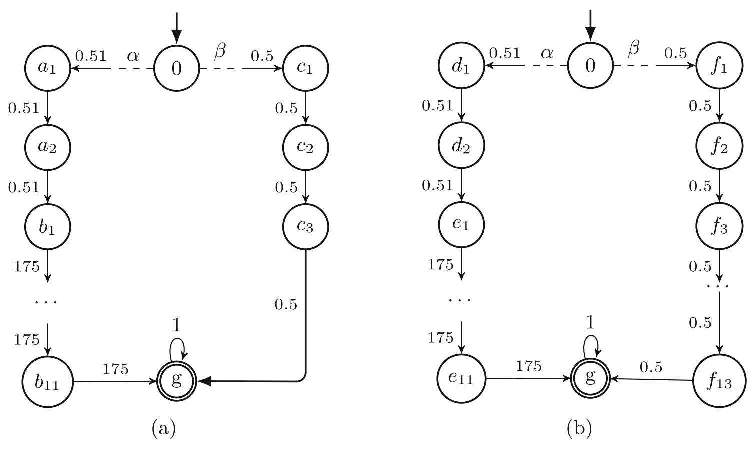


Fig. 5. An example of a CTMDP where the uniform scheduler delivers possibly smaller relevant subset than the optimal scheduler (a), and vice versa (b).

Example 3. Consider, the CTMDP in Fig. 5a and the time bound 3.0. Assuming that the goal state has not yet been sampled from the right and left chains, action α delivers higher reachability value than action β. For example, if states a_1 to a_2 are sampled from the chain on the left and c_1 to c_2 from the chain on the right, the reachability value of the respective over-approximating CTMDP when choosing action β is 0.1987 and when choosing action α is 0.1911. And this situation persists also when states $b_1 - b_{10}$ are sampled due to high exit rates of the respective transitions. However if state b_{11} is sampled, the reachability value when following α becomes 0.1906. Only at this moment the optimal behaviour is to choose action β. However, when following the uniform scheduler, there is a chance that the whole chain on the right is explored before any of the states b_i are visited. If the precision $\varepsilon = 0.01$, then at the moment the goal state is reached via the right chain and at least states a_1 to a_2 are sampled on the left, the algorithm has converged. Thus using the uniform scheduler SubspaceTBR may in fact explore fewer states than when using the optimal one.

Naturally, there are situation when following the optimal scheduler is the best one can do. For example, in the CTMDP in Fig. 5b it is enough to explore only state f_1 on the right to realise that action β is sub-optimal. From this moment on only action α is chosen for simulations, which is in fact the best way to proceed. At the moment the goal state is reached the algorithm has converged for precision 0.01.

One of the main advantages of the uniform scheduler is that it does not require too much memory and is simple to implement. Moreover, since some algorithms to compute time-bounded reachability probability do not provide an optimal scheduler in the classical way as defined in Sect. 2 ([BHHK15]), the use of $\pi_{\texttt{uniform}}$ may be the only option. In spite of its simplicity, in many cases this scheduler generates very succinct state spaces, as we will show in Sect. 4.

Using the uniform scheduler is beneficial in those cases when, for example, different actions of the same state have exit rates that differ drastically, e. g. by an order of magnitude. If the goal state is reachable via actions with high rates, choosing an action with low rate leads to higher residence times (due to properties of the exponential distribution) and therefore fewer states will be reachable within the time bound, compared to choosing an action with a high exit rate. In this case using the uniform scheduler may lead to larger sub-space, compared to using the optimal scheduler. However, the experiments show this difference is typically negligible.

The drawback of the uniform scheduler is that the probability of it choosing each action is positive. Thus it will choose also those actions that are clearly suboptimal and could be omitted during the simulations. The uniform scheduler $\pi_{\texttt{uniform}}$ does not take this information into account while the scheduler $\overline{\pi}_{\texttt{opt}}$ does. The latter is optimal on the sub-CTMDP obtained during the previous iterations. This scheduler will thus pick only those actions that look most promising to be optimal. Using this scheduler may induce smaller sampled state space than the one generated by $\pi_{\texttt{uniform}}$, as we also show in Sect. 4.

Notice that it is possible to alternate between using $\pi_{\texttt{uniform}}$ and $\overline{\pi}_{\texttt{opt}}$ at different iterations of Algorithm 1, for instance, when $\overline{\pi}_{\texttt{opt}}$ is costly to obtain or simulate. However, in our experiments, we always choose either one of the two, with the exception for the first iteration when only the uniform scheduler is available.

3.5 Step 5: Termination and Optimal Schedulers

The algorithm runs as long as the values of $\underline{\mathcal{M}}$ and $\overline{\mathcal{M}}$, as computed in Step 3 are not sufficiently close. It terminates when the difference becomes less than ε. The scheduler $\underline{\pi}_{\texttt{opt}}$ obtained in line 9 is ε-optimal for $\underline{\mathcal{M}}$ since it is obtained by running a standard TBR algorithm on $\underline{\mathcal{M}}$. From this scheduler one can obtain ε-optimal scheduler π for $\mathcal{M}$ itself by choosing the same actions as $\underline{\pi}_{\texttt{opt}}$ on the relevant subset of states (S' in Algorithm 1) and any arbitrary action on other states.

Lemma 2. *Scheduler π computed by Algorithm 1 is ε-optimal.*

Theorem 1. *Algorithm 1 converges almost surely.*

On any CTMDP, if $\pi_{\texttt{sim}} = \pi_{\texttt{uniform}}$, Algorithm 1 will, in the worst case, eventually explore the whole CTMDP. In such a situation, $\underline{\mathcal{M}}$ and $\overline{\mathcal{M}}$ will be the same as $\mathcal{M}$. The algorithm would then terminate since the condition on line 5 would be falsified. If $\pi_{\texttt{sim}} = \overline{\pi}_{\texttt{opt}}$, the system is continuously driven to the fringe as long as the condition on line 5 holds. This is because all unexplored states act as goal states in the upper-bound model. Such a scheduler will eventually explore the state-space reachable by the optimal scheduler on the original model and leave out those parts that are only reachable with suboptimal decisions.

4 Experiments

The framework described in Sect. 3 was evaluated against 5 different benchmarks available in the MAPA[3] language [TKvdPS12]:

Fault Tolerant Work Station Cluster (`ftwc`-n) [HHK00]: models two networks of n workstations each. Each network is interconnected by a switch. The switches communicate via a backbone. All the components may fail and can be repaired only one at a time. The system starts in a fully functioning state and a state is goal if in both networks either all the workstations or the switch are broken.

Google File System (`gfs`-n) [HCH+02, GGL03]: in this benchmark files are split into chunks, each maintained by one of n chunk servers. We fix the number of chunks a server may store to 5000 and the total number of chunks to 10000. The GFS starts in the state where for one of the chunks no replica is stored and the target is to have at least 3 copies of the chunk available.

Polling System (`ps`-j-k-g): We consider the variation of the polling system case [GHH+13, TvdPS13], that consists of j stations and one server. Incoming requests of j types are buffered in queues of size k each, until they are processed by the server and delivered to their station. The system starts in a state with all the queues being nearly full. We consider 2 goal conditions: (i) all the queues are empty (g=`all`) and (ii) one of the queues is empty (g=`one`).

Erlang Stages (`erlang`-k-r): this is a synthetic model with known characteristics [ZN10]. It has two different paths to reach the goal state: a fast but risky path or a slow but sure path. The slow path is an Erlang chain of length k and rate r.

Stochastic Job Scheduling (`sjs`-m-j) [BDF81]: models a multiprocessor architecture running a sequence of independent jobs. It consists of m identical processors and j jobs. As goal we define the states with all jobs completed;

[3] Translated to explicit state format by the tool Scoop [Tim11].

Our algorithm is implemented as an extension to PRISM [KNP11] and we use IMCA [GHKN12] in order to solve the sub-CTMDPs ($\underline{\mathcal{M}}$ and $\overline{\mathcal{M}}$). We would like to remark, however, that the performance of our algorithm can be improved by using a better toolchain than our PRISM-IMCA setup (see [ABHK18, Appendix A.2]).

In order to instantiate our framework, we need to describe how we perform Steps 1 and 3 (Sect. 3). Recall from Sect. 3.1 that we proposed two different schedulers to be used as the simulating scheduler $\pi_{\texttt{sim}}$: the uniform scheduler $\pi_{\texttt{uniform}}$ and the optimal scheduler $\overline{\pi}_{\texttt{opt}}$ obtained by solving $\overline{\mathcal{M}}$.

For Step 3, we select three algorithms for time-bounded reachability analysis: the first discretisation-based algorithm [NZ10] (D), and the two most competitive algorithms according to the comparison performed in [BHHK15], namely the adaptive version of discretization [BS11] (A) and the uniformisation-based [BHHK15] (U). SubspaceTBR instantiated with these algorithms and with $\pi_{\texttt{sim}} = \pi_{\texttt{uniform}}$ is referred to with $\texttt{D}_{\texttt{uni}}$, $\texttt{A}_{\texttt{uni}}$ and $\texttt{U}_{\texttt{uni}}$ respectively. For $\pi_{\texttt{sim}} = \overline{\pi}_{\texttt{opt}}$, the instantiations are referred to as $\texttt{D}_{\texttt{opt}}$, $\texttt{A}_{\texttt{opt}}$ and $\texttt{U}_{\texttt{opt}}$. Since U does not provide the scheduler in a classical form as defined in Sect. 2, we omit $\texttt{U}_{\texttt{opt}}$. We also omit experiments on $\texttt{D}_{\texttt{opt}}$ as our experience with D and $\texttt{D}_{\texttt{uni}}$ suggested that $\texttt{D}_{\texttt{opt}}$ would also run out of time on most experiments.

We compare the performance of the instantiated algorithms with their originals, implemented in IMCA. We set the precision parameter for SubspaceTBR and the original algorithms in IMCA to 0.01. Indicators such as the median model checking time (excluding the time taken to load the model into memory) and explored state-space are measured.

Table 1. An overview of the experimental results along with the state-space sizes. Runtime (in seconds) for the various algorithms are presented. '-' indicates a timeout (1800 s). $\texttt{U}_{\texttt{uni}}$, $\texttt{A}_{\texttt{uni}}$ and $\texttt{A}_{\texttt{opt}}$ perform quite well on erlang, gfs and ftwc while only $\texttt{A}_{\texttt{opt}}$ is better than U and A on the ps-one family of models. ps-4-8-all and sjs are hard instances for both $\pi_{\texttt{uniform}}$ and $\overline{\pi}_{\texttt{opt}}$. D times out on all benchmarks except on sjs due to its small state-space.

Benchmark	States	U	$\texttt{U}_{\texttt{uni}}$	A	$\texttt{A}_{\texttt{uni}}$	$\texttt{A}_{\texttt{opt}}$	D	$\texttt{D}_{\texttt{uni}}$
erlang-10^6-10	1,000k	71	**1**	4	**1**	**1**	-	299
gfs-120	1,479k	-	**2**	-	**2**	**2**	-	-
ftwc-128	597k	251	**10**	114	**11**	**15**	-	-
ps-4-24-one	7,562k	507	-	171	-	**105**	-	-
ps-4-8-all	119k	1,475	-	826	-	-	-	-
sjs-2-9	18k	6	99	2	139	-	1,199	-

Tables 1 and 2 summarize the main results of our experiments. Table 1 reports the runtime of the algorithms on several benchmarks, while Table 2 reports on the state-space complexity. Here the last column refers to the smallest relevant

Table 2. For each benchmark, we report (i) the size of the state-space; (ii) total states explored by our instantiations of SubspaceTBR; (iii) size of the final over-approximating sub-CTMDP $\overline{\mathcal{M}}$; and (iv) size of the relevant subset returned by the greedy search of Sect. 4.1. We use `ps-4-4-one` and `sjs-2-7` instead of larger models in their respective families as running the greedy search is a highly computation-intensive task.

Benchmark	States	Explored		Size of last $\overline{\mathcal{M}}$	Post greedy reduction
		by $\pi_{\texttt{sim}}$	%		
`erlang`-10^6-10	1,000k	559	0.06	561	496
`gfs`-120	1,479k	105	0.01	200	85
`ftwc`-128	597k	296	0.05	858	253
`sjs`-2-7	2k	2,537	93.86	2,704	1,543
`ps`-4-4-`one`	10k	697	6.63	2,040	696
`ps`-4-8-`all`	119k	-	-	-	-
`ps`-4-24-`one`	7,562k	23,309	0.31	-	-

subset of $\overline{\mathcal{M}}$ that we can obtain with reasonable effort. This subset is computed by running the greedy algorithm described in Sect. 4.1 on $\overline{\mathcal{M}}$. It attempts to reduce more states of the explored subset without sacrificing the precision too much. We run the greedy algorithm with a precision of $\varepsilon/10$, where ε is the precision used in SubspaceTBR.

We recall that our framework is targeted towards models which contain a small subset of valuable states. We can categorize the models into three classes:

Easy with Uniform Scheduler ($\pi_{\texttt{sim}} = \pi_{\texttt{uniform}}$)**.** Surprisingly enough, the uniform scheduler performs well on many instances, for example `erlang`, `gfs` and `ftwc`. For `erlang` and `gfs`, it was sufficient to explore a few hundred states no matter how the parameter which increased the state-space was changed (see description of the models above). Here the running time of the instantiations of our framework outperformed the original algorithms due to the fact that less than 1% of the state-space is sufficient to approximate the reachability value up to precision 0.01.

Easy with Optimal Scheduler ($\pi_{\texttt{sim}} = \overline{\pi}_{\texttt{opt}}$)**.** Predictably, there are cases in which uniform scheduler does not provide good results. For example consider the case of `ps`-4-24-`one`. Here the goal condition requires that one of the queues be empty. An action in this benchmark determines the queue from which the task to be processed is picked. Choosing tasks uniformly from different queues, not surprisingly, leads to larger explored state spaces and longer runtimes. Notice that all the instantiations that use uniform scheduler run out of time on this instance. On the other hand, targeted exploration with the most promising scheduler (column $\texttt{A}_{\texttt{opt}}$) performs even better than the original algorithm `A`, finishing within 105 s compared to 171 s and exploring only 0.31% of the state space.

Hard Instances. Naturally there are instances where it is not possible to find a small sub-CTMDP that preserves the properties of interest. For example in `ps`-4-8-`all`, the system is started with all queues being nearly full and the property queried requires all of the queues in the polling system to be empty. As discussed in the beginning of Sect. 3, most of the states of the model have to be explored in order to reach the goal state. In this model there is simply no small sub-CTMDP that preserves the reachability probabilities. As expected, all instantiations timed out and nearly all the states had to be explored. The situation is similar with `sjs`. We identified (using the greedy algorithm in Sect. 4.1) that on some small instances of this model, only 30% to 40% of the state-space can be sacrificed.

Explored State Space and Running Time. In general, as we have mentioned in Sect. 3, the problem is heavily dependent not only on the structure of the model, but also on the specified time-bound and the goal set. Increasing the time-bound for `erlang`, for example, leads to higher probability to explore fully the states of the Erlang chain. This in turn affects the optimal scheduler and for some time-bounds no small sub-CTMDP preserving the value exists.

Naturally, whenever the algorithm explored only a small fraction of the state space, the running time was usually also smaller than the running time of the respective original algorithm. The performance of our framework is heavily dependent on the parameter $\mathrm{n_{sim}}$. This is due to the fact that computation of the reachability value is an expensive operation when performed many times even on small models. Usually in our experiments the amount of simulations was in the order of several thousands. For more details please refer to [ABHK18, Appendix A.2].

4.1 Smallest Sub-CTMDP

In this section, we provide an argument that in the cases where our techniques do not perform well, the reason is not a poor choice of the relevant subsets, but rather that in such cases there are no small subsets which can be removed, at least not such that can be easily obtained. An ideal brute-force method to ascertain this would be to enumerate all subsets of the state space, make the states of the subset absorbing ($\underline{\mathcal{M}}$) or goal ($\overline{\mathcal{M}}$) and then to check whether the difference in values of $\underline{\mathcal{M}}$ and $\overline{\mathcal{M}}$ is ε-close only for small subsets. Unfortunately, this is computationally infeasible. As an alternative, we now suggest a greedy algorithm which we use to search for the largest subset of states one could remove in reasonable time.

The idea is to systematically pick states and observe their effect on the value when they are made absorbing ($\underline{\mathcal{M}}(s)$) or goal ($\overline{\mathcal{M}}(s)$). If a state does not influence the value of the original CTMDP too much, then $\delta(s) = \mathrm{val}_{\overline{\mathcal{M}}(s)}(T) - \mathrm{val}_{\underline{\mathcal{M}}(s)}(T)$ would be small. We first sort all the states in ascending order according to the value $\delta(s)$. And then iteratively build $\underline{\mathcal{M}}$ and $\overline{\mathcal{M}}$ by greedily picking states in this order and making them absorbing (for $\underline{\mathcal{M}}$) and goal (for $\overline{\mathcal{M}}$). The process is repeated until $\text{val}_{\overline{\mathcal{M}}}(T) - \text{val}_{\underline{\mathcal{M}}}(T)$ exceeds ε.

The results of running this algorithm is presented in the right-most column of Table 2. The comparison of the last two columns of the table shows that the portion of the state space our heuristic explored is of the same order of magnitude as what can be obtained with high computational effort. Consequently, this suggests that the surprising choice of the simple uniform scheduler is not poor, but typically indeed achieves the desired degree of reduction.

5 Conclusion

We have introduced a framework for time-bounded reachability analysis of CTMDPs. This framework allows us to run arbitrary algorithms from the literature on a subspace of the original system and thus obtain the result faster, while not compromising its precision beyond a given ε. The subspace is iteratively identified using simulations. In contrast to the standard algorithms, the amount of computation needed reflects not only the model, but also the property to be checked.

The experimental results have revealed that the models often have a small subset which is sufficient for the analysis, and thus our framework speeds up all three considered algorithms. For the exploration, already the uninformed uniform scheduler proves efficient in many settings. However, the more informed scheduler, fed back from the analysis tools, may provide yet better results. In cases where our technique explores the whole state space, our conjecture, confirmed by the preliminary results using the greedy algorithm, is that these models actually do not posses any small relevant subset of states and cannot be exploited by this approach.

This work is agnostic of the structure of the models. Given that states are typically given by a valuation of variables, the corresponding structure could be further utilized in the search for the small relevant subset. A step in this direction could follow the ideas of [PBU13], where discrete-time Markov chains are simulated, the simulations used to infer invariants for the visited states, and then the invariants used to identify a subspace of the original system, which is finally analyzed. An extension of this approach to a non-deterministic and continuous setting could speed up the subspace-identification part of our approach and thus decrease our overhead. Another way to speed up this process is to quickly obtain good schedulers (with no guarantees), e.g. [BBB+17], use them to identify the subspace faster and only then apply a guaranteed algorithm.

References

[ABHK18] Ashok, P., Butkova, Y., Hermanns, H., Křetínský, J.: Continuous-Time Markov Decisions Based on Partial Exploration. ArXiv e-prints (2018). https://arxiv.org/abs/1807.09641

[ACD+17] Ashok, P., Chatterjee, K., Daca, P., Kretínský, J., Meggendorfer, T.: Value iteration for long-run average reward in Markov decision processes. In: CAV (2017)

[ASSB96] Aziz, A., Sanwal, K., Singhal, V., Brayton, R.K.: Verifying continuous time Markov chains. In: CAV (1996)

[BBB+17] Bartocci, E., Bortolussi, L., Brázdil, T., Milios, D., Sanguinetti, G.: Policy learning in continuous-time Markov decision processes using gaussian processes. Perform. Eval. **116**, 84–100 (2017)

[BCC+14] Brázdil, T., et al.: Verification of Markov decision processes using learning algorithms. In: ATVA (2014)

[BDF81] Bruno, J.L., Downey, P.J., Frederickson, G.N.: Sequencing tasks with exponential service times to minimize the expected flow time or makespan. J. ACM **28**(1), 100–113 (1981)

[Ber95] Bertsekas, D.P.: Dynamic Programming and Optimal Control, vol. II. Athena Scientific (1995)

[BFK+09] Brázdil, T., Forejt, V., Krčál, J., Křetínský, J., Kučera, A.: Continuous-time stochastic games with time-bounded reachability. In: FSTTCS (2009)

[BHHK04] Baier, C., Haverkort, B.R., Hermanns, H., Katoen, J.: Efficient computation of time-bounded reachability probabilities in uniform continuous-time Markov decision processes. In: TACAS (2004)

[BHHK15] Butkova, Y., Hatefi, H., Hermanns, H., Krcál, J.: Optimal continuous time Markov decisions. In: ATVA (2015)

[BS11] Buchholz, P., Schulz, I.: Numerical analysis of continuous time Markov decision processes over finite horizons. Comput. OR **38**(3), 651–659 (2011)

[EHKZ13] Eisentraut, C., Hermanns, H., Katoen, J., Zhang, L.: A semantics for every GSPN. In: Petri Nets (2013)

[Fei04] Feinberg, E.A.: Continuous time discounted jump Markov decision processes: a discrete-event approach. Math. Oper. Res. **29**(3), 492–524 (2004)

[FRSZ11] Fearnley, J., Rabe, M., Schewe, S., Zhang, L.: Efficient approximation of optimal control for continuous-time Markov games. In: FSTTCS (2011)

[GGL03] Ghemawat, S., Gobioff, H., Leung, S.: The google file system. In: SOSP (2003)

[GHH+13] Guck, D., Hatefi, H., Hermanns, H., Katoen, J., Timmer, M.: Modelling, reduction and analysis of Markov automata. In: QEST (2013)

[GHKN12] Guck, D., Han, T., Katoen, J., Neuhäußer, M.R.: Quantitative timed analysis of interactive Markov chains. In: NFM (2012)

[HCH+02] Haverkort, B.R., Cloth, L., Hermanns, H., Katoen, J., Baier, C.: Model checking performability properties. In: DSN (2002)

[HH13] Hatefi, H., Hermanns, H.: Improving time bounded reachability computations in interactive Markov chains. In: FSEN (2013)

[HHK00] Haverkort, B.R., Hermanns, H., Katoen, J.: On the use of model checking techniques for dependability evaluation. In: SRDS'00 (2000)

[KNP11] Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 585–591. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_47

[Lef81] Lefèvre, C.: Optimal control of a birth and death epidemic process. Oper. Res. **29**(5), 971–982 (1981)

[MLG05] McMahan, H.B., Likhachev, M., Gordon, G.J.: Bounded real-time dynamic programming: RTDP with monotone upper bounds and performance guarantees. In: ICML (2005)

[Neu10] Neuhäußer, M.R.: Model checking nondeterministic and randomly timed systems. Ph.D. thesis, RWTH Aachen University (2010)

[NZ10] Neuhäußer, M.R., Zhang, L.: Time-bounded reachability probabilities in continuous-time Markov decision processes. In: QEST (2010)

[PBU13] Pavese, E., Braberman, V.A., Uchitel, S.: Automated reliability estimation over partial systematic explorations. In: ICSE, pp. 602–611 (2013)

[QQP01] Qiu, Q., Qu, Q., Pedram, M.: Stochastic modeling of a power-managed system-construction and optimization. IEEE Trans. CAD Integr. Circuits Syst. **20**(10), 1200–1217 (2001)

[Sen99] Sennott, L.I.: Stochastic Dynamic Programming and the Control of Queueing Systems. Wiley-Interscience, New York (1999)

[Tim11] Timmer, M.: Scoop: a tool for symbolic optimisations of probabilistic processes. In: QEST (2011)

[TKvdPS12] Timmer, M., Katoen, J.-P., van de Pol, J., Stoelinga, M.I.A.: Efficient modelling and generation of Markov automata. In: Koutny, M., Ulidowski, I. (eds.) CONCUR 2012. LNCS, vol. 7454, pp. 364–379. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32940-1_26

[TvdPS13] Timmer, M., van de Pol, J., Stoelinga, M.I.A.: Confluence reduction for Markov automata. In: Braberman, V., Fribourg, L. (eds.) FORMATS 2013. LNCS, vol. 8053, pp. 243–257. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40229-6_17

[ZN10] Zhang, L., Neuhäußer, M.R.: Model checking interactive Markov chains. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 53–68. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12002-2_5

Part II

Controller Representation

C SOS: Safe, Optimal and Small Strategies for Hybrid Markov Decision Processes (QEST 2019)

This paper has been published as a **peer reviewed conference paper**.

Pranav Ashok, Jan Kretínský, Kim Guldstrand Larsen, Adrien Le Coënt, Jakob Haahr Taankvist and Maximilian Weininger. 'SOS: Safe, Optimal and Small Strategies for Hybrid Markov Decision Processes'. In: *Quantitative Evaluation of Systems, 16th International Conference, QEST 2019, Glasgow, UK, September 10-12, 2019, Proceedings.* Ed. by David Parker and Verena Wolf. Vol. 11785. Lecture Notes in Computer Science. Springer, 2019, pp. 147–164. DOI: `10.1007/978-3-030-30281-8\_9`. URL: `https://doi.org/10.1007/978-3-030-30281-8%5C_9`

Summary

We consider the problem of synthesizing controllers for cyber-physical systems, modelled using the hybrid Markov decision process formalism. `Uppaal Stratego` is an extension of the popular timed-automata tool `Uppaal` developed at Aalborg University, Denmark, that allows for synthesizing strategies satisfying safety and liveness specifications, as well as optimizing them for cost.

Safe controllers produced by `Uppaal Stratego` come in the form of a lookup table, which is large and incomprehensible. Moreover, the controller needs to be queried a large number of times in the process of optimizing its cost. We introduce `Stratego`$^+$, a framework built on top of `Uppaal Stratego`, making use of decision trees as the data structure of choice to represent these controllers. We use machine learning techniques to learn the decision tree, taking special care to represent the strategy without any loss of information. This allows us to preserve permissiveness as well as the guarantees provided by the original controller. Decision trees already gives orders-of-magnitude improvements in the size of the controller compared to the lookup table. Further, we introduce two techniques to compress the decision trees even more, at the expense of permissiveness. The pipeline also allows to use the reinforcement learning engines of `Uppaal Stratego` to optimize the decision tree controller for some cost. Finally, the

tree-base controller can also be exported as C-code, easily adaptable and implementable on embedded devices.

Contribution

Composition and revision of the manuscript with leading role in writing sections 3 and 4 of the manuscript [AKL+19]. Discussion and development of the ideas, implementation and evaluation with the following notable individual contributions: development of the idea of safe pruning, leading role in implementation of the pipeline and evaluations.

SOS: Safe, Optimal and Small Strategies for Hybrid Markov Decision Processes

Pranav Ashok[1], Jan Křetínský[1], Kim Guldstrand Larsen[2], Adrien Le Coënt[2], Jakob Haahr Taankvist[2], and Maximilian Weininger[1]( )

[1] Technical University of Munich, Munich, Germany
maxi.weininger@tum.de
[2] Aalborg University, Aalborg, Denmark

Abstract. For hybrid Markov decision processes, Uppaal Stratego can compute strategies that are safe for a given safety property and (in the limit) optimal for a given cost function. Unfortunately, these strategies cannot be exported easily since they are computed as a very long list. In this paper, we demonstrate methods to learn compact representations of the strategies in the form of decision trees. These decision trees are much smaller, more understandable, and can easily be exported as code that can be loaded into embedded systems. Despite the size compression and actual differences to the original strategy, we provide guarantees on both safety and optimality of the decision-tree strategy. On the top, we show how to obtain yet smaller representations, which are still guaranteed safe, but achieve a desired trade-off between size and optimality.

1 Introduction

Cyber-physical systems often are safety-critical and hence strong guarantees on their safety are paramount. Furthermore, resource efficiency and the quality of the delivered service are strong requirements; the behaviour needs to be optimized with respect to these objectives, while of course staying within the bounds of what is still safe. In order to achieve this, controllers of such systems can be either implemented manually or automatically synthesized. In the former case, due to the complexity of the system, coming up with a controller that is safe is difficult, even more so with the additional optimization requirement. In the latter case, the synthesis may succeed with significantly less effort, though the requirement on both safety and optimality is still a challenge for current synthesis methods. However, due to the size of the systems, the produced controllers may be very complex, hard to understand, implement, modify, or even just output. Indeed, even for moderately sized systems, we can easily end up with gigabytes-long descriptions of their controllers (in the algorithmic context called strategies).

This research was funded in part by TUM IGSSE project 10.06 (PARSEC), the German Research Foundation (DFG) project KR 4890/2-1 “Statistical Unbounded Verification” and the ERC Advanced Grant LASSO.

D. Parker and V. Wolf (Eds.): QEST 2019, LNCS 11785, pp. 147–164, 2019.
https://doi.org/10.1007/978-3-030-30281-8_9

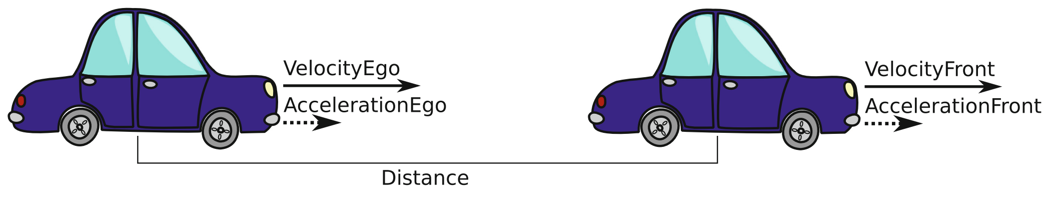


Fig. 1. The two cars, *Ego* and *Front*. We control *Ego* and the environment controls *Front*. Both cars have an acceleration and a velocity. In addition, we know the distance between the cars.

In this paper, we show how to provide a more compact representation, which can yield acceptably short and simple code for resource-limited embedded devices, and consequently can be more easily understood, maintained, modified, debugged, and the requirements are better traceable in the final controller. To this end, as the formalism for the compact representation we choose decision trees [41]. This representation is typically several orders of magnitude smaller than the classical explicit description and also is known for its interpretability and understandability [9,41,47]. The resulting encoded strategy may differ from the original one, but despite that and despite being smaller, it is still guaranteed to be safe and as nearly-optimal as the original one. Moreover, we can trade off additional decrease in size for decrease in performance (getting farther from optimum) to a desired degree, while maintaining safety.

Example 1. As a running example and one of the case studies, we use the following example introduced in [35] and expanded with stronger safety guarantees in [34].

We consider two cars *Ego* and *Front*, depicted in Fig. 1. We control *Ego*, whereas *Front* is controlled by the environment. *Ego* is driving behind *Front*, and both cars have a discrete input (the acceleration) and a continuous state (the velocity). The goal of the adaptive cruise control in *Ego* is, first, to stay safe (by keeping the distance between the cars greater than a given safe distance), and second, to drive as close to *Front* as possible, i.e. to optimize the aggregated distance between the cars.

We use UPPAAL TIGA [2] to get a safe strategy for *Ego* as in [34], and then UPPAAL STRATEGO [18] to learn a (near-)optimal strategy for a desired cost function, given the constraints from the safe strategy. The resulting strategy is output as a list with almost 6 million configurations. Using our new methods, we obtain a decision tree representing the strategy, that has only about 2713 nodes. Additionally, we can trade performance to reduce the size even further, e.g. by increasing the aggregated distance reasonably we can reduce the size to 1247 nodes.

Our Contribution:

– We design and implement a framework STRATEGO^{+} to transform safe and (near-)optimal strategies into their decision-tree representation, preserving safety and the same level of optimality, while being much smaller.

– We provide several transformations and ways to yet further decrease the size while preserving safety, but relaxing the optimality to a desired extent.
– We test our methods on three case studies, where we show size reductions of up to three orders of magnitude, and quantify the additional size-performance trade-off.

Our techniques can be used to represent (finite-memory non-stochastic) strategies for arbitrary systems exhibiting non-determinism (e.g. Markov decision processes, timed/concurrent/stochastic games). This paper demonstrates the technique on hybrid Markov decision processes, as that is the formalism used in Uppaal Stratego.

Related Work: The problem of computing strategies for hybrid systems has been extensively studied in the past years. Most approaches rely on abstraction techniques: the continuous and infinite state space of the system is represented with a finite number of symbols, *e.g.* discrete points [24,50], sets of states [15], etc. However, it is still hard to deal with uncontrollable components, even though some approaches exist such as robust control [26], or contract-based design [51], but they usually consider the uncontrollable component as a bounded perturbation and do not tackle stochastic behaviour. The tool PESSOA [38,48] can synthesize controllers for cyber-physical systems represented by a set of smooth differential equations with a specification in a fragment of Linear Temporal Logic (LTL). Abstraction techniques are used in [27] for synthesizing strategies for a class of hybrid systems that involve random phenomena together with discrete and continuous behaviours. Discrete, stochastic dynamical systems are considered in [54], where the synthesis of strategies with respect to LTL objectives is made possible with an abstraction-refinement method. In [22] a number of benchmarks for hybrid system verification has been proposed, including a room heating benchmark. In [16] Uppaal SMC was applied to the performance evaluation of several strategies proposed in the benchmark. However, there was no focus on safety in this approach. In our work, the safety strategy synthesis relies on a discretization of the continuous variables, leading to a decidable problem that can be handled by Uppaal Tiga, but we furthermore provide safety guarantees for the original system with the use of a Timed Game abstraction based on a guaranteed Euler scheme [36].

In artificial intelligence, compact (factored) representations of Markov decision processes (MDPs) have been developed using dynamic Bayesian networks [7,31], probabilistic STRIPS [33], algebraic decision diagrams [30], and also decision trees [7]. For a detailed survey of compact representations see [5]. Formalisms used to represent MDPs can, in principle, be used to represent strategies as well. In particular, variants of decision trees are probably the most used [7,13,32]. Decision trees have been also used in connection with real-time dynamic programming and reinforcement learning [6,44].

In the context of verification, MDPs are often represented using variants of (MT) BDDs [19,28,39], and strategies by BDDs [55]. Learning a compact decision-tree representation of a strategy has been investigated in [37] for the case of body sensor networks, in [9] for finite (discrete) MDPs, and in [10] for finite games, but only with Boolean variables. Moreover, these decision trees can only predict a single action for a state configuration whereas in this work, we allow the trees to predict more than one action for a single configuration. In control theory, [56] proves that the problem of computing size-optimal determinisiation of controllers is NP-complete and hence discuss various heuristic-based determinisation algorithms. None of these works consider the optimization aspect, which being a soft constraint enables the trade-offs.

Permissive strategies have been studied in e.g. [3,8,20].

2 Preliminaries

2.1 Hybrid Markov Decision Processes

We describe the mathematical modelling framework. The correspondence to the Uppaal models is straightforward.

Definition 1 (HMDP). *A* hybrid Markov decision process *(HMDP) $\mathcal{M}$ is a tuple (C, U, X, F, δ) where:*

1. *the controller C is a finite set of (controllable) modes $C = \{c_1, \ldots, c_k\}$,*
2. *the uncontrollable environment U is a finite set of (uncontrollable) modes $U = \{u_1, \ldots, u_l\}$,*
3. *$X = (x_1, \ldots, x_n)$ is a finite tuple of continuous (real-valued) variables,*
4. *for each $c \in C$ and $u \in U$, $F_{c,u} : \mathbb{R}_{>0} \times \mathbb{R}^X \to \mathbb{R}^X$ is the flow-function that describes the evolution of the continuous variables over time in the combined mode (c, u), and*
5. *δ is a family of probability functions $\delta_\gamma : U \to [0, 1]$, where $\gamma = (c, u, \boldsymbol{x})$ is a global configuration. More precisely, $\delta_\gamma(u')$ is the probability that u in the global configuration $\gamma = (c, u, \boldsymbol{x})$ will change to the uncontrollable mode u'.*

In the following, we denote by $\mathbb{C}$ the set of global configurations $C \times U \times \mathbb{R}^X$ of the HMDP $\mathcal{M}$. The above notion of HMDP actually describes an infinite-state Markov Decision Process [43], where choices of mode for the controller is made periodically and choice of mode for the uncontrollable environment is made probabilistically according to δ. Note that abstracting δ_γ to the support $\hat{\delta_\gamma} = \{u \mid \delta_\gamma(u) > 0\}$, turns $\mathcal{M}$ into a (traditional) hybrid two-player game. The inclusion of δ allows for a probabilistic refinement of the uncontrolled environment in this game. Such a refinement is irrelevant for the purposes of guaranteeing safety; however, it will be useful for optimizing the cost of operating the system. Indeed, rather than optimizing only the worst-case performance, we wish to optimize the overall expected behaviour.

Strategies. A – memoryless and possibly non-deterministic – strategy σ for the controller C is a function $\sigma : \mathbb{C} \to 2^C$, i.e. given the current configuration $\gamma = (c, u, \boldsymbol{x})$, the expression $\sigma(\gamma)$ returns the set of allowed *actions* in that configuration; in our setting, the actions are the controllable modes to be used for the duration of the next period. Non-deterministic strategies are also called permissive since they permit many actions instead of prescribing one.

The evolution of system over time is defined as follows. Let $\gamma = (c, u, \boldsymbol{x})$ and $\gamma' = (c', u', \boldsymbol{x}')$. We write $\gamma \xrightarrow{\tau} \gamma'$ in case $c' = c, u' = u$ and $\boldsymbol{x}' = F_{(c,u)}(\tau, \boldsymbol{x})$.

A *run* is an interleaved sequence $\pi \in \mathbb{C} \times (\mathbb{R} \times \mathbb{C} \times \mathbb{C} \times \mathbb{C})^*$ of configurations and relative time-delays of some given period P:

$$\pi = \gamma_o :: P :: \alpha_1 :: \beta_1 :: \gamma_1 :: P :: \alpha_2 :: \beta_2 :: \gamma_2 :: P :: \cdots$$

Then π is a *run according to the strategy* σ if after each period P the following sequence of discrete (instantaneous) changes are made:

1. the value of the continuous variables are updated according to the flow of the current mode, i.e. $\gamma_{i-1} = (c_{i-1}, u_{i-1}, \boldsymbol{x}_{i-1}) \xrightarrow{P} (c_{i-1}, u_{i-1}, \boldsymbol{x}_i) =: \alpha_i$;
2. the environment changes to any possible new mode, i.e. $\beta_i = (c_{i-1}, u_i, \boldsymbol{x}_i)$ where $\delta_{\alpha_i}(u_i) > 0$;
3. the controller changes mode according to the strategy σ, i.e. $\gamma_i = (c_i, u_i, \boldsymbol{x}_i)$ with $c_i \in \sigma(\beta_i)$.

Safety. A strategy σ is said to be *safe* with respect to a set of configuration $S \subseteq \mathbb{C}$, if for any run π according to σ all configurations encountered are within S, i.e. $\alpha_i, \beta_i, \gamma_i \in S$ for all i and also $\gamma_i' \in S$ whenever $\gamma_i \xrightarrow{\tau} \gamma_i'$ with $\tau \leq P$. Note that the notion of safety does not depend on the actual δ, only on its supports. Recall that almost-sure safety, i.e. with probability 1, coincides with sure safety.

We use a guaranteed set-based *Euler method* introduced in [34] to ensure safety of a strategy not only at the configurations where we make decisions, but also in the continuum in between them. We refer the reader to [1, Appendix A.2] for a brief reminder of this method.

Optimality. Under a given *deterministic* (i.e. permitting one action in each configuration) strategy σ the game $\mathcal{M}$ becomes a completely stochastic process $\mathcal{M} \upharpoonright \sigma$, inducing a probability measure on sets of runs. In case σ is *non-deterministic* or *permissive*, the non-determinism in $\mathcal{M} \upharpoonright \sigma$ is resolved uniformly at random. On such a process, we can evaluate a given optimization function. Let $H \in \mathbb{N}$ be a given time-horizon, and D a random variable on runs, then $\mathbb{E}^{\mathcal{M},\gamma}_{\sigma,H}(D) \in \mathbb{R}_{\geq 0}$ is the expected value of D on the space of runs of $\mathcal{M} \upharpoonright \sigma$ of length[1] H starting in the configuration γ. As an example of D, consider the integrated deviation of a continuous variable, e.g. distance between *Ego* and *Front*, with respect to a given target value.

[1] Note that there is a bijection between length of the run and time, as the time between each step, P, is constant.

Consequently, given a (memoryless non-deterministic) safety strategy σ_{safe} with respect to a given safety set S, we want to find a deterministic sub-strategy[2] σ_{opt} that optimizes (minimizes or maximizes) $\mathbb{E}^{\mathcal{M},\gamma}_{\sigma_{safe},H}(D)$.

2.2 Decision Trees

From the perspective of machine learning, *decision trees* (DT) [41] are a classification tool assigning classes to data points. A data point is a d-dimensional vector $\boldsymbol{v} = (v_1, v_2, \ldots, v_d)$ of features with each v_i drawing its value from some set D_i. If D_i is an ordered set, then the feature corresponding to it is called *ordered* or *numerical* (e.g. *velocity* $\in \mathbb{R}$) and otherwise, it is called *categorical* (e.g. *color* $\in \{red, green, blue\}$). A (multi-class) DT can represent a function $f\colon \prod_{i=1}^{d} D_i \to A$ where A is a finite set of classes.

A (single-label) DT over the domain $D = \prod_{i=1}^{d} D_i$ with labels A is a tuple $\mathcal{T} = (T, \rho, \theta)$, where T is a finite binary tree, ρ assigns to every inner node predicates of the form $x_i \sim c$ where $\sim \ \in \{\leq, =\}$, $c \in D_i$, and θ assigns to every leaf node a list of natural numbers $[m_1, m_2, \ldots, m_{|A|}]$. For every $\boldsymbol{v} \in D$, there exists a *decision path* from the root node to some leaf $\ell_{\boldsymbol{v}}$. We say that $\boldsymbol{v}$ satisfies a predicate $\rho(t)$ if $\rho(t)$ evaluates to true when its variables are evaluated as given by $\boldsymbol{v}$. Given $\boldsymbol{v}$ and an inner node t with a predicate $\rho(t)$, the decision path selects either the *left* or *right* child of t based on whether $\boldsymbol{v}$ satisfies $\rho(t)$ or not. For $\boldsymbol{v}$ from the training set, we say that the leaf node $\ell_{\boldsymbol{v}}$ *contains* $\boldsymbol{v}$. Then m_a of a leaf is the number of points contained in the leaf and classified a in the training set. Further, the classes assigned by a DT to a data point $\boldsymbol{v}$ (from or outside of the training set) are given by $\arg\max \theta(\ell_{\boldsymbol{v}}) = \{i \mid \forall i, j \leq |A|.\ \theta(\ell_{\boldsymbol{v}})_i \geq \theta(\ell_{\boldsymbol{v}})_j\}$, i.e. the most frequent classes in the respective leaf.

Decision trees may also predict sets of classes instead of a single class. Such a generalization (representing functions of the type $\prod_{i=1}^{d} D_i \to 2^A$) is called a *multi-label* decision tree. In these trees, θ assigns to every leaf node a list of tuples $[(n_1, y_1), (n_2, y_2), \ldots, (n_{|A|}, y_{|A|})]$ where $n_a, y_a \in \mathbb{N}$ are the number of data points in the leaf *not* labelled by class a and labelled by class a, respectively. The (multi-label) classification of a data point is then typically given by the majority rule, i.e. it is classified as a if $n_a < y_a$.

A DT may be constructed using decision-tree learning algorithms such as ID3 [45], C4.5 [46] or CART [11]. These algorithms take as input a training set, i.e. a set of vectors whose classes are already known, and output a DT classifier. The tree constructions start with a single root node containing all the data points of the training set. The learning algorithms explore all possible predicates $p = x_i \sim c$, which split the data points of this node into two sets, X_p and $X_{\neg p}$. The predicate that minimizes the sum of entropies[3] of the two sets is selected. These sets are added as child nodes to the node being split and the whole process

[2] i.e. a strategy that for every configuration returns a (non-strict) subset of the actions allowed by the safe strategy.

[3] Entropy of a set X is $H(X) = \sum_{a \in A} p_a \log_2(p_a) + (1 - p_a) \log_2(1 - p_a)$, where p_a is the fraction of samples in X belonging to class a. See [14] for more details.

is repeated, splitting each node further and further until the entropy of the node becomes 0, i.e. all data points belong to the same class. Such nodes are called *pure* nodes. This construction is extended to the multi-label setting by some of the algorithms. A multi-label node is called *pure* if there is at least one class that is endorsed by all data points in that node, i.e. $\exists a \in A : n_a = 0$.

If the tree is grown until all leaves have zero entropy, then the classifier memorizes the training data exactly, leading to *overfitting* [41]. This might not be desirable if the classifier is trained on noisy data or if it needs to predict classes of unknown data. The learning algorithms hence provide some parameters, known as *hyperparameters*, which may be tuned to generalize the classifier and improve its accuracy. Overfitting is not an issue in our setup where we want to learn the strategy function (almost) precisely. However, we can use the hyperparameters to produce even smaller representations of the function, at the "expense" of not being entirely precise any more. One of the hyperparameters of interest in this paper is the minimum split size k. It can be used to stop splitting nodes once the number of data points in them become smaller than k. By setting larger k, the size of the tree decreases, usually at the expense of increasing the entropy of the leaves. There also exist several pruning techniques [21,40], which remove either leaves or entire subtrees after the construction of the DT.

2.3 Standard Uppaal Stratego Workflow

The process of obtaining an optimized safe strategy σ_{opt} using Uppaal Stratego is depicted as the grey boxes in Fig. 2. First, the HMDP $\mathcal{M}$ is abstracted into a 2-player (non-stochastic) timed game $\mathcal{TG}$, ignoring any stochasticity of the behaviour. Next, Uppaal Tiga is used to synthesize a safe strategy $\sigma_{safe}\colon \mathbb{C} \to 2^C$ for $\mathcal{TG}$ and the safety specification φ, which is specified using a simplified version of timed computation tree logic (TCTL) [2]. After that, the safe strategy is applied on $\mathcal{M}$ to obtain $\mathcal{M} \upharpoonright \sigma_{safe}$. It is now possible to perform reinforcement learning on $\mathcal{M} \upharpoonright \sigma_{safe}$ in order to learn a sub-strategy σ_{opt} that will optimize a given quantitative cost, given as any run-based expression containing e.g. discrete variables, locations, clocks, hybrid variables. For more details, see [17,18].

3 Stratego$^+$

In this section, we discuss the new Uppaal Stratego$^+$ framework following with each of its components are elucidated.

3.1 New Workflow

Uppaal Stratego$^+$ extends the standard workflow in two ways: Firstly, in the top row, we generate the DT $\mathcal{T}_{opt}$ that exactly represents σ_{opt}, yielding a small representation of the strategy.

The DT learning algorithm can make use of two (hyper-)parameters k and p which may be used to prune the DT; this approach is described in Sect. 3.4.

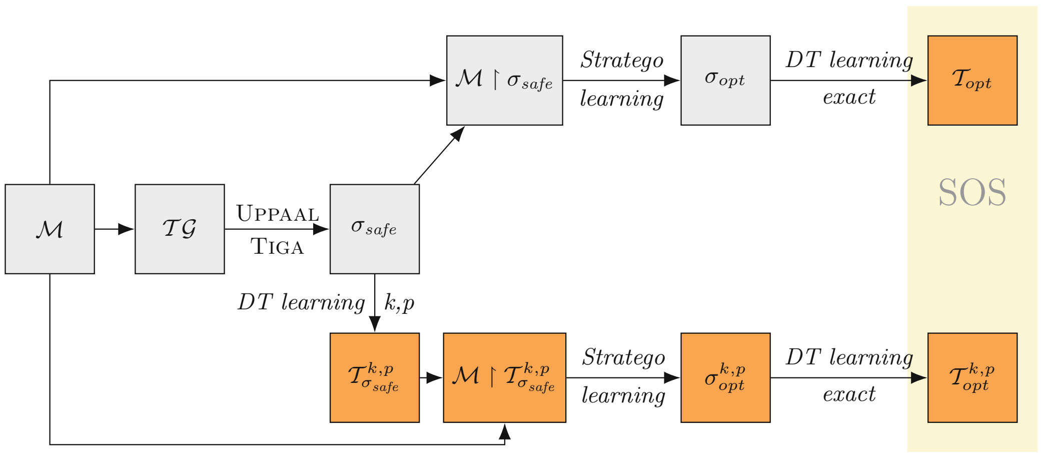


Fig. 2. Uppaal Stratego$^+$ workflow. The dark orange nodes are the additions to the original workflow, which now involve DT learning, the yellow-shaded area delimits the desired safe, optimal, and small strategy representations. (Color figure online)

While pruning reduces the size of the DT, the resultant tree no longer represents the strategy exactly. Hence it is not possible to prune a DT representing deterministic strategies, like in the case of the σ_{opt} described in the first row of the workflow, as safety would be violated.

However, for our second extension we apply the DT learning algorithm to the non-deterministic, permissive strategy σ_{safe}, resulting in $\mathcal{T}^{k,p}_{\sigma_{safe}}$. This DT is less permissive, thereby smaller, since the pruning disallows certain actions; yet it still represents a safe strategy (details in Sect. 3.4). Next, as in the standard workflow, this less permissive safe strategy is applied to the game and Stratego is used to get a near-optimal strategy $\sigma^{k,p}_{opt}$ for the modified game $\mathcal{M} \upharpoonright \mathcal{T}^{k,p}_{\sigma_{safe}}$. In the end, we again construct a DT exactly representing the optimal strategy, namely $\mathcal{T}^{k,p}_{opt}$. Note that in the game restricted to $T^{k,p}_{\sigma_{safe}}$ fewer actions are allowed than when it is restricted only to σ_{safe}, and hence the resulting strategy could perform worse. For example, let σ_{safe} allow decelerating or remaining neutral for some configuration, while $T^{k,p}_{\sigma_{safe}}$ pruned the possibility to remain neutral and only allows decelerating. Thus, σ_{opt} remains neutral, whereas $\sigma^{k,p}_{opt}$ has to decelerate and thereby increase the distance that we try to minimize.

In both cases, the resulting DT is safe by construction since we allow the DT to predict only pure actions (actions allowed by all configurations in a leaf, see next section for the formal definition). We convert these trees into a nested if-statements code, which can easily be loaded onto embedded systems.

3.2 Representing Strategies Using DT

A DT with domain $\mathbb{C}$ and labels C can learn a (non-deterministic) strategy $\sigma\colon \mathbb{C} \to 2^C$. The strategy is provided as a list of tuples of the form $(\gamma, \{a_1, \ldots, a_k\})$, where γ is a global configuration and $\{a_1, \ldots, a_k\}$ is the set of actions permitted by σ.The training data points are given by the integer configurations $\gamma \in \mathbb{C}$ (safety for non-integer points is guaranteed by the Euler method; see Sect. 2.1) and the set of

distance	velocity	actions
2	51	{*dec*}
3	20	{*dec*}
5	30	{*dec*}
7	1	{*dec*, *neu*}
20	46	{*dec*, *neu*}
25	25	{*dec*, *neu*, *acc*}
45	70	{*dec*, *neu*}

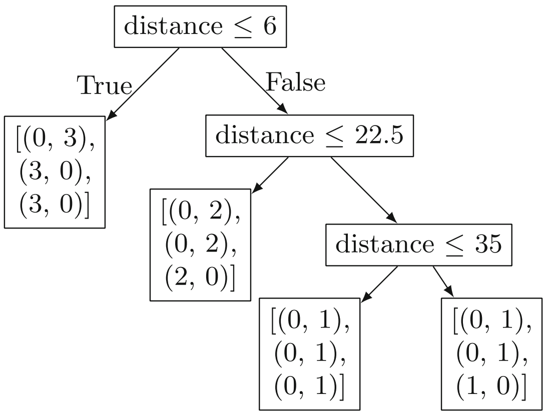


Fig. 3. A sample dataset (left); and a (multi-label) decision tree generated from the dataset (right). The leaf nodes contain the list of tuples assigned by θ, the inner nodes contain the predicates assigned by ρ

classes for each γ is given by $\sigma(\gamma)$. Consequently, a multi-label decision tree learning algorithm as described in Sect. 2.2 can be run on this dataset to obtain a tree $\mathcal{T}_\sigma$ representing the strategy σ.

Each node of the tree contains the set of configurations that satisfy the decision path traced from the root of the tree to the node. The leaf attribute θ gives, for each action a, the number of configurations in the leaf where the strategy disallows and allows a, respectively. For example, consider a node with 10 configurations with $\theta = [(0, 10), (2, 8), (9, 1)]$. This means that the first action is allowed by all 10 configurations in the node, the second action is disallowed by 2 configurations and allowed by 8, and the third action is disallowed by 9 configurations and allowed only by 1.

Since we want the DT to exactly represent the strategy, we need to run the learning algorithm until the entropy of all the leaves becomes 0, i.e. all configurations of the leaf agree on every action. More formally, given a leaf ℓ with n configurations we require $\theta(\ell) = (0, n)$ or $\theta(\ell) = (n, 0)$ for every action. We call an action that all configurations allow a *pure action*.

The table on the left of Fig. 3 shows a toy strategy. Based on values of distance d and velocity v, it permits a subset of the action set $\{dec, neu, acc\}$. A corresponding DT encoding is displayed on the right of Fig. 3.

3.3 Interpreting DT as Strategy

To extract a strategy from a DT, we proceed as follows: Given a configuration C, we pick the leaf ℓ_C associated with it by evaluating the predicates and following a path through the DT. Then we compute $\theta(\ell_C) = [(n_1, y_1), (n_2, y_2), \dots, (n_{|A|}, y_{|A|})]$ where $n_a, y_a \in \mathbb{N}$ are the number of data points in the leaf *not* labelled by class a and labelled by class a, respectively. The classes assigned to ℓ_C are exactly its pure actions, i.e. $\{a \mid (0, y_a) \in \theta(\ell_C)\}$.

Note that allowing only pure actions is necessary in order to preserve safety. We do not follow the common (machine learning) way of assigning classes to the nodes based on the majority criterion, i.e. the majority of the data points in that node allow the action; because then the decision tree might prescribe unsafe actions just because they were allowed in most of the configurations in the node. This is also the reason why the DT-learning algorithm described in the previous section needs to run until the entropy of all leaves becomes 0.

3.4 Learning Smaller, Yet Safe DT

We now describe how to learn a DT for a safe strategy that is smaller than the exact representation, but still preserves safety. A tree obtained using off-the-shelf DT learning algorithms is unlikely to exactly represent the original strategy.[4] We use two different methods to achieve the goal: firstly, we use the standard hyperparameter named *minimum split size*, and secondly, we introduce a new post-processing algorithm called *safe pruning*. Both methods rely on the given strategy being non-deterministic/permissive, i.e. permitting several actions in a leaf.

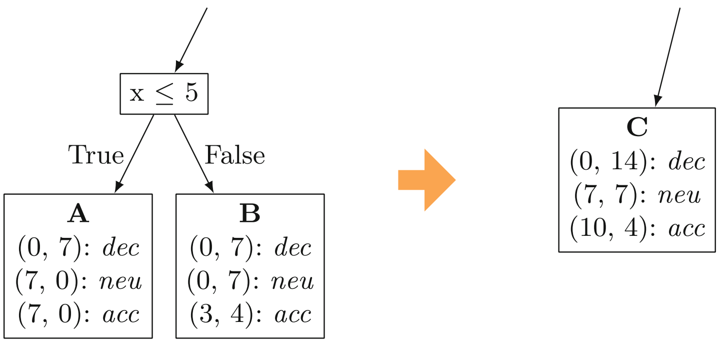


Fig. 4. Illustration of safe pruning applied to a node. The pure action of leaf **A** is just *dec*, for **B** it is both *dec* and *neu*. Safe pruning replaces the nodes with **C**, where only *dec* is a pure action.

(1) Using Minimum Split Size. The splitting process can be stopped before the entropy becomes 0. We do this by introducing a parameter k, which determines the minimum number of data points required in a node to consider splitting it further. During the construction of the tree, a node is usually split if its entropy is greater than 0. When k is set to an integer greater than 2, a node is split only if both the entropy is greater than 0 *and* the number of data points (configurations) in the node is at least k. The strategy given by such a tree is safe as long as it predicts only pure actions, i.e. a with $n_a = 0$. In order to obtain a fully expanded tree, k may be set to 2 (in nodes with <2 configurations,

[4] This is because DT learning algorithms are usually configured to avoid overfitting on the dataset.

there is nothing to split). For larger k, the number of pure actions in the leaves decreases. Ultimately, for too large k, we would obtain a tree that has some leaf nodes not containing *any* pure actions. In such a case, the strategy represented by the DT would not be well-defined, as for some data point no action could be picked. However, this can be detected immediately during the construction.

Algorithm 1. Safe Pruning

```
1:  procedure SAFE-PRUNING(DT $\mathcal{T}_\sigma = (T, \rho, \theta)$, $p \in \mathbb{N}$)
2:      for $i \leftarrow 1..p$ do
3:          $N \leftarrow \{n \in T \mid LEFT(n)$ and $RIGHT(n)$ are leaves$\}$
4:                                                  ▷ Candidate nodes for pruning
5:          for each $n \in N$ do
6:              $c_\ell \leftarrow LEFT(n)$, $c_r \leftarrow RIGHT(n)$
7:              if $\theta(c_\ell) \cap \theta(c_r) \neq \emptyset$ then
8:                                   ▷ Prune and keep the common classification
9:                  Convert $n$ to a leaf node
10:                 $\theta(n) \leftarrow \theta(c_\ell) \cap \theta(c_r)$
11:                 Remove $c_\ell$ and $c_r$ from $T$
```

(2) Using Safe Pruning. Another way of obtaining a smaller tree is by using a procedure to prune the leaves of the produced tree by merging them while preserving safety. For example, consider the decision node on the left of Fig. 4 with two children that are leaves **A** and **B**. For **A**, only the action *dec* is pure (i.e. allowed by all configurations in the leaf), while for **B** both *dec* and *neu* are pure. Since the sets of pure actions of the two leaf nodes intersect, we can safely remove both **A** and **B** and replace the decision node with a new leaf node **C** that contains only those actions that are in the intersection, in this case only *dec*.

Algorithm 1 describes the pruning process formally. If θ returns only safe actions, then the tree obtained after pruning is guaranteed to represent a safe strategy, although a less-permissive one. The algorithm may be run for multiple (possibly 0) rounds, denoted by p, at most until we get a "fully pruned" tree representing a safe but deterministic strategy. We denote by $\mathcal{T}^{k,p}_{\sigma_{safe}}$ the decision tree for σ_{safe} constructed by only splitting nodes with k or more data points, followed by p rounds of safe pruning. Clearly, the more permissive the original strategy is, the more we can prune using safe pruning.

When generating $T^{k,p}_{\sigma_{safe}}$, we use a modified implementation of the CART decision tree learning algorithm implemented in the `DecisionTreeClassifier` class of the Python-based machine learning library Scikit-learn [42]. Since we construct the DT from a safe strategy and as long as we let the DT-encoded strategy have at least one pure action in each leaf, the strategy will remain safe. With this in mind, we can freely change the parameters of the `DecisionTreeClassifier` class. However, in our experiments, we picked only the minimum split size k from the Scikit-parameters as a demonstrative example, as well as our newly introduced p. The methods described in this paper would work with other parameters as well.

3.5 Comparing DTs to Binary Decision Diagrams

A Binary Decision Diagram (BDD, e.g. [12]) is a popular data structure that can be used to represent boolean functions $f\colon \mathbb{B}^n \to \mathbb{B}$. It may also be used to represent strategies by encoding configurations and actions into a suitable form via bit-blasting, i.e. converting them into propositional formulae. For example, the configuration-action pair $((x = 6, y = 2), a_0)$ can be represented as $(x_2 \wedge x_1 \wedge \neg x_0 \wedge \neg y_2 \wedge y_1 \wedge \neg y_0 \wedge a_0)$, if it is known that the maximum value that x and y can take is less than 8 (3 bits). A strategy can be seen as a disjunction $\bigvee_{\gamma, a \in \sigma(\gamma)} (\gamma, a)$ of all configuration-action pairs (γ, a) permitted by the strategy σ. Such an encoding allows for an easy conversion into a BDD. Though theoretically straightforward, there are some practical concerns involved when constructing the BDD. Mainly, the ordering of the variables in the BDD can drastically change its size. While computing the optimal ordering so as to have the smallest BDD is an NP-complete problem [4], various heuristics exist that can be used to get better orderings. We use the CUDD package [53] to construct the BDD, along with Rudell's Sifting reordering technique [49].

The main disadvantage of DTs compared to BDDs is that isomorphic subgraphs are not merged (DTs are trees, BDDs are directed acyclic graphs); and even if merging was allowed, it would not save much. Indeed, since DT may choose different predicates on the same level (which is an advantage in contrast to BDD with a fixed variable ordering) isomorphic subgraphs occur rarely. There are further advantages of DT, related to learning, that make them more compact than BDD in some contexts, e.g. [9,10]. Firstly, they can be learnt fast, using the entropy-based heuristic, compared to the graph processing and variable reordering of BDDs. Secondly, a DT can ignore "don't-care inputs"; these inputs are encodings of things that are not valid configuration-action pairs, in the sense that either the action is not available in the configuration or that it is not a valid configuration at all. In contrast, a BDD has to explicitly either allow or disallow these inputs. Thirdly, DT learning can also be used to represent the strategy imprecisely using a smaller DT, which can be model checked for safety. For the modifications described in Sect. 3.4, we do not even need to re-verify safety, because this property is preserved by both our size reduction techniques. Fourthly, DT can use much wider class of predicates, compared to single bit tests for a bit representation in a BDD. This final point is also a reason (together with the smaller size) why DT is a more understandable representation than a BDD [9,10]. We also illustrate this point on a case-study in Remark 1.

4 Case Studies and Experimental Results

In this section, we evaluate the techniques discussed above on three different case studies: (1) the adaptive cruise control model introduced in the motivation; (2) a two tank case study introduced in [29]; and (3) the heating system of a two room apartment adapted from [25].

Table 1 compares representations for our case studies obtained in different ways. We discuss results for the three case studies, denoted `cruise`, `twotanks`,

Table 1. Sizes of the different representations: explicit list as output by Uppaal Stratego, the relevant part of the list, BDD displaying [minimum/median/maximum] over the 40 trials, and DT according to the upper path in Fig. 2.

	#Variables	Stratego list	List	BDD[min/med/max]	DT $\mathcal{T}_{opt}$ Size
`cruise`$_{\texttt{non-Euler}}$	5	1,790,034	308,216	[3,718/5,066/5,890]	2,899
`cruise`	7	5,931,154	304,752	[3,470/4,728/4,742]	2,713
`twotanks`	9	23,182	23,182	[65/69/91]	1
`tworooms`	11	1,924,708	509,715	[16,370/20,214/25,909]	487

Table 2. Tables displaying the number $|\mathcal{T}_{opt}^{k,p}|$ of nodes of $\mathcal{T}_{opt}^{k,p}$ (left) and the expected performance $\mathbb{E}_{\sigma,H}^{\mathcal{M},\gamma}(D)$ (right) for various k and p, i.e. using the bottom path of Fig. 2, for the `cruise` model. Higher performance corresponds to a lower number. (Color table online)

Min split size (k)	Rounds of pruning (p) 0	1	2
2	2,713	1,725	1,267
10	2,705	1,733	1,249
20	2,667	1,733	1,131
30	2,657	1,695	993
40	2,627	1,669	1,015
50	2,557	1,695	1,003
60	2,635	1,489	963
70	2,613	1,441	955
80	2,519	1,537	915
90	2,455	1,323	923
100	1,929	1,023	877

Min split size (k)	Rounds of pruning (p) 0	1	2
2	2,627	3,618	4,240
10	2,696	3,596	4,210
20	2,778	3,625	14,039
30	2,778	3,589	14,108
40	2,778	3,600	14,096
50	2,825	3,614	14,037
60	2,905	3,673	14,074
70	2,898	3,714	14,095
80	2,907	3,717	14,092
90	3,006	3,741	14,077
100	3,030	14,061	14,292

and `tworooms` respectively. Additionally, the first line displays `cruise` without the integrated Euler method, to illustrate the effect of Euler method on the final size. All the representations are safe and as optimal as σ_{opt} produced by Uppaal Stratego.

For each of the models we display the following information: the third column lists the number of items in the explicit list representation of σ_{opt} output by Uppaal Stratego. The fourth column lists the number of those items that are actually relevant, i.e. sets of configurations where an actual decision is to be made. The fifth and sixth column list the sizes of BDD and DT representations learnt from σ_{opt}, i.e. the upper path in Fig. 2. For BDDs, since the initial ordering plays a role in the size of the final result despite applying the re-ordering heuristics, we ran 40 experiments for each model with random initial variable orderings. For creating BDDs, we used the free Python library *tulip-control/dd* as an interface to CUDD.

We conclude that both BDDs and DTs reduce the size by several order of magnitude. DTs are slightly better in all cases, and 2 orders of magnitude smaller in the tworooms model. Note that reliably achieving good results when

Table 3. Tables displaying the number $|\mathcal{T}_{opt}^{k,p}|$ of nodes of $\mathcal{T}_{opt}^{k,p}$ (left) and the expected performance $\mathbb{E}_{\sigma,H}^{\mathcal{M},\gamma}(D)$ (right) for various k and p, i.e. using the bottom path of Fig. 2, for the `tworooms` model. Higher performance corresponds to a lower number. (Color table online)

Min split size (k)	Rounds of pruning (p) 0	1	2	3
2	543	403	283	191
10	525	387	271	185
50	497	365	251	171
125	445	317	219	151
250	387	265	179	123
500	323	211	139	97
750	277	175	111	77

Min split size (k)	Rounds of pruning (p) 0	1	2	3
2	2,096	2,353	2,821	3,156
10	2,156	2,460	3,285	3,283
50	1,989	2,778	3,287	3,281
125	2,374	2,053	3,280	3,284
250	2,283	2,071	3,288	3,282
500	2,563	2,155	3,280	3,282
750	2,333	2,210	3,279	3,286

constructing the BDD relies on repeating the construction several times; since already constructing a single BDD and applying the heuristics [49] already took roughly 10 times longer than DT learning, DT can be obtained one or two orders of magnitude faster than BDDs, depending on how many times one tries constructing the BDD. Further, for the two tanks, only DT realizes that the strategy is actually trivial. The main reason for BDD not to spot this is the point of ignoring "don't-care" inputs addressed in Sect. 3.5.

Table 2 shows how the size of the DT can be further reduced by the bottom path of Fig. 2, when the "exact representation" criterion is relaxed. It displays the performance, i.e. the aggregated distance to *Front* car, and size of $\mathcal{T}_{opt}^{k,p}$ for different combinations of the pruning parameters k and p. Recall that using no pruning ($k = 2, p = 0$) yields the same DT as the upper path of Fig. 2, i.e. $\mathcal{T}_{opt}^{2,0} = \mathcal{T}_{opt}$.

We observed that for `cruise`, increasing the values of k and p buys a reduction in size of the DT against a reduction in performance. For instance, using $k = 80, p = 0$, one can decrease the size to 2485 (by 8.4%) while deteriorating the performance to 2907 (by 10%). Allowing for half the performance (double the aggregated distance), one can make the DT even smaller than half of its original size, e.g. by setting $k = 10, p = 2$. The shading and colouring of the table display different "trade-off zones", each with comparable savings/losses. The same conclusions hold for $\texttt{cruise}_{\texttt{non-Euler}}$, see the similar Table in [1, Appendix A.3]. For `tworooms` (Table 3), the best performance is observed not with $k = 2, p = 0$, but with $k = 50, p = 0$. We conjecture that the less permissive safe strategy assists STRATEGO in performing the optimisation faster by reducing the size of the search space. As a result, here we get a both smaller and more performant strategy. In the case of `twotanks`, already $\mathcal{T}_{opt}$ has only a single node, hence no further reductions are possible.

Remark 1. Interestingly, domain knowledge can reduce the DT size further and make the representation more understandable. Indeed, for the `cruise` model we were able to construct a DT with only 25 nodes, designing our predicates based on the car kinematics. For example, the expected time until the front car reaches minimal velocity if it only decelerates from now on (1) plays an important role in the decision making and (2) can be easily expressed by solving the standard kinematics equation $v(t) = v_{\text{current}} - a_{\text{dec}} \cdot t$. The resulting DT (illustrated in [1, Appendix A.4] is thus very small and easy to interpret, as each of the few nodes has a clear kinematic interpretation. The DT thus open the possibility for strategy representation to profit from predicate/invariant synthesis.

5 Conclusion

We have provided a framework for producing small representations of safe and (near-)optimal strategies, without compromising safety. As to (near-)optimality, we can choose between two options: (i) not compromising it, or (ii) finding a suitable trade-off between compromising it (causing drops of performance) and additional size reductions. Compared to the original sizes, we achieve orders-of-magnitude reductions, allowing for efficient usage of the strategies in e.g. embedded devices. Compared to BDD representation, the size of the DT representation is smaller and can be computed faster; additionally trivial solutions are represented by trivial DTs. DTs are more readable as argued in [9,10].

A detailed examination of the latter point in the hybrid context remains future work. Further, candidates for more complex predicates could be automatically generated based on given domain knowledge or learnt from the data similarly to invariants from program runs [23,52]. As illustrated in Remark 1, this could lead to further reduction in size and improved understandability. Additionally, isomorphic/similar subtrees could be merged as in decision diagrams and further optimizations for algebraic decision diagrams [56] could be employed. Finally, we plan to visualize the DT representation of the strategies directly in Uppaal Stratego$^+$ for convenience of the users.

References

1. Ashok, P. Křetínský, J., Larsen, K.G., Coënt, A.L., Taankvist, J.H., Weininger, M.: SOS: Safe, optimal and small strategies for hybrid Markov decision processes. Technical report (2019)
2. Behrmann, G., Cougnard, A., David, A., Fleury, E., Larsen, K.G., Lime, D.: UPPAAL-Tiga: time for playing games!. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 121–125. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73368-3_14
3. Bernet, J., Janin, D., Walukiewicz, I.: Permissive strategies: from parity games to safety games. ITA **36**, 261–275 (2002)
4. Bollig, B., Wegener, I.: Improving the variable ordering of OBDDs is NP-complete. IEEE Trans. Comput. **45**(9), 993–1002 (1996)
5. Boutilier, C., Dean, T.L., Hanks, S.: Decision-theoretic planning: structural assumptions and computational leverage. J. Artif. Intell. Res. **11**, 1–94 (1999)

6. Boutilier, C., Dearden, R.: Approximating value trees in structured dynamic programming. In: ICML (1996)
7. Boutilier, C., Dearden, R., Goldszmidt, M.: Exploiting structure in policy construction. In: IJCAI (1995)
8. Bouyer, P., Markey, N., Olschewski, J., Ummels, M.: Measuring permissiveness in parity games: mean-payoff parity games revisited. In: Bultan, T., Hsiung, P.-A. (eds.) ATVA 2011. LNCS, vol. 6996, pp. 135–149. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24372-1_11
9. Brázdil, T., Chatterjee, K., Chmelík, M., Fellner, A., Křetínský, J.: Counterexample explanation by learning small strategies in Markov decision processes. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 158–177. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_10
10. Brázdil, T., Chatterjee, K., Křetínský, J., Toman, V.: Strategy representation by decision trees in reactive synthesis. In: Beyer, D., Huisman, M. (eds.) TACAS 2018. LNCS, vol. 10805, pp. 385–407. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89960-2_21
11. Breiman, L.: Classification and Regression Trees. Routledge, Abingdon (2017)
12. Bryant, R.E.: Symbolic manipulation of boolean functions using a graphical representation. In: DAC (1985)
13. Chapman, D., Kaelbling, L.P.: Input generalization in delayed reinforcement learning: an algorithm and performance comparisons. In: IJCAI. Morgan Kaufmann (1991)
14. Clare, A., King, R.D.: Knowledge discovery in multi-label phenotype data. In: De Raedt, L., Siebes, A. (eds.) PKDD 2001. LNCS (LNAI), vol. 2168, pp. 42–53. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44794-6_4
15. Coënt, A.L., Sandretto, J.A.D., Chapoutot, A., Fribourg, L.: An improved algorithm for the control synthesis of nonlinear sampled switched systems. Formal Methods Syst. Design **53**(3), 363–383 (2018)
16. David, A., Du, D., Larsen, K.G., Mikucionis, M., Skou, A.: An evaluation framework for energy aware buildings using statistical model checking. Sci. China Inform. Sci. **55**(12), 2694–2707 (2012)
17. David, A., et al.: On time with minimal expected cost!. In: Cassez, F., Raskin, J.-F. (eds.) ATVA 2014. LNCS, vol. 8837, pp. 129–145. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11936-6_10
18. David, A., Jensen, P.G., Larsen, K.G., Mikučionis, M., Taankvist, J.H.: Uppaal stratego. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 206–211. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_16
19. de Alfaro, L., Kwiatkowska, M., Norman, G., Parker, D., Segala, R.: Symbolic model checking of probabilistic processes using MTBDDs and the kronecker representation. In: Graf, S., Schwartzbach, M. (eds.) TACAS 2000. LNCS, vol. 1785, pp. 395–410. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-46419-0_27
20. Dräger, K., Forejt, V., Kwiatkowska, M., Parker, D., Ujma, M.: Permissive controller synthesis for probabilistic systems. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 531–546. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54862-8_44
21. Esposito, F., Malerba, D., Semeraro, G.: Decision tree pruning as a search in the state space. In: Brazdil, P.B. (ed.) ECML 1993. LNCS, vol. 667, pp. 165–184. Springer, Heidelberg (1993). https://doi.org/10.1007/3-540-56602-3_135
22. Fehnker, A., Ivančić, F.: Benchmarks for hybrid systems verification. In: Alur, R., Pappas, G.J. (eds.) HSCC 2004. LNCS, vol. 2993, pp. 326–341. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24743-2_22

23. Garg, P., Löding, C., Madhusudan, P., Neider, D.: ICE: a robust framework for learning invariants. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 69–87. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_5
24. Girard, A.: Controller synthesis for safety and reachability via approximate bisimulation. Automatica **48**(5), 947–953 (2012)
25. Girard, A.: Low-complexity quantized switching controllers using approximate bisimulation. Nonlinear Anal.: Hybrid Syst. **10**, 34–44 (2013)
26. Girard, A., Martin, S.: Synthesis for constrained nonlinear systems using hybridization and robust controllers on simplices. IEEE Trans. Automat. Control **57**(4), 1046–1051 (2012)
27. Hahn, E.M., Norman, G., Parker, D., Wachter, B., Zhang, L.: Game-based abstraction and controller synthesis for probabilistic hybrid systems. In: QEST (2011)
28. Hermanns, H., Kwiatkowska, M.Z., Norman, G., Parker, D., Siegle, M.: On the use of mtbdds for performability analysis and verification of stochastic systems. J. Log. Algebr. Program. **56**(1–2), 23–67 (2003)
29. Hiskens, I.A.: Stability of limit cycles in hybrid systems. In: HICSS (2001)
30. Hoey, J., St-Aubin, R., Hu, A., Boutilier, C.: SPUDD: stochastic planning using decision diagrams. In: UAI (1999)
31. Kearns, M., Koller, D.: Efficient reinforcement learning in factored MDPs. In: IJCAI (1999)
32. Koller, D., Parr, R.: Computing factored value functions for policies in structured MDPs. In: IJCAI (1999)
33. Kushmerick, N., Hanks, S., Weld, D.: An algorithm for probabilistic least-commitment planning. In: AAAI (1994)
34. Larsen, K.G., Le Coënt, A., Mikučionis, M., Taankvist, J.H.: Guaranteed control synthesis for continuous systems in Uppaal Tiga. In: Chamberlain, R., Taha, W., Törngren, M. (eds.) CyPhy/WESE -2018. LNCS, vol. 11615, pp. 113–133. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-23703-5_6
35. Larsen, K.G., Mikučionis, M., Taankvist, J.H.: Safe and optimal adaptive cruise control. In: Meyer, R., Platzer, A., Wehrheim, H. (eds.) Correct System Design. LNCS, vol. 9360, pp. 260–277. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23506-6_17
36. Coënt, A.L., De Vuyst, F., Chamoin, L., Fribourg, L.: Control synthesis of nonlinear sampled switched systems using Euler's method. In: SNR (2017)
37. Liu, S., Panangadan, A., Talukder, A., Raghavendra, C.S.: Compact representation of coordinated sampling policies for body sensor networks. In: 2010 IEEE Globecom Workshops (2010)
38. Majumdar, R., Render, E., Tabuada, P.: Robust discrete synthesis against unspecified disturbances. In: HSCC (2011)
39. Miner, A., Parker, D.: Symbolic representations and analysis of large probabilistic systems. In: Baier, C., Haverkort, B.R., Hermanns, H., Katoen, J.-P., Siegle, M. (eds.) Validation of Stochastic Systems. LNCS, vol. 2925, pp. 296–338. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24611-4_9
40. Mingers, J.: An empirical comparison of pruning methods for decision tree induction. Mach. Learn. **4**, 227–243 (1989)
41. Mitchell, T.M.: Machine Learning. McGraw-Hill, Inc., New York (1997)
42. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., VanderPlas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E.: Scikit-learn: machine learning in Python. J. Mach. Learn. Res. **12**, 2825–2830 (2011)

43. Puterman, M.L.: Markov Decision Processes. Wiley, Hoboken (1994)
44. Pyeatt, L.D.: Reinforcement learning with decision trees. Appl. Inform. 26–31 (2003)
45. Quinlan, J.R.: Induction of decision trees. Mach. Learn. **1**, 81–106 (1986)
46. Quinlan, J.R.: C4.5: Programs for Machine Learning. Elsevier, Amsterdam (2014)
47. Riddle, P.J., Segal, R., Etzioni, O.: Representation design and brut-force induction in a boeingmanufacturing domain. Appl. Artif. Intell. **8**, 125–147 (1994)
48. Roy, P., Tabuada, P., Majumdar, R.: Pessoa 2.0: a controller synthesis tool for cyber-physical systems. In: HSCC (2011)
49. Rudell, R.: Dynamic variable ordering for ordered binary decision diagrams. In: CAD (1993)
50. Rungger, M., Zamani, M.: Scots: a tool for the synthesis of symbolic controllers. In: HSCC (2016)
51. Saoud, A., Girard, A., Fribourg, L.: On the composition of discrete and continuous-time assume-guarantee contracts for invariance. In: ECC (2018)
52. Sharma, R., Gupta, S., Hariharan, B., Aiken, A., Nori, A.V.: Verification as learning geometric concepts. In: Logozzo, F., Fähndrich, M. (eds.) SAS 2013. LNCS, vol. 7935, pp. 388–411. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38856-9_21
53. Somenzi, F.: CUDD: CU decision diagram package-release 2.4. 2 (2009). http://vlsi.colorado.edu/~fabio/CUDD
54. Svoreňová, M., Křetínskỳ, J., Chmelík, M., Chatterjee, K., Černá, I., Belta, C.: Temporal logic control for stochastic linear systems using abstraction refinement of probabilistic games. Nonlinear Anal.: Hybrid Syst. **23**, 230–253 (2017)
55. Wimmer, R., et al.: Symblicit calculation of long-run averages for concurrent probabilistic systems. In: QEST (2010)
56. Zapreev, I.S., Verdier, C., Mazo, M.: Optimal symbolic controllers determinization for BDD storage. In: ADHS (2018)

D dtControl: Decision Tree Learning Algorithms for Controller Representation (HSCC 2020)

This paper has been published as a **peer reviewed conference paper**.

Pranav Ashok, Mathias Jackermeier, Pushpak Jagtap, Jan Kretínský, Maximilian Weininger and Majid Zamani. 'dtControl: decision tree learning algorithms for controller representation'. In: *HSCC '20: 23rd ACM International Conference on Hybrid Systems: Computation and Control, Sydney, New South Wales, Australia, April 21-24, 2020.* Ed. by Aaron Ames, Sanjit A. Seshia and Jyotirmoy Deshmukh. ACM, 2020, 17:1–17:7. DOI: `10.1145/3365365.3382220`. URL: `https://doi.org/10.1145/3365365.3382220`

Summary

Inspired by the promising results of Paper C [AKL+19], in this paper, we introduce an extensible toolkit `dtControl` with pipelines not only to `Uppaal Stratego`, but also to the cyber-physical systems synthesis tool `SCOTS`. While the pipeline of [AKL+19] could only handle certain algorithms and models, `dtControl` allows users to synthesize any controller using `Uppaal Stratego` or `SCOTS` and obtain a decision tree for it. Further, in addition to the standard decision tree learning algorithm, `dtControl` also allows users to construct decision trees with linear predicates obtained via logistic regression, linear support vector machines (SVMs), or OC1 [MKS+93]. Moreover, `dtControl` also supports multiple impurity measures to choose between predicates. We also develop a novel determinizing strategy MaxFreq, that produces deterministic controllers much smaller than traditional determinizing strategies such as picking the action with the minimum norm or randomly resolving the non-determinism. Experimental evaluation show that, for many controllers, lookup tables containing millions of state-action pairs can be reduced to trees with tens of nodes using MaxFreq.

`dtControl` has been made available as an open-source software along with detailed user and developer documentation at dtcontrol.model.in.tum.de.

Contribution

Composition and revision of the manuscript. Discussion and development of the ideas, implementation and evaluation with the following individual contributions: laying groundwork for the concept of the tool and co-leading its development, establishing collaboration with the Hybrid Control Systems group and developers of `SCOTS`, creation of the repeatability evaluation artefact, user manual and website.

`dtControl`: Decision Tree Learning Algorithms for Controller Representation

Pranav Ashok
Mathias Jackermeier
Pushpak Jagtap
Jan Křetínský
Maximilian Weininger
Technical University of Munich
Munich, Germany

Majid Zamani
University of Colorado Boulder
Boulder, USA
Ludwig Maximilian University of Munich
Munich, Germany

ABSTRACT

Decision tree learning is a popular classification technique most commonly used in machine learning applications. Recent work has shown that decision trees can be used to represent provably-correct controllers concisely. Compared to representations using lookup tables or binary decision diagrams, decision trees are smaller and more explainable. We present `dtControl`, an easily extensible tool for representing memoryless controllers as decision trees. We give a comprehensive evaluation of various decision tree learning algorithms applied to 10 case studies arising out of correct-by-construction controller synthesis. These algorithms include two new techniques, one for using arbitrary linear binary classifiers in the decision tree learning, and one novel approach for determinizing controllers during the decision tree construction. In particular the latter turns out to be extremely efficient, yielding decision trees with a single-digit number of decision nodes on 5 of the case studies.

CCS CONCEPTS

• **Computer systems organization** → **Embedded and cyber-physical systems**; • **Computing methodologies** → **Classification and regression trees**; **Control methods**.

KEYWORDS

Controller representation, Decision tree, Machine learning, Symbolic control, Non-uniform quantizer, Explainability, Invariance entropy

ACM Reference Format:
Pranav Ashok, Mathias Jackermeier, Pushpak Jagtap, Jan Křetínský, Maximilian Weininger, and Majid Zamani. 2020. `dtControl`: Decision Tree Learning Algorithms for Controller Representation. In *23rd ACM International Conference on Hybrid Systems: Computation and Control (HSCC '20), April 22–24, 2020, Sydney, NSW, Australia.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3365365.3382220

1 INTRODUCTION

Formal synthesis of controllers enforcing complex specifications on cyber-physical systems has gained significant attention in the last few years. This is mainly due to the need for obtaining formally verified control strategies rendering some complex tasks; these are usually represented using temporal logic specifications or (in)finite strings over automata. There are several techniques and tools available that provide automated, correct-by-construction, controller synthesis for cyber-physical systems by utilizing symbolic models (a.k.a. finite abstractions) [5, 36], in which the uncountable continuous states and inputs are aggregated to finite symbolic states and inputs via quantization (a.k.a. discretization). The so-called symbolic controllers are then computed by utilizing algorithmic machinery from computer science and then mapped back for use in the original systems. The state-of-the-art tools to synthesize such controllers are, e.g., `SCOTS` [32], `pFaces` [18], `QUEST` [15], `Pessoa` [16], `CoSyMA` [24], or Uppaal Stratego [12]. These tools give a huge list of state-action pairs (a.k.a. lookup tables) representing controllers.

Storing these symbolic controllers in the memory is a major problem because they usually need to run on embedded devices with limited memory. However, if we do not store the controllers as lookup tables, but take advantage of decision trees (DT) [23], which exploit their hidden structure to represent them in a more compact way, we can mitigate this problem. As shown in [3], DTs can be orders of magnitude smaller than lookup tables. Such a concise representation opens the door for better readability, understandability, and explainability of the controllers, while reducing memory requirements and preserving correctness guarantees. Moreover, human-understandable controllers may also provide insight into the models themselves, thus aiding their validation, as we illustrate in the example below.

Our setting is inherently different from the usual use of DT in machine learning; there, in order to generalize well, DTs typically do not fit the training data exactly; in contrast, in this work, DTs have to exactly represent the given controllers in order to preserve their correctness guarantee. Therefore, our requirements on DTs differ: beside the size and the explainability, it is also the *perfect fitting*. Consequently, it is necessary to thoroughly re-evaluate current DT-learning algorithms and possibly also modify them.

A basic technique used to represent controllers more concisely is to *determinize* them, i.e. to make them not (maximally) permissive but only retain a single action for each state. To this end, one can use, for instance, the action with the minimum norm from

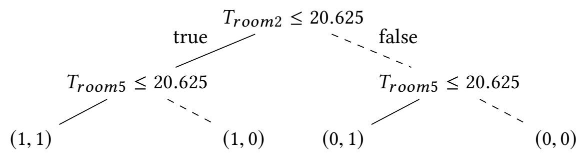


Figure 1: Decision tree for the temperature controller

a reference input, when least energy consuming controllers are preferred [22], or the previously applied action (if possible), when lazy controllers are preferred [16, 24]. Such a size reduction by determinization can be applied as *pre-processing before* learning the DT representation of the controller, typically yielding also a smaller DT. Alternatively, one can apply other kinds of reduction by determinization as *post-processing after* constructing the DT. For instance, in "safe pruning" of [3], the DT constructed for the maximally permissive controller is modified as follows. The leaves of the tree are merged in a bottom-up fashion, thereby reducing the size and partially determinizing it. In contrast, here we introduce a novel approach for determinizing the controllers *during* the construction of the DT, with advantages to both pre-processing and post-processing methods. Firstly, since the choice of the action for each state greatly affects the size and structure of the DT, it is advantageous to guide the choice by the concrete, already built part of the DT, compared to a-priori choices made by pre-processing approaches. Secondly, while the post-processing approaches have to construct a large tree first, our new technique constructs an already reduced tree, avoiding the intermediate large one, thus making it more scalable.

Motivating Example. Consider a temperature control system running in a building with 10 rooms with the heater installed only in 2 rooms as described in [15]. The permissive controller maintaining the temperatures of all the rooms within a certain range obtained using SCOTS is a lookup table with 52,488 state-action pairs. By naively determinizing, we get a lookup table with 26,244 symbolic states (i.e. domain of the controller) and their respective actions. The standard DT-learning, e.g. [9], applied to these two lookup tables yields DT with 8,648 and with 2,703 decision nodes, respectively. While this is an improvement, it is far from being explainable. With the help of our novel determinization strategy presented in Section 4.2, we are able to obtain the decision tree with only 3 (!) decision nodes, see Figure 1. Apart from obtaining a compact and easily implementable controller representation while preserving correctness guarantees, the result is so small that it is immediately explainable and, moreover, allows us to improve on the implementation: one can readily see that we only need to install temperature sensors in two rooms instead of all 10 rooms, which will help users to reduce the system deployment cost as well as the required bandwidth to transfer the state information to the controller. Only 4 symbols (leaves of the tree) need to be transferred to realize the controller.

We also obtain a controller with very few nodes for the cruise-control model of [21]. From such a clear representation one immediately notices that the controller makes the car decelerate when the car in front of it is far away. This counter-intuitive behaviour has thus revealed a bug in the model, which did not actually describe the intended behaviour of the system.

The contribution of this paper can be summarized as follows:

- We present `dtControl`, an open-source tool to convert formally verified controllers to decision trees preserving their correctness guarantees. `dtControl` has a simple input format and already supports automated conversion for controllers generated by two state-of-the-art tools – Uppaal Stratego [12] and SCOTS [32]. It supports several output formats, most importantly the graphical output as DOT files, useful for further analysis and visual presentation, and the C source code, useful for closed-loop simulation or for loading onto embedded devices.
- We introduce a new technique for using arbitrary binary classifiers in the DTs and a novel approach for determinizing controllers during the DT learning. Our approach is tuned towards obtaining extremely small, explainable DTs. In 5 out of 8 case studies where it is applicable (the original controllers are non-deterministic), it produces trees with single-digit numbers of decision nodes.
- We present a comprehensive evaluation of 8 DT-learning algorithms on 10 case studies.

Related Work. DTs [23, Chapter 3] are a well-known class of data structures, particularly known for their interpretability, used mostly by machine learning practitioners in classification or regression tasks. Our work is based on well-known algorithms for decision tree learning, namely CART [9], C4.5 [30] and OC1 [25].

There has been previous work on combining decision trees with classifiers, namely Perceptrons [37], Logistic Regression models [19], piece-wise functions [27] or Support-Vector Machines [1, 11]. We generalize those approaches by allowing for arbitrary binary classifiers to be used in our trees. Additionally, those methods are either restricted to only use two labels, which is not applicable for controllers with more than two possible actions, or they only allow linear classifiers in leaf nodes [1, 27]. In contrast, our approach is applicable with an arbitrary number of actions and also leverages the power of linear classifiers in inner nodes.

An alternative to DTs are binary decision diagrams (BDD) [10]. As seen in [3, 7, 8], BDDs have several disadvantages: firstly, they do not retain the inherent flavour of decisions of strategies as maps from states to actions due to their bit-level representation and, hence, are hardly explainable. Secondly, they are notoriously hard to minimize [8], also because finding the best variable ordering is NP-complete [10]. BDDs only allow binary classification, so the actions have to be joined with the state space to represent a controller. The recent result in [38] discusses various heuristic-based determinization algorithms for BDDs representing controllers; however, they still suffer from those disadvantages we mentioned for BDDs. Algebraic decision diagrams (ADD) [4] are an extension of BDDs that allow to have more than two labels, i.e. associate every action to a leaf node. However, they still suffer from the same drawbacks as BDDs. In [13] ADDs are used for controller representation; however, no concrete algorithm is provided.

The formal methods community has made use of decision trees to represent controllers and counterexamples arising out of model checking Markov decision processes, stochastic games and LTL synthesis [1, 3, 7, 8]. DTs have also been used to represent learnt policies from reinforcement learning [29]. However, in contrast to our paper, [29] does not preserve safety guarantees, only considers axis-aligned splits and does not consider non-determinism. [17] suggests the possibility of using regression trees for representing policies, whereas we consider classification trees.

2 TOOL

dtControl is an easy-to-use open-source tool for post-processing memoryless symbolic controllers into various compact and more interpretable representations. We report the input and output formats as well as the algorithms that are currently supported. Note that the tool can easily be extended with new formats and algorithms. dtControl is distributed as an easy-to-install pip package[1] along with a user and developer manual[2].

Dependencies. dtControl works with Python version 3.6.7 or higher. The core of the tool which runs the learning algorithms require numpy, pandas and scikit-learn [28]. Optionally, dtControl may also require the C-based oblique decision tree tool OC1 [25].

Input formats. dtControl currently accepts controllers in three formats: (i) a raw comma-separated values (CSV) format with each row consisting of a vector of state variables concatenated with a vector of input variables; (ii) a sparse matrix format used by SCOTS; and (iii) the raw strategy produced by Uppaal Stratego. More details about the various formats are described in the user manual.

Algorithms. dtControl offers a range of parameters to adjust the DT learning algorithm, which are described in Section 4.

Output formats. dtControl outputs the decision tree in the DOT graph representation language (for visual presentation of the tree), as well as C code that can be directly used for implementation; see [2, Appendix A] for the DOT and C output that dtControl produces for the DT in Figure 1. Additionally, dtControl reports statistics for every constructed tree, namely size, the minimum number of bits required to represent symbols in obtained controller, and the construction time.

3 PRELIMINARIES - DECISION TREE LEARNING

A decision tree (DT) over the domain X with the set of labels $\mathcal{U}$ is a tuple $(\mathsf{T}, \lambda, \rho)$, where T is a finite full binary tree (every node has exactly 0 or 2 children), λ assigns to every leaf node (node with 0 children) a label $u \in \mathcal{U}$ and ρ assigns to every inner node (node with 2 children, also called decision node) of the tree a predicate, which is a boolean function $X \longrightarrow \{0, 1\}$.

The semantics of a DT is as follows: given a state $\vec{x}$, there is a unique *decision path* through the tree T starting from the root node (the only node with no parent) to a leaf node ℓ. This means that the label for state $\vec{x}$ is $\lambda(\ell)$. The decision path is defined by starting at the root node, and then for each decision node n evaluating the predicate on the state, i.e. computing $\rho(n)(\vec{x})$, and picking the left child if the predicate is true and the right child otherwise.

For example, consider the DT in Figure 1: T has 7 nodes, 3 of which are decision nodes (including the root node) and 4 of which are leaf nodes. A state of the system is a vector of 10 temperatures, e.g. $\vec{x} = (20.1, 20.2, 20.3, 20.4, 20.5, 20.6, 20.7, 20.8, 20.9, 21.0)$. To find the decision for this state, we first evaluate the predicate in the root node. Since the temperature in the second room is smaller than 20.625, the predicate is true and we go to the left child. We evaluate the next predicate in the same fashion and arrive at the leaf node labelled $(1, 1)$, which gives us a safe control input, in this case to turn on both heaters.

All DT learning algorithms implemented in dtControl follow the same underlying structure: given a finite set $C \subseteq X \times \mathcal{U}$ of feature-label pairs, it returns a DT that represents C precisely; this means that for every $(\vec{x}, u) \in C$, the leaf node of the decision path for $\vec{x}$ has the label u. In the setting of this paper, C is a controller, features are states and labels are actions[3].

To learn the DT, the algorithm tries to minimize the entropy of C, denoted $\mathtt{entr}(C)$, by splitting it according to a predicate. Formally, for some $C \subseteq \{(\vec{x}, u) \mid \vec{x} \in X, u \in \mathcal{U}\}$,

$$\mathtt{entr}(C) := -\sum_{u \in \mathcal{U}} p_u \log(p_u),$$

where $p_u := \frac{|\{(\vec{x},u) \in C\}|}{|C|}$ is the empirical probability of label u being in C; notation $|\cdot|$ denotes the cardinality of a set. The underlying algorithm works recursively as follows:

- **Base case:** If $\mathtt{entr}(C) = 0$, i.e. all pairs $(\vec{x}, u) \in C$ have the same label u, then return the following DT: the tree T has only a single node r, with $\lambda(r) = y$, and ρ has no domain in this case, as there are no decision nodes.
- **Recursive case:** If $\mathtt{entr}(C) \neq 0$, C needs to be split; for that, we use some predicate $P \in \mathsf{PREDS}$ which splits C, where the set PREDS to be picked here is a parameter of the algorithm that is discussed in Section 4.1. We pick the predicate that minimizes the entropy after the split, i.e.,

$$\arg\min_{P \in \mathsf{PREDS}} \mathtt{entr}(\{(\vec{x}, u) \in C \mid P(\vec{x})\}) + \mathtt{entr}(\{(\vec{x}, u) \in C \mid \neg P(\vec{x})\}).$$

  Intuitively, the best predicate is the one which is able to split C into two parts which are as homogeneous as possible. Given the best predicate, we recursively call the algorithm on the subsets resulting from the split, getting two DTs $(\mathsf{T}_t, \lambda_t, \rho_t)$ and $(\mathsf{T}_f, \lambda_f, \rho_f)$; the indices t and f indicate whether the predicate was true or false, respectively. Then we return the following DT: the tree T has the root node r, with the left child being the root of T_t and the right child the root of T_f. λ uses λ_t for leaves of the left sub-tree and λ_f for the right sub-tree. ρ is defined similarly on the inner nodes of the left and right sub-trees, with the addition that $\rho(r) = P$, i.e. the predicate of the root of T is the predicate we used for the split.

The symbolic controllers designed by SCOTS and Uppaal Stratego are generated by correct-by-construction synthesis procedures. In order to use these controllers for original systems (i.e. with infinite continuous states and inputs), we need to refine the controllers. For more details on refinement procedures, we kindly refer the interested reader to [20, 31, 36].

dtControl preserves the correctness guarantees by representing the symbolic controllers precisely, i.e. iterating until the entropy

[1]pip is a standard package-management system used to install and manage software packages written in Python. See https://pypi.org/project/dtcontrol/.
[2]Available at https://dtcontrol.readthedocs.io/en/latest/

[3]We use the term actions instead of control inputs, to avoid confusion because of the fact that the control inputs are the outputs of a DT.

in all leaf nodes is 0. In the case of determinization, dtControl represents one of the deterministic sub-controllers precisely, which is chosen on-the-fly during the construction.

4 METHODS

There are two parameters of dtControl: the set of predicates to consider (PREDS) and the way in which non-determinism is handled. For each of these, dtControl implements existing ideas and introduces new ones. Here, we only report the high-level ideas; for a more detailed description, refer to the user or developer manual.

4.1 Predicates

4.1.1 Existing idea: Axis-aligned splits. In the standard algorithms, e.g [9, 30], only *axis-aligned splits* are considered; i.e. predicates that can only have the form $x_i \sim b$, where x_i is one of the state variables, $b \in \mathbb{R}$, and $\sim \in \{\leq, \geq\}$. In our setting, the set of possible predicates is greatly restricted due to discretization (quantization). The number of splits to be evaluated for each variable x_i is equal to the number of discrete values of x_i.

4.1.2 Existing idea: Oblique splits. Beside the standard axis-aligned splits, dtControl also supports predicates of the form $\vec{w}^T\vec{x} \leq b$, where $\vec{w}, \vec{x} \in \mathbb{R}^n, b \in \mathbb{R}$. These *oblique predicates* [25] incorporate information from multiple state variables in a single split and thus have the potential to greatly simplify the induced decision tree [1]. However, due to combinatorial explosion, it is too costly to simply enumerate all possible oblique predicates even in the discretized space, due to which different heuristics are employed [25]. In this regard, dtControl supports the usage of predicates obtained using (an adapted version of) the OC1 algorithm [25].

4.1.3 New technique: Using binary machine-learnt classifiers. It is possible to find non-axis-aligned predicates splitting the controller by using classification techniques from machine learning. As our main goal is for the resulting tree to be explainable, we want to avoid complex predicates, and thus we restrict the classifiers we consider in two ways: (i) we only consider linear classifiers, and (ii) we restrict to binary classifiers, so that the resulting tree is binary.

We use these binary linear classifiers in a way that is similar to the classical one-vs-the-rest classification, e.g. [6, Chapter 4]: For each action u, we train a classifier LC_u that tries to separate the states with that action from the rest. We then pick that classifier whose predicate minimizes the entropy, i.e.

$$\boxed{LC := \arg\min_{u \in \mathcal{U}} \begin{array}{c} \mathtt{entr}(\{(\vec{x}, u) \in C \mid LC_u(\vec{x}) = 1\}) \\ + \\ \mathtt{entr}(\{(\vec{x}, u) \in C \mid LC_u(\vec{x}) = 0\}). \end{array}}$$

We considered various linear classification techniques including Logistic Regression [6, Chapter 4], linear Support Vector Machines (SVM) [6, Chapter 7], Perceptrons [6, Chapter 5], and Naive Bayes [39]. However, the latter two yielded significantly larger DTs in all of our experiments, so dtControl does not offer these algorithms to the end-user.

In summary, dtControl currently supports four possibilities for the set PREDS: axis-aligned predicates, the modified oblique split heuristic from [25] and oblique splits obtained either via logistic regression or linear SVM classifiers. Due to the modular structure of the code, it is easy to extend the existing approaches or add new methods, as described in our developer manual.

4.2 Non-determinism

In the general algorithm described in Section 3, for the sake of simplicity, we restricted our procedure to controllers that deterministically choose a single control input. In case of non-deterministic (also called permissive) controllers, the tuples in the controller C have the form $(\vec{x}, u)$, where u is now a set $\{u_1, u_2, \ldots, u_m\}$ of admissible control inputs. One approach to handle non-determinism is to simply assign a *unique label* to each set, and hence reduce the setting to the case where for every state there is only a single label. This means that the DT algorithm can be used in exactly the same way as described in Section 3. This method retains all information that was initially present in the given controller.

The disadvantage of handling non-determinism like this is that the number of unique classes may be as large as $2^{|\mathcal{U}|}$. In order to avoid this blow-up and optimize memory, one can decide to determinize the controller. If we have some knowledge about which value of a control input is optimal, e.g. from domain knowledge or since it was computed by an *optimization algorithm* as in Uppaal Stratego [12], this information can be used, eliminating the non-deterministic choice. Otherwise, one can use a standard determinization approaches, e.g. picking the value with the *minimum norm.* The tree can then simply be constructed from the determinized labels. Additionally, we propose the following alternative to these determinization approaches.

Novel determinization approach: Maximal frequencies. Our new determinization technique MaxFreq aims to minimize the size of the resulting DT. The underlying general idea is simple: if many of the data points share the same label, a DT learning algorithm should group them together under the common label. This idea naturally gives a determinizing strategy when applied in our context.

Consider a set C of pairs of state and sets of actions. The goal is to identify for each state a single action which can be assigned to it. Let f be the function for action frequency, which maps actions to their number of occurrences in C. Then, for each state $\vec{x}$ such that $(\vec{x}, \{u_1, u_2, \ldots, u_m\}) \in C$, we re-assign to $\vec{x}$ the single label u' which appears with the highest frequency. Formally, our determinization procedure produces for each state $\vec{x}$, an action $u'(\vec{x})$, where

$$\boxed{\forall(\vec{x}, \{u_1, \ldots, u_m\}) \in C.\, u'(\vec{x}) = \arg\max_{u \in \{u_1, \ldots, u_m\}} f(u).}$$

Once we have determinized C, we can use any method presented in Section 4.1 to find a predicate for the current node. After the set is split, the procedure is recursively applied to both child nodes, recomputing the action frequency each time.

In summary, dtControl offers 3 different possibilities to handle non-determinism: unique labels retaining the information, determinizing upfront by picking the action with the minimal norm, and using the novel heuristic MaxFreq.

5 EXPERIMENTS

All experiments were conducted on a server running on an Intel Xeon W-2123 processor with a clock speed of 3.60GHz and 64 GB

Table 1: Result of running the various methods on 10 different case studies. The 'Lookup table' column gives the size of the domain of the original controller. For all other columns, the number of decision paths in the constructed tree is indicated. The case studies are grouped together by the number of control inputs and methods based on whether they preserve non-determinism. ∞ indicates that the computation did not finish within 3 hours; n/a indicates that the approach is not applicable (we cannot determinize, as the model is already deterministic).

	Most permissive controller					Determinized controller			
Case Study	Lookup table	CART	LinSVM	LogReg	OC1	MaxFreq	MaxFreqLC	MinNorm	MinNormLC
Single-input non-deterministic									
cartpole [14]	271	127	126	100	92	**6**	7	56	39
2D Thermal [13]	40,311	14	14	8	12	5	**4**	8	**4**
helicopter [14]	280,539	3,174	2,895	1,877	∞	**115**	134	677	526
cruise [21]	295,615	494	543	392	374	**2**	**2**	282	197
dcdc [32]	593,089	136	140	70	90	**5**	**5**	11	11
Multi-input non-deterministic									
10D Thermal [15]	26,244	8,649	67	74	2,263	**4**	10	2,704	28
truck_trailer[18]	1,386,211	169,195	∞	∞	∞	21,598	**12,611**	95,417	30,888
traffic[35]	16,639,662	6,287	∞	4,477	∞	98	**80**	690	∞
Multi-input deterministic									
vehicle [32]	48,018	6,619	6,592	5,195	**4,886**	n/a	n/a	n/a	n/a
aircraft [33]	2,135,056	456,929	∞	**407,523**	∞	n/a	n/a	n/a	n/a

RAM. We ran the unique-label approach with all 4 possible predicate classes (see Section 4.1): axis-aligned predicates (CART) [9], oblique predicates with linear support-vector machines (LinSVM), logistic regression (LogReg), and the heuristic from [25], called OC1. Note that all these resulting trees represent the maximally permissive controller for the finite abstraction. Additionally, on all the non-deterministic models we ran our novel determinization approach (see Section 4.2) with axis-aligned predicates (MaxFreq), and with oblique predicates (MaxFreqLC where LC stands for linear classifier). For the results in Table 1, we used logistic regression as linear classifier, because it reliably performed well. As a competitor for our determinization approach we use a-priori determinization with the minimum norm, again both with axis-aligned predicates (MinNorm) and with logistic regression for linear predicates (MinNormLC). Additionally, we compare to the random a-priori determinization, to get an impression for possible cases where MinNorm would not be a natural choice but no better is given. However, since the results are always worse, we only report the numbers in [2, Appendix B]. Since some of the algorithms rely on randomization, we ran all experiments thrice and report the median.

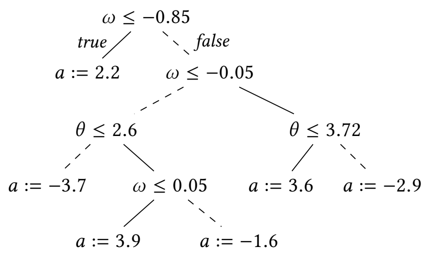
We run the discussed algorithms on ten case studies, five of which are marked as multi-input, containing control inputs which are multi-dimensional, i.e. $u = (u_1, \dots, u_m)$. All our algorithms work by giving each multi-dimensional control input a single action label, and then working on these labels as in the case of single-dimensional control inputs.

In order to compare the sizes of the representations of the controllers fairly, we provide two different ways. Firstly, the straightforward way is to compare the number of nodes used in the DT and the number of rows in the lookup table, which we do in Table 2 of [2, Appendix B]. However, a practically more relevant comparison should reflect the number of state symbols needed to capture the behaviour of the controller; these can also be directly related to memory requirements. To this end, in Table 1 for DTs we report the number of decision paths, as these induce a partitioning of the state space into symbolic states. For more information on this and an example, see Figure 2 and the discussion in Section 6.

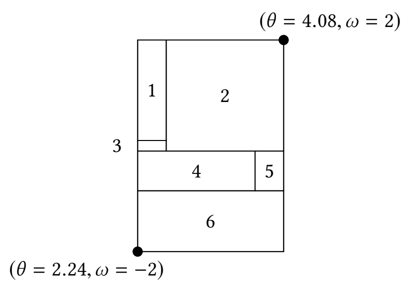
Beside comparing DTs to the lookup tables, we also compare them to BDDs. However, BDDs do not directly correspond to the state symbols. Hence we refrain from the state-symbols comparison and do not report BDD sizes in Table 1, but only in [2, Appendix B]. There, we compare the number of nodes in the BDDs to the number of nodes (not decision paths) generated by our DT algorithms. The BDDs were generated using SCOTS for all models but the two from Uppaal Stratego, cruise and 2D Thermal; for these two, we used the dd and autoref Python libraries. The BDDs were minimized as much as possible by calling reordering heuristics until convergence. The results show that the DT algorithms which determinize or which do not use oblique predicates are more scalable, as they were able to compute the result for all case studies, while BDDs timed out on dcdc and traffic. Depending on the case study, BDDs are usually in the same order of magnitude as CART, sometimes better, sometimes worse. On the one hand, on 10D Thermal and truck_trailer, BDDs have an order of magnitude less nodes, but on the other hand CART is able to produce results for dcdc and traffic. Compared to MaxFreq, there is the exception of truck_trailer, where the best BDD has a quarter of the size; on all other models, MaxFreq is at least one order of magnitude better.

6 DISCUSSION

Table 1 shows that DTs are always better than lookup tables. In the case of DTs exactly representing the most permissive controller, our linear-classifier-based algorithm, LogReg, generally performs better than the standard DT learning algorithm CART. An inspection of the trees showed that oblique splits indeed aid in this reduction. In order to save memory, however, our determinizing algorithms may



(a) Decision tree representation



(b) Non-uniform quantizer as a coder on the sensor side

Symbol	Input
1	-1.6
2	-3.7
3	3.9
4	3.6
5	-2.9
6	2.2

(c) Lookup table for the DT-based controller

Figure 2: End-to-end usage of DT-based controller: First, a DT representation is synthesized with the help of `dtControl` (the result of running MaxFreq on cartpole is shown here). Then a non-uniform quantizer is implemented at the sensor side, which for each decision path (i.e. a region in the state-space), sends a state symbol to the controller. At the controller, this symbol gives actual control input. In this case, the information needs to be sent over the sensor-controller channel is $\lceil \log_2(6) \rceil = 3$ bits per time unit. The theoretical lower bound on the data rate in this example is 1 bit per time unit to achieve invariance [34].

be used. Here, MaxFreq and its linear classifier variant, MaxFreqLC, easily outperform all other discussed algorithms, returning trees which can be drawn on a single sheet of paper in most of our case studies! The controller produced by MaxFreq for the case study cartpole is depicted in Figure 2a.

Apart from the compact representation of the controllers and efficient determinization, `dtControl` makes controllers more understandable. This helps to do some analysis for the systems and corresponding controllers. A few analyses were mentioned for the temperature control example in the introduction. Another application is that `dtControl` learns how to efficiently partition the state space. In general, the tools synthesizing symbolic controllers use uniform partitioning, i.e. a uniform quantizer is used to discretize the state set. Therefore, they need a large number of symbols to represent the state set. `dtControl` aggregates state symbols where the same control input is admissible to reduce the number of symbols required. In other words, `dtControl` provides a scheme to design non-uniform quantizers (i.e., state encoders with non-uniform partitioning of state-set), illustrated in Figure 2b.

The entries in Table 1 correspond to the necessary number of state symbols. For instance, consider the cartpole example in Table 1. The controller obtained using SCOTS requires 271 symbols to represent the domain of the controller, which implies that one needs to send 9 bits per time unit over the sensor-controller channel to achieve invariance. After processing the controller using `dtControl` with MaxFreq, we only need 6 symbols to represent the controller, corresponding to only 3 bits information. One can directly relate this idea of constructing efficient static coders to the notion of invariance feedback entropy introduced in [34]. This notion characterizes the necessary state information required by any coder-controller to enforce the invariance condition in the closed loop. For example, in the case of cartpole, the theoretical lower-bound on average bit rate for any static coder-controller to achieve invariance is 1 (obtained through the invariance feedback entropy [34]), which is not far from 3, computed using `dtControl`.

In summary, one can utilize the results provided in this paper for constructing efficient coder-controllers for invariance properties which is an active topic in the domain of information-based control [26].

7 CONCLUSION

We presented `dtControl`, an open-source, easily extensible tool for post-processing controllers synthesized by various tools such as SCOTS and Uppaal Stratego into small, efficient and interpretable representations. The tool allows for a comparison between various representations in terms of size and performance and also allows us to export the controller both as a graphic and as a code. We also presented a new determinization technique, MaxFreq, which easily converts non-deterministic controllers into extremely small deterministic decision trees. Further algorithms for controller representation were thoroughly evaluated and made accessible to the end-user. We believe these small representations will not only allow us to save memory but also help us in understanding and validating the model. As for future work, `dtControl` can be extended with

- further input and output formats, to also support tools such as `pFaces`[18] and `QUEST`[15];
- different predicates: this can be other, possibly even non-linear or non-binary, machine-learning classifiers or richer algebraic predicates utilizing domain knowledge;
- other *impurity measures* instead of entropy, which decide the predicate used for the split

ACKNOWLEDGMENTS

This work was supported in part by the H2020 ERC Starting Grant *AutoCPS* (grant agreement no 804639), the German Research Foundation (DFG) through the grants ZA 873/1-1 and KR 4890/2-1 *Statistical Unbounded Verification*, and the TUM International Graduate School of Science and Engineering (IGSSE) grant 10.06 *PARSEC*.

REFERENCES

[1] P. Ashok, T. Brázdil, K. Chatterjee, J. Křetínský, C. H. Lampert, and V. Toman. 2019. Strategy Representation by Decision Trees with Linear Classifiers. In *QEST (1)*. Springer, 109–128.

[2] Pranav Ashok, Mathias Jackermeier, Pushpak Jagtap, Jan Křetínský, Maximilian Weininger, and Majid Zamani. 2020. dtControl: Decision Tree Learning Algorithms for Controller Representation. arXiv:cs.LG/2002.04991

[3] P. Ashok, J. Křetínský, K. G. Larsen, A. Le Coënt, J. H. Taankvist, and M. Weininger. 2019. SOS: Safe, Optimal and Small Strategies for Hybrid Markov Decision Processes. In *QEST (1)*, D. Parker and V. Wolf (Eds.). Springer, 147–164.

[4] R. Iris Bahar, Erica A. Frohm, Charles M. Gaona, Gary D. Hachtel, Enrico Macii, Abelardo Pardo, and Fabio Somenzi. 1997. Algebraic Decision Diagrams and Their Applications. *Formal Methods in System Design* 10, 2/3 (1997), 171–206.

[5] C. Belta, B. Yordanov, and E. A. Gol. 2017. *Formal methods for discrete-time dynamical systems*. Vol. 89. Springer.

[6] C. M. Bishop. 2007. *Pattern recognition and machine learning, 5th Edition*. Springer.

[7] T. Brázdil, K. Chatterjee, M. Chmelik, A. Fellner, and J. Kretínský. 2015. Counterexample Explanation by Learning Small Strategies in Markov Decision Processes. In *CAV (1) (Lecture Notes in Computer Science)*, Vol. 9206. Springer, 158–177.

[8] T. Brázdil, K. Chatterjee, J. Kretínský, and V. Toman. 2018. Strategy Representation by Decision Trees in Reactive Synthesis. In *TACAS (1) (Lecture Notes in Computer Science)*, Vol. 10805. Springer, 385–407.

[9] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. 1984. *Classification and Regression Trees*. Wadsworth.

[10] R. E. Bryant. 1986. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.* 100, 8 (1986), 677–691.

[11] Ioannis T. Christou and Sofoklis Efremidis. 2007. An Evolving Oblique Decision Tree Ensemble Architecture for Continuous Learning Applications. In *AIAI (IFIP)*, Vol. 247. Springer, 3–11.

[12] A. David, P Gjøl Jensen, K. Guldstrand Larsen, M. Mikucionis, and J. H. Taankvist. 2015. Uppaal Stratego. In *TACAS (Lecture Notes in Computer Science)*, Vol. 9035. Springer, 206–211.

[13] Antoine Girard. 2013. Low-complexity quantized switching controllers using approximate bisimulation. *Nonlinear Analysis: Hybrid Systems* 10 (2013), 34–44.

[14] Pushpak Jagtap, Fardin Abdi, Matthias Rungger, Majid Zamani, and Marco Caccamo. 2018. Software Fault Tolerance for Cyber-Physical Systems via Full System Restart. *CoRR* abs/1812.03546 (2018).

[15] Pushpak Jagtap and Majid Zamani. 2017. QUEST: A Tool for State-Space Quantization-Free Synthesis of Symbolic Controllers. In *QEST (Lecture Notes in Computer Science)*, Vol. 10503. Springer, 309–313.

[16] Manuel Mazo Jr., Anna Davitian, and Paulo Tabuada. 2010. PESSOA: A Tool for Embedded Controller Synthesis. In *CAV (Lecture Notes in Computer Science)*, Vol. 6174. Springer, 566–569.

[17] Kyle D. Julian, Mykel J. Kochenderfer, and Michael P. Owen. 2018. Deep Neural Network Compression for Aircraft Collision Avoidance Systems. *CoRR* abs/1810.04240 (2018).

[18] Mahmoud Khaled and Majid Zamani. 2019. pFaces: an acceleration ecosystem for symbolic control. In *HSCC*. ACM, 252–257.

[19] Niels Landwehr, Mark A. Hall, and Eibe Frank. 2003. Logistic Model Trees. In *ECML (Lecture Notes in Computer Science)*, Vol. 2837. Springer, 241–252.

[20] Kim Guldstrand Larsen, Adrien Le Coënt, Marius Mikucionis, and Jakob Haahr Taankvist. 2018. Guaranteed Control Synthesis for Continuous Systems in Uppaal Tiga. In *CyPhy/WESE (Lecture Notes in Computer Science)*, Vol. 11615. Springer, 113–133.

[21] Kim Guldstrand Larsen, Marius Mikucionis, and Jakob Haahr Taankvist. 2015. Safe and Optimal Adaptive Cruise Control. In *Correct System Design (Lecture Notes in Computer Science)*, Vol. 9360. Springer, 260–277.

[22] Philipp J. Meyer, Matthias Rungger, Michael Luttenberger, Javier Esparza, and Majid Zamani. 2017. Quantitative Implementation Strategies for Safety Controllers. *CoRR* abs/1712.05278 (2017).

[23] T. M. Mitchell. 1997. *Machine learning*. McGraw-Hill.

[24] Sebti Mouelhi, Antoine Girard, and Gregor Gößler. 2013. CoSyMA: a tool for controller synthesis using multi-scale abstractions. In *HSCC*. ACM, 83–88.

[25] S. K. Murthy, S. Kasif, S. Salzberg, and R. Beigel. 1993. OC1: A Randomized Induction of Oblique Decision Trees. In *AAAI*. AAAI Press / The MIT Press, 322–327.

[26] G. N. Nair, F. Fagnani, S. Zampieri, and R. J. Evans. 2007. Feedback control under data rate constraints: An overview. *Proc. of the IEEE* 95, 1 (2007), 108–137.

[27] Daniel Neider, Shambwaditya Saha, and P. Madhusudan. 2016. Synthesizing Piece-Wise Functions by Learning Classifiers. In *TACAS (Lecture Notes in Computer Science)*, Vol. 9636. Springer, 186–203.

[28] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[29] Larry D. Pyeatt and Adele E. Howe. 1998. *Decision Tree Function Approximation in Reinforcement Learning*. Technical Report. Computer Science Department, Colorado State University.

[30] J. R. Quinlan. 1993. *C4.5: Programs for Machine Learning*. Morgan Kaufmann.

[31] G. Reissig, A. Weber, and M. Rungger. 2016. Feedback refinement relations for the synthesis of symbolic controllers. *IEEE Trans. Automat. Control* 62, 4 (2016), 1781–1796.

[32] M. Rungger and Zamani M. 2016. SCOTS: A Tool for the Synthesis of Symbolic Controllers. In *HSCC*. ACM, 99–104.

[33] Matthias Rungger, Alexander Weber, and Gunther Reissig. 2015. State space grids for low complexity abstractions. In *CDC*. IEEE, 6139–6146.

[34] Matthias Rungger and Majid Zamani. 2017. Invariance Feedback Entropy of Uncertain Control Systems. *CoRR* abs/1706.05242 (2017).

[35] A. Swikir and M. Zamani. 2019. Compositional synthesis of symbolic models for networks of switched systems. *IEEE Control Systems Letters* 3, 4 (2019), 1056–1061.

[36] P. Tabuada. 2009. *Verification and control of hybrid systems: a symbolic approach*. Springer Science & Business Media.

[37] Paul E. Utgoff. 1988. Perceptron Trees: A Case Study In Hybrid Concept Representations. In *AAAI*. AAAI Press / The MIT Press, 601–606.

[38] Ivan S. Zapreev, Cees Verdier, and Manuel Mazo Jr. 2018. Optimal Symbolic Controllers Determinization for BDD storage. In *ADHS (IFAC-PapersOnLine)*, Vol. 51. Elsevier, 1–6.

[39] Harry Zhang. 2004. The Optimality of Naive Bayes. In *FLAIRS Conference*. AAAI Press, 562–567.