TUM School of Computation, Information and Technology
Technische Universität München

Efficient Control via Reachability Synthesis for Cyber-Physical Systems

Kush Grover

Vollständiger Abdruck der von der TUM School of Computation, Information and Technology der Technischen Universität München zur Erlangung eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitz:
Prof. Dr. Helmut Seidl

Prüfende der Dissertation:
1. Prof. Dr. Jan Křetínský
2. Prof. Dr. Kim Guldstrand Larsen
3. Prof. Dr. Nils Jansen

Die Dissertation wurde am 08.11.2023 bei der Technischen Universität München eingereicht und durch die TUM School of Computation, Information and Technology am 13.03.2024 angenommen.

Abstract

Cyber-physical systems merge computing with physical processes to interact and operate in the physical world. These systems need to continuously make decisions in order to complete their objectives. The plans that describe the decisions in each state are called controllers and the problem of finding a right controller is called controller synthesis. To ensure their reliability, these controllers need to be verified. Controller synthesis is also possible using formal methods which produces correct by construction controllers and when done for reachability objectives can be called reachability synthesis.

In this thesis, we employed Markov Decision Processes (MDPs) to model these systems in a stochastic environment. Since they frequently operate in continuous domains, MDPs with finite states and actions may not accurately represent reality. Even though it is possible to generalize the definition of an MDP to have uncountable state and action spaces, this generalization, if left unchecked, can make the reachability problem undecidable. To tackle this challenge, we introduced some restrictions on the system, while trying to keep them as minimal as possible. Under these assumptions, we extended iterative algorithms Value Iteration and Bounded Real-time Dynamic Programming that already exist for solving reachability problems in finite MDPs.

We also used existing algorithms for reachability in finite MDPs and applied them to problems from the domain of robotics and aerospace. First is the task planning problem, where we modeled the high-level behavior of a robot using a finite MDP and synthesized a plan satisfying a reachability objective. Second is fault isolation, in particular for satellites, where we synthesized strategies to isolate faults by modeling it as an MDP. Both of these applications were supplemented with additional techniques to improve their effectiveness.

In contrast to task planning, where high-level plans are devised, motion planning deals with finding low-level paths adhering to reachability or temporal specifications, such as Linear Temporal Logic (LTL). Another layer of complexity is introduced here when the environment is assumed to be unknown i.e. the robot can only sense things within a certain radius. We tackle this problem for the first time by learning the semantic relations present in the environment and using them to bias the further search. Our experiments report that this idea improves the movement of the robot by more than 50% when compared to a naive approach which explores the whole environment first and then finds a path in the known environment.

Zusammenfassung

Cyber-physische Systeme verbinden Datenverarbeitung mit physischen Prozessen, um in der physikalischen Welt zu interagieren und zu arbeiten. Diese Systeme müssen ständig Entscheidungen treffen, um ihre Ziele zu erreichen. Die Pläne, die die Entscheidungen in jedem Zustand beschreiben, werden als Controller bezeichnet, und das Problem, den richtigen Controller zu finden, heißt Controller-Synthese. Um ihre Zuverlässigkeit zu gewährleisten, müssen diese Controller verifiziert werden. Formale Methoden können für die Synthese von Controllern benutzt werden. Werden insbesondere Erreichbarkeitsziele untersucht, wird die Synthese auch als Errechbarkeitssynthese bezeichnet.

In dieser Arbeit haben wir Markov-Entscheidungsprozesse (MDPs) verwendet, um solche Systeme in einer stochastischen Umgebung zu modellieren. Da sie häufig in kontinuierlichen Domänen operieren, können MDPs mit endlichen Zuständen und Aktionen die Realität nicht genau abbilden. Obwohl es möglich ist, die Definition eines MDP auf nicht abzählbare Zustands- und Aktionsmengen zu erweitern, kann diese Erweiterung das Erreichbarkeitsproblem unentscheidbar machen, wenn diese nicht überprüft wird. Um diese Herausforderung zu bewältigen, haben wir einige Einschränkungen für das System eingeführt, wobei wir versuch t haben, diese so minimal wie möglich zu halten. Unter diesen Annahmen haben wir die iterativen Algorithmen Value Iteration und Bounded Real-time Dynamic Programming erweitert, die bereits zur Lösung von Erreichbarkeitsproblemen in endlichen MDPs existieren.

Wir haben auch bestehende Algorithmen für die Erreichbarkeit in endlichen MDPs verwendet und sie auf Probleme aus der Robotik und der Luft- und Raumfahrt angewendet. Das erste Problem ist ein Aufgabenplanungsproblem, bei dem, auf einem hohen Abstraktionsniveau, das Verhalten eines Roboters mithilfe eines endlichen MDPs modelliert wird. Basierend auf dem MDP, haben wir anschließend eine Strategie synthetisiert, der ein Erreichbarkeitsziel erfüllt. Das zweite Problem beschäftigt sich mit Fehlerisolierung bei Satelliten. Dafür wurde das Verhalten des Satellieten mit einem MDP modelliert und anschließend eine Strategie zur Isolierung von Fehlern synthetisiert. Beide Anwendungen wurden mit zusätzlichen Techniken ergänzt, um ihre Effektivität zu verbessern.

Im Gegensatz zur Aufgabenplanung, bei der Pläne mit hohen abstraktionsniveau entworfen werden, geht es bei der Bewegungsplanung darum, detaillierte Pfade zu finden, die Erreichbarkeitsziele oder Temporale Formeln einhalten, wie zum Beispiel Linear Temporal Logic (LTL). Eine weitere Ebene der Komplexität wird hier eingeführt, wenn die Umgebung als un bekannt angenommen wird, d.h. der Roboter kann nur die Umgebung im bestimmten Radius wahrnehmen. Wir gehen dieses Problem zum ersten Mal an, indem wir die in der Umgebung vorhandenen semantischen Beziehungen lernen und sie für die weitere Suche nutzen. Unsere Experimente zeigen, dass dadurch die Bewegung des Roboters um mehr als 50% verbessert, verglichen mit einem naiven Ansatz, wo zuerst die gesamte Umgebung erkundet wird und anschließend einen Pfad in den bekannten Umgebungen findet.

सारांश

साङ्गणिक-भौतिक-प्रणाल्यः भौतिक-जगति अन्तरक्रियां कर्तुं कार्यं च साधयितुं साङ्गणिककार्यप्रणालीभिः भौतिकप्रक्रियाभिः च सह सम्मिलिताः भवन्ति। एतासां व्यवस्थानां उद्देश्यं पूर्णं कर्तुं निरन्तरं निर्णयः करणीयः। प्रत्येकस्यां अवस्थायां निर्णयानां वर्णनं कुर्वन्ति तत्र योजनाः नियन्त्रकाः इति उच्यन्ते तथा च सम्यक् नियन्त्रकस्य अन्वेषणसमस्या नियन्त्रकसंश्लेषणम् इति उच्यते। तेषां विश्वसनीयतां सुनिश्चयं कर्तुं एतेषां नियन्त्रकाणां सत्यापनम् आवश्यकम्। औपचारिकपद्धतीनां उपयोगेन नियन्त्रकसंश्लेषणं सम्भवति यत् निर्माणेन सम्यक् उत्पादयति नियन्त्रकाः तथा च यदा भवति तदा प्राप्यतासंश्लेषणम् इति प्राप्यतालक्ष्याणां उपयोगेन।

अस्मिन् शोधप्रबन्धे वयं वातावरणे आकस्मिकतया सह एतासां प्रणालीनां आदर्श प्रतिरूपणार्थं मार्कोवनिर्णयप्रक्रियाः नियोजितवन्तः। परन्तु यतः ता प्रतिक्रियाः बहुधा सततप्रान्तेषु कार्यं कुर्वन्ति, परिमितावस्थाभिः क्रियाभिः च सह मार्कोवनिर्णयप्रक्रियाः कदाचित् वस्तुजगतः परिदृश्यस्य समीचीनतया उपस्थापनं न कुर्वन्ति। मार्कोव निर्णयप्रक्रियाणां परिभाष्या सामान्यीकरणं सम्भवति यत् अगणनीयस्थितिः समष्टि क्रियास्थानानि च सन्ति किन्तु एतत् सामान्यीकरणं प्राप्यतासमस्यां अनिर्णययोग्यां करोति। एतस्याः समस्यायाः समाधानार्थं वयं प्रणाल्यां कति प्रतिबन्धान् प्रवर्तयामः, तथा च तान् यथासम्भवं न्युन्तमं कर्तुं प्रयत्नशीलाः। एतासां धारणानामन्तर्गतं वयं पुनरावर्तनीय-एल्गोरिदम् मूल्य-पुनरावृत्तिं तथा च सीमाबद्ध-वास्तविकसमय-गतिक-क्रमदेस्ह इति विस्तारितवन्तः ये परिमित-मार्कोवनिर्णयप्रक्रियाः मध्ये प्राप्यता-समस्यानां समाधानार्थं पूर्वमेव विद्यन्ते।

वयं परिमित-मार्कोव-निर्णय-प्रक्रियाणां कृते विद्यमान-सत्यापन-एल्गोरिदम्-इत्यस्य उपयोगं कृतवन्तः, स्वयंक्रिययन्त्रवायवीयक्षेत्रस्य च समस्यासु तान् प्रयुक्तवन्तः। प्रथमं कार्यनियोजनसमस्या अस्ति, यत्र वयं परिमितमार्कोवनिर्णयप्रक्रियाणां उपयोगेन स्वयंक्रिययन्त्रस्य च उच्चस्तरीयव्यवहारस्य प्रतिरूपणं कुर्मः तथा च एकं प्राप्यतालक्ष्यं सन्तुष्टं कृत्वा योजनां संश्लेषयामः। द्वितीयं दोषपृथक्करणं, विशेषतः उपग्रहाणां कृते, यत्र वयं दोषपृथक्करणाय मार्कोवनिर्णयप्रक्रियारूपेण तस्य प्रतिरूपणं कृत्वा रणनीतयः संश्लेषयामः। एतयोः द्वयोः अपि अनुप्रयोगयोः प्रभावशीलतां वर्धयितुं अतिरिक्तप्रविधिभिः पूरितम् आसीत्।

कार्यनियोजनस्य विपरीतम् यत्र उच्चस्तरीययोजनाः परिकल्प्यन्ते गतिनियोजनं रेखीयकालतर्कादिकं प्राप्यतायां वा कालविनिर्देशानां वा अनुसरणं कुर्वन्तः निम्नस्तरीयमार्गान् अन्वेष्टुं निबध्नाति। जटिलतायाः अन्यः स्तरः तदा भवति यदा पर्यावरणं अज्ञातं भवति अर्थात् स्वयंक्रिययन्त्रम् केवलं निश्चितत्रिज्यायाः अन्तः एव वस्तूनि इन्द्रियगोचरं करोति। वयं प्रथमस्थावत पर्यावरणे उपस्थितान् अर्थसम्बन्धान् ज्ञात्वा अग्रे अन्वेषणस्य पूर्वाग्रहं कर्तुं तेषां उपयोगं कृत्वा एतस्याः समस्यायाः निवारणं कुर्मः। अस्माकं प्रयोगाः प्रतिवेदयन्ति यत् एषः विचारः रोबोटस्य गतिं ५०% अधिकं संशोधयति यदा तस्य तुलना सहजदृष्टिकोणस्य भवति यत्र प्रथमं सम्पूर्णं वातावरणं अन्वेषयति ततः ज्ञातेषु वातावरणेषु मार्गं प्राप्नोति।

Acknowledgments

First and foremost, I would like to express my heartfelt gratitude to my advisor, Jan Křetínský, whose unwavering support and guidance have been pivotal throughout my Ph.D. His encouragement provided me with the strength and confidence to pursue my projects and achieve success.

I want to thank Pranav for being my mentor and helping me settle in during the initial phases. I am also deeply grateful to my colleagues, Muqsit, Debraj, Sudeep, Tobias, Steffi, Max, Sabine, and Maximillian, for the wonderful time we had together. I look forward to continuing our enriching discussions and delightful interactions whenever we meet again. Being part of our I7 chair at TUM and the ConVeY Research Training Group broadened my perspective, and the social and learning aspects of our frequent retreats nurtured me both personally and professionally. In that regard, I want to also thank Prof. Helmut Seidl as head of ConVeY for always encouraging me to achieve my goals. I also want to express my appreciation to my other co-authors, Fernando Barbosa, Alexander Bork, Jonis Kiesbye, and, Shruti Misra for the outstanding collaborations we had. I am grateful to Prof. Jana Tumova for her guidance during my exploratory days of entering robotics.

Beyond the academic realm, I am fortunate to have found a supportive community in Munich. Special thanks to Shrestha, Naman, and Agnishrota for making my time in Munich memorable. I am also deeply indebted to Ajeet and Ritika for being a daily part of my life and enjoying cooking, movies, chai, discussions together. My friends at home, Vasu Tyagi and Dhananjay Sharma, have been constant source of encouragement. Ritam Raha and Pankaj Pundir have been steadfast companions, and our frequent meetups across different cities have been a source of joy. I extend my appreciation to my friends from ISI and CMI, Aayush Agrawal, Jitesh Gupta, Badri Vishal Pandey, Mridul Sachdeva, Tejaswi Tripathi, Vasudha Sharma, and Rajarshi Roy, for catching up every now and then that evoke fond memories of our time at ISI and CMI.

I am grateful to Mr. Dinabandhu Chakravarty for his help in translating my abstract into Sanskrit.

Finally, I would like to express my deepest gratitude to my parents, Ved Prakash Grover and Sanjeeta Grover, for their unconditional love and support. Their encouragement has been the foundation of my academic pursuits. I am also grateful to my sister-in-law, Nisha Arora, for bringing new light into my life, and to my brother, Love Grover, for being my biggest fan and for instilling in me a passion for research and an unwavering belief in my abilities.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

Cyber-Physical Systems. Cyber-physical systems are becoming increasingly prevalent in our daily lives and are poised to play an even more significant role in the future. Examples of these systems include robot vacuum cleaners, autonomous cars and robots, unmanned aerial vehicles (UAVs), etc. While interacting with the physical world, they are required to do complex decision making. Their actions have consequences which can endanger lives [Pri18], result in significant financial losses [Mal11], or both [DH03]. To be able to trust such safety-critical systems, their behaviors need to be verified.

Formal Methods. Formal verification deals with techniques to ensure correctness and reliability of software and hardware systems. These methods can also be applied to synthesize verified controllers for cyber-physical systems. Controllers synthesized using these methods are also called *correct by construction*. There are several types of verification queries that can be asked for cyber-physical systems, such as reaching specific goal states (*reachability*), avoid ever reaching bad states (*safety*), and ensuring specific temporal relations between events (e.g., "after every request, there is a grant"), etc. Temporal logic, particularly Linear Temporal Logic (LTL) [Pnu77] is a natural and widely used specification language for expressing such temporal relations while abstracting away from precise event timings. Even though LTL is an expressive and powerful tool, *reachability* holds fundamental importance in verification, often serving as a foundation for addressing more intricate specifications. For instance, LTL model checking can be reduced to reachability problem [BK08, Section 10.3].

Various formalisms are available to model cyber-physical systems. Hybrid automata [Hen96] can represent both continuous time dynamics through differential equations and discrete changes in the system. On the other hand, Markov Decision Process (MDP) [Put94] is suitable for modeling the stochastic behavior within the system and its environment. When modeling systems with continuous domains (e.g., robot position coordinates or speed in a real number interval) using MDPs, they are typically assumed to be finite state models, achieved by discretization. While efficient in practice, sometimes these approximations are not good enough and a more precise analysis is desired. But, verifying systems with uncountable domains while giving strong guarantees is a challenging task. In fact, the reachability problem for an MDP with uncountable state and action spaces becomes undecidable in the general case (see Section 3.1). Nevertheless, many real-world systems exhibit structural characteristics that, when leveraged alongside specific restrictions, can make their analysis feasible. We identified these assumptions and gave two algorithms that solve the reachability problem. First algorithm only gives a lower bound on the result but the second algorithm, gives both lower and upper bound.

The verification techniques are now being developed more than ever with the growing formal methods community. New applications are being discovered where these techniques can be effectively employed. While some applications readily accommodate these techniques right off the shelf, others demand customized adaptations to cater to their specific requirements. One such application domain is *task planning* [JZK+19], which revolves around the challenge of devising a high-level plan for a robot to reach a desired goal state. In practice, things like imprecise control and inaccurate perception introduce probabilistic elements into the planning process. Given the "high-level" nature of the problem, it operates in a finite domain, task planning becomes an ideal candidate to be modeled as a finite MDP. We used a probabilistic model checker, in particular, PRISM [KNP11] to solve the MDP and generate a controller.

Another domain with increased autonomy is aerospace where of satellites, it becomes necessary to predefine protocols for handling faults that may arise during their missions. Fault Detection, Isolation, and Recovery (FDIR) [MMG+20] concepts address how satellite faults are managed. Following fault detection, the fault is isolated by observing different architectural elements. Once isolated, the recovery part assess whether the component can be replaced by a redundancy or if an alternate assembly can perform its function. In this thesis, our main focus is on the fault isolation part. The cost of checking the modules combined with the failure probabilities of components again suggests the use of finite MDPs. Using verification algorithms, we can derive cost-optimal strategies for efficiently isolating faults. However, we encountered the state space explosion problem [CKN+12] for our case study. To tackle this issue, we employed a Monte Carlo Tree Search (MCTS) [Cou07] based technique to trim the model and reduce it to a reasonable size.

In contrast to task planning, *motion planning* [Jau01] deals with finding a low-level plan that also considers the dynamics of the robot. Motion planning problems have been studied for different objectives, starting with just reachability [LaV98; KF11] to specifications from μ-calculus [KF09] or LTL [VB13]. However, these works assume knowledge of where the obstacles are and where the things of interest are. We look at the problem when these things are not known beforehand and the robot needs to satisfy a complex temporal tasks. We gave an algorithm, that learns from the explored regions and uses that information to bias the future search to quickly find a path that adheres to the specification.

1.2 Summary of Contributions

We give a brief and high-level overview of the main contributions. Additional details can be found in Chapters 3 to 5.

Verification of Uncountable MDPs. In our paper [GKM+22a], we introduce two algorithms, designed to solve the reachability problem in MDPs with uncountable state and action spaces. The general reachability problem is undecidable however by imposing certain assumptions, we can extend existing iterative methods designed for finite MDPs to work here. Our primary contribution in this paper is the identification of these necessary assumptions, striving to keep them as minimal as possible, thus pushing the boundaries of systems that can be reliably analyzed.

We present extensions of *Value Iteration* and *Bounded Real-time Dynamic Programming*, two well known algorithms for solving reachability in finite MDPs. The value iteration algorithm provides a lower bound on the target value, requiring fewer assumptions in comparison to previous approaches with guarantees. In contrast, the BRTDP algorithm provides both lower and upper bounds but demands more assumptions due to the inclusion of the additional upper bound. Importantly, both of these algorithms are *anytime* algorithms, continuously improving their estimations and ultimately converging to the true value in the limit.

Task Planning Using Verification of MDPs. In the domain of robotics, *task planning* is the problem of finding a high-level plan for a robot satisfying some given specification. In our paper [KGA+22], we converted it to a reachability problem for MDP by abstracting away from low level dynamics and only take into account the high level actions such as ”move to a certain pose, etc”. The task at hand was for a Franka Emika robotic arm to efficiently retrieve objects from a container and place them onto a conveyor belt. During runtime, this bin-picking task can be susceptible to various faults such as environmental changes, inaccurate perception, or imprecise robot control, resulting in failure of the high-level actions. These faults require some kind of recovery action to be taken by the robot. We used the PRISM model checker to derive a controller that acts as a universal plan in contrast to other planners which gives a sequence of actions to execute. This universal controller usually is quite large making it an undesirable choice. We solved the problem by employing dtControl to transform it into a decision tree. The decision tree controller was significantly compact, more explainable (in terms of understandability by humans), and orders of magnitude faster in finding the next action compared to the replanning approach. In addition to enhancing the controller, we generated another decision tree that pinpointed states with less likelihood of reaching the target state. This insight allowed for the incorporation of additional recovery actions, thereby enhancing the model’s effectiveness.

Fault Isolation Using Verification of MDPs. Fault tolerance is a critical requirement for various systems, particularly in the context of space systems like satellites. Handling this is achieved through Fault Detection, Isolation and Recovery (FDIR) concepts. Fault isolation is a crucial component of FDIR, and in [KGK23], we reduced it to the reachability problem for MDPs which can then be solved by a probabilistic model checker to find an optimal strategy. However, the state space can grow exponentially with the number of variables, leading to a state-space explosion issue. To address this, we employ a method based on Monte Carlo Tree Search (MCTS) to trim the state space, that leads to retaining only significant decisions and states. This results in a much smaller MDP, for which we can quickly find an optimal policy. Ultimately, this policy is converted into a decision tree using dtControl as before.

In addition to fault isolation, we generate analysis reports that provide information about components that are not entirely isolable. This information can be used to improve the architecture during the development phase of a satellite. We built a comprehensive tool for these tasks, complete with a user-friendly GUI.

Motion Planning in Unknown Environments. Going in another direction, our work in [GBT+21] delved into LTL motion planning in unknown environments. Unknown environment means that the robot could only perceive obstacle and labeling within a certain radius around it. We gave an algorithm which learns the semantic relations present in the environment to figure out similar transitions it should look for in the future to satisfy the formula as soon as possible. Additionally, to maintain record of explored and unexplored area we employed a frontier-based approach which can also suggest the direction with most unexplored area to guide the robot's movements. We combined the two biases and incorporated them in an RRG style algorithm.

1.3 Publication Summary

We list the publications by the thesis author that we discuss in this publication-based thesis. All papers are included in the appendix, preceded by a page that presents the full citation, a short summary, and a description of the thesis author's contributions.

Part I of the appendix contains two publications in which the author of this thesis is the **first author**. In these publications, the thesis author contributed more than 50% of the substantive findings:

B Kush Grover, Fernando S. Barbosa, Jana Tumova and Jan Křetínský. "Semantic Abstraction-Guided Motion Planning for scLTL Missions in Unknown Environments".
RSS, 2021. [GBT+21]

A Kush Grover, Jan Křetínský, Tobias Meggendorfer, Maximilian Weininger. "Anytime Guarantees for Reachability in Uncountable Markov Decision Processes".
CONCUR, 2022. [GKM+22a]

Part II of the appendix contains the following two publications in which the author of this thesis is **not the first author**:

C Jonis Kiesbye, Kush Grover, Pranav Ashok and Jan Křetínský. "Planning via Model Checking With Decision-tree Controllers".
ICRA, 2022. [KGA+22]

D Jonis Kiesbye, Kush Grover and Jan Křetínský. “Model Checking for Proving and Improving Fault Tolerance of Satellites”.
AEROCONF, 2023. [KGK23]

All four publications included in the dissertation are written in English and have been published in peer-reviewed proceedings of internationally recognized conferences.

Other Publications

In addition to the included publications, the author has co-authored the following papers while working on this thesis. These papers have been published in peer-reviewed conference proceedings and, while they are not part of the thesis, they are mentioned here for the sake of completeness.

- Maximilian Weininger, Kush Grover, Shruti Misra and Jan Křetínský. “Guaranteed Trade-Offs in Dynamic Information Flow Tracking Games”.
  CDC, 2021. [WGM+21]

1.4 Outline of Thesis

Chapter 2 introduces the notations used, some basic definitions and some preliminary concepts related to finite and uncountable MDP, Linear Temporal Logic, and motion planning.

Chapter 3 describes our algorithm for verification of uncountable MDPs with the assumptions required published in [GKM+22a]. Furthermore, Chapter 4 discusses our methods to apply probabilistic verification of MDPs to the fields of robotics [KGA+22] and aerospace [KGK23].

Lastly, motion planning problem for scLTL formulas in unknown environment is covered in Chapter 5, where the results from [GBT+21] are discussed.

2 Preliminaries

In this chapter, we start by setting up the notations, then we formally define Markov Decision Process with an example succeeded by a couple of algorithms to solve reachability for MDPs. Later on, we look at the syntax and semantics for LTL, and finally we see motion planning algorithms for reachability and LTL formulas.

2.1 Notation

We use $\mathbb{N}$, $\mathbb{R}$, and $\mathbb{R}^{\geq 0}$ to denote the set of natural numbers, reals, and non-negative reals respectively. For a set A, we use 2^A to denote its power set, A^* to denote the set of all finite strings over A and A^ω to denote the set of all infinite strings over A. For a probability measure $\mu \in \Pi(A)$, $supp(\mu)$ denotes the set $\{a \in A \mid \mu(a) > 0\}$.

Given two metric spaces A and B with metrics d_A and d_B, a function $f : A \to B$ is called *Lipschitz continuous* if $\exists K$ such that for every $a_1, a_2 \in A$, $d_B(f(a_1), f(a_2)) \leq K \cdot d_A(a_1, a_2)$. The constant K here is called the Lipschitz constant.

For Chapter 3, we assume familiarity with some basic measure theory concepts like measurable set or measurable function. For a measure space X with sigma-algebra Σ_X, $\Pi(X)$ denotes the set of all probability measures on X.

2.2 Markov Decision Processes

Markov Decision Process (MDP)s [Put94] are widely used formalism to model systems that have non-deterministic as well as probabilistic behaviors. They are defined using a set of finite states the system can be in, and a finite set of actions, the system can take. Upon taking an action from a state, the system goes to another state according to a probability distribution over successor states. The *reachability problem* is defined w.r.t. a set of desired states for which you either want to minimize or maximize the probability of system reaching them.

2.2.1 Formal Definition

Definition 1. *A* Markov Decision Process *is a tuple* $\mathcal{M} = (S, Act, Av, \Delta, s_0)$, *where*

- S *is a finite set of* states,
- Act *is a finite set of* actions,
- $Av : S \to 2^{Act}$ *assigns a set of* available actions *to every state,*

- $\Delta : S \times Act \rightarrow Dist(S)$ *is a* probabilistic transition function *that maps each state-action pair to a distribution over successor states, and*
- s_0 *is an* initial states.

It is also possible to replace the initial state s_0 with a set of initial states S_0. They can be proved equivalent by adding an auxiliary initial state $s_\perp$ and add transitions from $s_\perp$ to each $s \in S_0$ with probability 1. We abuse the notation and use $\Delta(s, a, s')$ to denote $\Delta(s, a)(s')$.

Paths. An *infinite path* in an MDP is an infinite sequence $\rho = s_1 a_1 s_2 a_2 \cdots \in (S \times Act)^\omega$ such that for every $i \in \mathbb{N}$, $\Delta(s_i, a_i, s_{i+1}) > 0$. A *finite path* is a non-empty and finite sequence $\varrho = s_1 a_1 s_2 a_2 \ldots s_n \in (S \times Act)^* \times S$ such that for every $i \in \mathbb{N}$, $\Delta(s_i, a_i, s_{i+1}) > 0$. The set of all finite paths of an MDP starting in state s is denoted by $\texttt{FPaths}_{\mathcal{M},s}$ and the set of all infinite paths is referred by $\texttt{Paths}_{\mathcal{M}}$.

End Components. An *end component* is a pair of sets of states and actions (S', Act') where $S' \subseteq S$, $Act \subseteq \bigcup_{s \in S'} Av(s)$ such that
(i) $supp(\Delta(s, a)) \subseteq S'$ for all $s \in S'$ and $a \in Act'$,
(ii) for all $s, s' \in S'$ there is a path $s_0 a_0 s_1 \ldots s_k$ where $s_0 = s$, $s_k = s'$, $a_i \in Act'$.
Intuitively, the definitions says that once you enter a state from S', it is not possible to go out by playing actions from Act', and it is possible to reach any state from any state in the future. A *Maximal End Component (MEC)* is an end component (S', Act') for which there does not exist another end component (S'', Act'') such that $(S', Act') \subsetneq (S'', Act'')$.

Strategies. A *strategy* (also called *policy*, *scheduler*, or *controller*) for an MDP $\mathcal{M} = (S, Act, Av, \Delta, s_0)$ is a function $\pi : \texttt{FPaths}_{\mathcal{M},s} \rightarrow Act$ which selects an action for a given finite path (or *history*). We use $\Pi_{\mathcal{M}}$ to denote the set of all strategies for an MDP $\mathcal{M}$. A strategy is called *memoryless* if it only depends on the last state instead of the whole path i.e. $\pi : S \rightarrow Act$. A memoryless strategy for an MDP induces a Markov chain[1] where the only action available in each state is defined by the strategy.

Example 1. *Figure 2.1 shows example of an MDP with 4 states. A possible path in this MDP is $q_0 a (q_1 a q_2 b)^\omega$. An example of an end component is $(\{q_2\}, \{a\})$, which is not a MEC. Whereas, $(\{q_1, q_2\}, \{a, b\})$ is an example of a MEC. A memoryless strategy π could be given as $\pi(q_0) = a, \pi(q_1) = c, \pi(q_2) = b$, and $\pi(q_3) = a$. Induced Markov chain under this strategy is shown in Figure 2.2.*

[1]A *Markov Chain* can be defined as an MDP which has only one action available in each state.

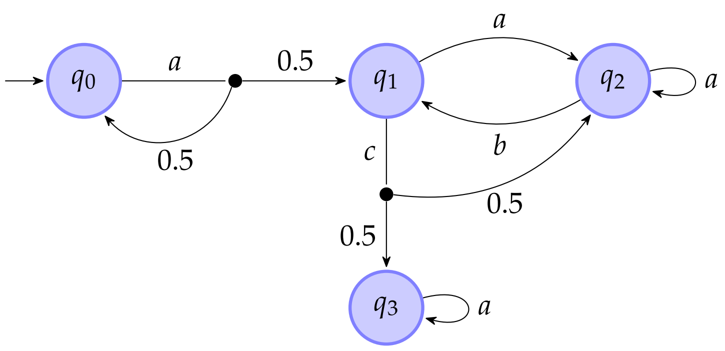


Figure 2.1: Example of an MDP

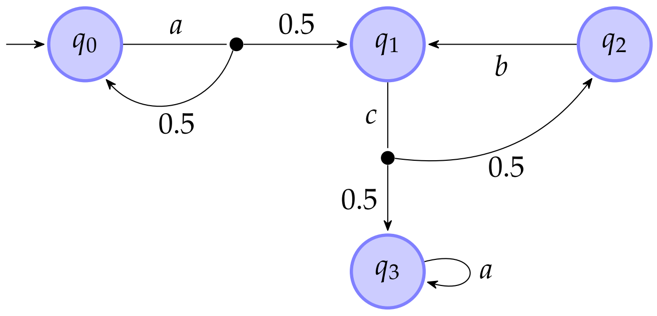


Figure 2.2: Example of an induced Markov chain for the MDP shown in Figure 2.1

2.3 Verification of MDPs

There are several types of queries one can ask for an MDP using different logics such as Probabilistic Linear Temporal Logic (pLTL) [BK08], Probabilistic Computation Tree Logic (pCTL) [BK08], etc. However, one of the most fundamental questions is the *reachability problem*, which inquires about reaching a set of goal states. In the context of MDPs, it asks about the probabilities of reaching the goal, also defined as the *value*.

Reachability Problem. The *reachability problem* for an MDP is defined w.r.t. a target set $T \subseteq S$ as *"what is the maximum probability of reaching T from the initial state s_0"?*[2] Along with the *value*, sometimes it is also desired to compute the strategy which achieves this value. We extend this definition to other states and define the *value function*, which assigns each state, the probability of reaching the target set.

Formally, for a given target set T, we define the *value function* as

$$v(s) = \max_{\pi \in \Pi_{\mathcal{M}}} \mathtt{Pr}^{\pi}_{\mathcal{M}}(\mathbf{F}\ T)$$

[2]It is also possible to define the dual problem, i.e. what is the minimum probability?, which can also be solved in a similar way.

Here, **F** T represents the event *eventually reaching* T. Finding a solution to this equation can be reduced to solving the *Bellman equation* given by

$$v(s) = \max_{a \in Av(s)} \sum_{s' \in S} \Delta(s, a, s') \cdot v(s')$$

This is a fix-point equation which already suggest techniques like fix-point iteration to solve it. The iterative methods, however, don't necessarily find the exact solution, instead they find an approximation. Another method to solve the Bellman equation is to use *Linear programming* [Put94]. A linear program can be formulated by encoding the Bellman equation for each state, which can then be solved using methods such as the simplex algorithm.

Complexity. The computational complexity of the LP-based approach is polynomial, whereas for an iterative method like VI, it becomes exponential. However these are worst case complexities and in practice, VI is often found to be more efficient and is commonly preferred for most applications.

For the purpose of this thesis, we will only look at iterative methods, in particular, Value Iteration (VI) and Bounded Real-time Dynamic Programming (BRTDP).

2.3.1 Value Iteration

Similar to a generic fix-point iteration method, a current approximation of the value function is maintained and updated in each iteration until the value is deemed good enough. The true value is reached in the limit, but the algorithm can be stopped at any time making this algorithm *anytime*.

We start with the initial values $v_0(s) = 0$ for all states $s \in S \setminus T$ and $v_0(s) = 1$ for all states $s \in T$, which is a lower bound of the actual values. Now, fix-point iteration is applied until the difference in the values of two consecutive iterations is smaller than a given threshold ϵ. The pseudo-code is shown in Algorithm 1. In the ith iteration, new values are computed by applying Bellman updates to all of the states given by

$$v_{i+1}(s) = \max_{a \in Av(s)} \sum_{s' \in S} \Delta(s, a, s') \cdot v_i(s')$$

The algorithm stops when the difference between the values of the initial state in two consecutive iterations is smaller than the given threshold i.e. $|v_{i+1}(s_0) - v_i(s_0)| \leq \epsilon$. This stopping criterion does not ensure that the computed value is close to the actual values, see [FKN+11, Section 4.2] for a detailed discussion.

Since we start with a correct lower bounds, it can be proved that the values we have will always be a lower bound to the actual value [Put94, Proposition 6.3.2]. Therefore, at whatever value the algorithm stops, it is a sound lower bound. This variant of VI is called the *synchronous VI* because in each iteration, values of all the states get updated. There is another variant called *asynchronous VI* [WB93; Put94] where in each iteration,

Algorithm 1 Value Iteration

```
Input: MDP $\mathcal{M} = (S, Act, Av, \Delta, s_0)$, target set $T \subseteq S$, and precision $\epsilon > 0$
1: for all $s \in S$ do
2:     $v_0(s) \leftarrow 0$
3: $\delta \leftarrow \infty$
4: $i \leftarrow 0$
5: while $\delta > \epsilon$ do
6:     for all $s \in S$ do
7:         $v_i(s) \leftarrow \max_{a \in Av(s)} \sum_{s' \in S} \Delta(s, a, s') \cdot v_{i-1}(s')$
8:     $i \leftarrow i + 1$
9:     $\delta \leftarrow \max_{s \in S} |v_i(s) - v_{i-1}(s)|$
   return $v_i(s_0)$
```

it samples a path starting from the initial state and ends when it reaches a target state or reaches the maximum path length. Then the values of states on this path are updated by backtracking from the last state until the initial state. This has the advantage that the important states (with higher probability of reaching) are updated more often, potentially reaching the correct value faster.

2.3.2 Bounded Real-time Dynamic Programming

In Value Iteration, only the lower bound is considered. However, if we assume the existence of a set $R \subseteq S$, referred to as "sink states", from which reaching a goal state is impossible, it becomes possible to establish an upper bound as well. In this approach, you also start with a conservative upper bound and gradually reduce it until you reach a satisfactory value.

If you keep both lower and upper bounds, it is called Bounded Value Iteration (BVI) and the result is an interval where the true value lies. The convergence criterion now is that the difference between lower and upper bound for the initial state is smaller than a given threshold. However, to make the lower and upper bounds converging to a value, we need the assumption that there are no end-components. If we don't have this assumption, there can be more than one solution to the Bellman equation and the algorithm might not find the optimal solution see [BCC+14, Example 1] for more details.

The asynchronous and the bounded versions combined gives the recipe for BRTDP [MLG05]. In [BCC+14], Brázdil et. al. also figured out a way to relax the end-component assumption. They sample paths to update values asynchronously and introduced a way to figure out when a simulation is most likely stuck in an end-component. Here, we only show the pseudo code of the BRTDP algorithm under the assumption that there are no end-components (Algorithm 2).

Algorithm 2 Bounded Real-time Dynamic Programming

```
Input: EC-free MDP $\mathcal{M} = (S, Act, Av, \Delta, s_0)$, target set $T \subseteq S$, sink set $R \subseteq S$, and a
    threshold $\epsilon > 0$
1: $L(\cdot,\cdot) \leftarrow 0$
2: $U(\cdot,\cdot) \leftarrow 1$
3: while $U(s_0) - L(s_0) \geq \epsilon$ do
4:     $(s, a) \leftarrow$ GetPair(())
5:     if $s \in T$ then $L(s, \cdot) \leftarrow 1$
6:     else if $s \in R$ then $U(s, \cdot) \leftarrow 0$
7:     else
8:         $U(s, a) \leftarrow \sum_{s' \in S} \Delta(s, a, s') \cdot U(s')$
9:         $L(s, a) \leftarrow \sum_{s' \in S} \Delta(s, a, s') \cdot L(s')$
   return $(L(s_0), U(s_0))$
```

2.3.3 Representing Strategies Using Decision Trees

Strategies generated by model checkers such as PRISM [KNP11] and STORM [DJK+17] are usually computed and exported as a lookup table. This lookup table has as many rows as the number of states in the model, which for any reasonable model (that can be solved by the model checkers), ranges from a few thousands to a few millions. This problem can be solved by using Binary Decision Diagram (BDD) [Lee59] or Algebraic Decision Diagram (ADD) [BFG+93] since they are way more succinct. However, they also come at the cost of explainability as discussed in [AJJ+20; BCC+15].

To solve both of these problems, [AJJ+20] introduces the methodology and a tool called `dtControl` to convert the lookup table strategy into a decision tree. These decision trees turn out to be extremely compact as shown by experiments in [AJJ+20] and way more explainable because of the size and structure of the decision trees.

2.4 Linear Temporal Logic

Linear Temporal Logic (LTL) [Pnu77] is a commonly employed specification language for analyzing a system's temporal behaviors. It offers a straightforward and intuitive means for humans to describe complex temporal behaviors while still being abstract about the exact timings of the events.

Syntax. We define an extended syntax of LTL here, which includes the eventually and globally operators. There are other operators that can be defined e.g. *weak until* but we don't need them here.

Definition 2. *Extended syntax for LTL is defined as:*

$$\phi := \top \mid \alpha \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid \phi_1 \wedge \phi_2 \mid \mathbf{X}\,\phi \mid \phi_1\,\mathbf{U}\,\phi_2 \mid \mathbf{F}\,\phi \mid \mathbf{G}\,\phi$$

where $\top$ *denotes* true, α *comes from a finite set of atomic propositions* Σ, $\neg, \vee, \wedge$ *are the usual* Boolean *operators,* $\mathbf{X}$ *is the* next *operator,* $\mathbf{U}$ *is the* until *operator,* $\mathbf{F}$ *is the* eventually *operator, and finally* $\mathbf{G}$ *is the* always *operator.*

Syntactically, $\mathbf{F}$ and $\mathbf{G}$ are defined using the until operator as $\mathbf{F}\,\phi := \top\,\mathbf{U}\,\phi$ and $\mathbf{G} := \neg\mathbf{F}\,\neg\phi$. LTL formulas are define over timed infinite words where each word is an infinite sequence over valuations of atomic propositions $w = w_0 w_1 \dots$ such that each $w_i \in 2^{\Sigma}$.

Semantics. Formally, a word w satisfies an LTL formula ϕ (denoted by $w \models \phi$) as

- $w \models \alpha$ if $\alpha \in w_0$
- $w \models \neg\phi$ if $w \not\models \phi$
- $w \models \phi_1 \vee \phi_2$ if $w \models \phi_1$ or $w \models \phi_2$
- $w \models \phi_1 \wedge \phi_2$ if $w \models \phi_1$ and $w \models \phi_2$
- $w \models \mathbf{X}\,\phi$ if $w^1 \models \phi$
- $w \models \phi_1\,\mathbf{U}\,\phi_2$ if $\exists\, i \geq 0$ such that $w^i \models \phi_2$ and $w^j \models \phi_1\ \forall\ 0 \leq j < i$.
- $w \models \mathbf{F}\,\phi$ if $\exists\, i \geq 0$ such that $w^i \models \phi$
- $w \models \mathbf{G}\,\phi$ if $\forall\, i \geq 0\ w^i \models \phi$

LTL to Automata. It is possible to convert any LTL formula to an equivalent non-deterministic automaton [BK08]. These automata can then be used for model checking by negating the specification and taking product with the system that can help to find a trace violating the specification. There are several variants of automaton one can use like Büchi, Rabin [Far02], Müller [Mul63], etc. Authors in [SEJ+16] show that it is also possible to convert any LTL formula to an equivalent *Limit-Deterministic Büchi Automaton (LDBA)* which is easier to use for model checking purposes. In these automata, the states are partitioned into two parts, deterministic and non-deterministic. As the name suggests, every execution eventually reaches the deterministic part and never goes back to the non-deterministic part. This also means that the accepting states are only in the deterministic part which in turn entails that the loop part of an accepting word lies in the deterministic part.

Syntactic Co-Safety. The syntactic co-safe fragment of LTL (scLTL) has the same syntax as LTL barring the operator $\mathbf{G}$. A *safety* specification is usually written as "something bad never happens" which in LTL translates to $\mathbf{G}\,\neg$`bad`. It is not possible to write safety specifications in this fragment hence the name. It can also be noted that without the $\mathbf{G}$ operator, specifications have to be satisfied within finite time horizon which also means that they can be converted to a deterministic Büchi automaton.

2.5 Motion Planning

Motion planning [Jau01] is a ubiquitous task in robotics which by and large can be described as the problem of finding a path in a workspace or an environment. The most fundamental motion planning problem is similar to the reachability problem we described earlier in Section 2.3, where the robot wants to reach a target set starting from an initial point.

Environment and Trajectories. Consider a robot moving in an environment $\mathcal{X} \subset \mathbb{R}^n$ starting from an initial point $x_0 \in \mathcal{X}$. Let $\mathcal{O} = \{\mathcal{O}_1, \mathcal{O}_2, \dots, \mathcal{O}_k\}$ be a set of obstacles in the environment where each $\mathcal{O}_i \subset \mathcal{X}$. We use $\mathcal{X}_{free}$ to denote the free space $\mathcal{X} \setminus \bigcup_{i=1}^{k} \mathcal{O}_i$. A *trajectory* σ in the environment $\mathcal{X}$ is given by a continuous function $\sigma : [0,1] \to \mathcal{X}$. A trajectory is called *collision-free* if $\sigma(t) \in \mathcal{X}_{free}$ for all $t \in [0,1]$.

Problem Statement. Given $\mathcal{X}$, $x_0 \in \mathcal{X}$, $\mathcal{O}$, and a target region $\mathcal{T} \subseteq \mathcal{X}$, find a collision-free trajectory σ such that $\sigma(0) = x_0$ and $\sigma(1) \in \mathcal{T}$.

Solutions. There are many ways to approach this problem, e.g. applying graph search algorithms like *Dijkstra* [Dij59] or A^* [HNR68] to a discretization of the environment or sampling-based techniques like Rapidly-exploring Random Trees (RRT) [LaV98] or Rapidly-exploring Random Graphs (RRG) [KF11]. Since in [GBT+21], we use *RRG*, we are going to look at it briefly.

2.5.1 Rapidly-Exploring Random Graphs

Algorithm 3 shows the pseudo code of the RRG algorithm adopted from [KF11]. It starts by initializing the graph with the initial point (line 1). A loop is then run m times and in each iteration of the loop a random point is sampled which is then used as the direction to grow our graph into. The sampled point is stored in x_{rand} and its closest point $x_{nearest}$ in V is found (line 3,4). This point $x_{nearest}$ is then fed to the STEER function, which tries to go towards x_{rand} from $x_{nearest}$ upto the distance δ and returns the new point x_{new} (line 5). If the steering from $x_{nearest}$ to x_{new} is collision-free, x_{new} is added to V and the edge $(x_{nearest}, x_{new})$ to E (line 6,7). Then, all neighbors of x_{new} in V within a certain radius are determined and are joined to x_{new} whenever possible (line 8-11). The neighborhood radius can be a constant or dependent on some heuristic e.g. current size of V.

2.5.2 Unknown Environments

When a robot needs to navigate in an unknown environment, it becomes necessary to explore the environment. One of the most common method for exploration is a *frontier-based* approach [Yam97], where a discretized grid-map of the environment with each cell storing whether it has been explored so far or not. The unexplored cells which

Algorithm 3 Rapidly-exploring Random Graphs (RRG)

```
Input: Initial point $x_0$, set of goal points $\mathcal{T}$ and step size $\delta$
 1: $V \leftarrow x_0, E \leftarrow \emptyset$
 2: for $i = 1, \ldots, m$ do
 3:     $x_{rand} \leftarrow$ SAMPLE
 4:     $x_{nearest} \leftarrow$ NEAREST($V, x_{rand}$)
 5:     $x_{new} \leftarrow$ STEER($x_{nearest}, x_{rand}, \delta$)
 6:     if COLLISIONFREE($x_{nearest}, x_{new}$) then
 7:         $V \leftarrow V \cup x_{new}, E \leftarrow E \cup (x_{nearest}, x_{new})$
 8:         $X_{near} \leftarrow$ NEAR($V, x_{new}$)
 9:         for $x_{near} \in X_{near}$ do
10:             if COLLISIONFREE($x_{near}, x_{new}$) then
11:                 $E \leftarrow E \cup (x_{near}, x_{new})$
    return $(V, E)$
```

have an explored neighbor are called *frontier cells* and a chain of frontier cells is called a *frontier*. These algorithms then make the robot go towards these frontiers. If there are multiple frontiers to explore, the algorithm picks a frontier based on some heuristic. One class of commonly used heuristics is *information gain* which can be defined for a frontier as

$$IG = size \times f(d)$$

where, *size* is the size of the frontier, d is its distance from the current position, and $f : \mathbb{R}^{\geq 0} \to \mathbb{R}$ is a monotonically decreasing function. The motivation behind inverse dependence on f is that you should explore the near things first.

3 Uncountable Markov Decision Processes

This chapter focuses on our contribution concerning the verification of uncountable MDP, as presented in [GKM+22a]. We begin by providing essential definitions to familiarize the reader with the model. Subsequently, we explore current state of the art and later elucidate how our contribution helps.

Many of the definitions here closely resemble their finite counterparts, allowing us to employ the same notations whenever feasible for the sake of simplicity and consistency.

Definition 3. *An uncountable MDP is given by a tuple $\mathcal{M} = (S, Act, Av, \Delta)$ where*

- *S is a compact set of states with topology $\mathcal{T}_S$ and Borel σ-algebra $\mathfrak{B}(\mathcal{T}_S)$,*
- *Act is a compact set of states with topology $\mathcal{T}_{Act}$ and Borel σ-algebra $\mathfrak{B}(\mathcal{T}_{Act})$,*
- *$Av : S \to 2^{Act} \setminus \emptyset$ is a function which assigns a set of available actions to each states, and*
- *$\Delta : S \times Act \to \Pi(S)$ maps a state action pair to a probability measure over successor states.*

We can also associate an initial state s_0 with an uncountable MDP, similar to the finite case. Definitions of *infinite* and *finite* paths can be borrowed from the finite MDP case. We also use $\texttt{FPaths}_{\mathcal{M},s}$ and $\texttt{Paths}_{\mathcal{M}}$ to denote the set of finite paths starting in s and the set of infinite paths respectively. The definition of a strategy too can be defined in a similar manner $\pi : \texttt{FPaths}_{\mathcal{M},s} \to Act$ and given a target set T, we can define the *reachability problem*. We define the *value* function which can be adjusted to this setting as

$$v(s) = \sup_{\pi \in \Pi_{\mathcal{M}}} \texttt{Pr}^{\pi}_{\mathcal{M}}(\diamond T)$$

The Bellman equation can be written for this value function as

$$v(s) = \sup_{a \in Av(s)} \int_{s' \in S} \Delta(s, a, s') v(s') ds'$$

We denote the value of state s under action a as $v(s, a)$ which is also called the *Q-value*

$$v(s, a) = \int_{s' \in S} \Delta(s, a, s') v(s') ds'$$

Given an uncountable MDP $\mathcal{M}$ and a function $f : S \to \mathbb{R}$, we define the successor expectation as $\Delta(s, a)\langle f \rangle := \int_{s' \in S} \Delta(s, a, s') f(s') ds'$. We also assume the existence a sampling procedure GETPAIR which returns a state and an action. We use $d_\times$ to denote a metric on the space of state-action pairs which satisfies that (i) for two pairs (s, a) and (s, a'), $k \cdot d_{Act}(a, a') \leq d_\times((s, a), (s, a')) \leq K \cdot d_{Act}(a, a')$ for some constants $k, K \geq 0$, and (ii) for two pairs (s, a) and (s', a), $k \cdot d_S(s, s') \leq d_\times((s, a), (s, a')) \leq L \cdot d_S(s, s')$ for some constants $l, L \geq 0$. One example of such metric could be $d_\times((s, a), (s', a)) := d_S(s, s') + d_{Act}(a, a')$.

Given two probability measures μ and ν over a sigma-algebra Σ_X, we define the *total variation distance* between them as $\delta_{TV}(\mu, \nu) := 2 \cdot \sup_{Y \in \Sigma_X} |\mu(Y) - \nu(Y)|$.

3.1 State of the art

Undecidability. Finding solution for the general case of the reachability problem for uncountable MDPs is undecidable. This can be proved by considering that ensuring almost sure termination in probabilistic programs is a special case of the reachability in uncountable MDP [FC19]. Even for non-stochastic linear hybrid systems, precise reachability analysis remains an undecidable problem [HKP+98].

Finite Horizon/Discounted Properties. While literature exists on finite-horizon [LL05; APL+08] and discounted properties [GHK04; HJS+20; Has12], our focus here is on infinite-horizon properties.

Reinforcement Learning. Learning techniques such as reinforcement learning can be used her but it only guarantees convergence to the true result over infinite time [MMR08]. Our focus here is to provide stronger guarantees.

Discretization Methods. Alternatively, researchers have explored increasingly precise discretization methods [JJG+19; TMK+13], also giving convergence guarantees in the limit. For safety-critical systems, just these convergence guarantees may not be enough and a bound on the error might be desired.

Lipschitz Continuity. One potential approach to address undecidability while maintaining robust guarantees involves imposing certain assumptions on the system. The Lipschitz continuity assumption is such an example, which has been adopted in several works. Even for finite horizon and discounted reward properties, it is necessary to assume Lipschitz continuity of the transition function, along with knowledge of the Lipschitz constant [AKL+10; SA11; Ber75; TMK+13].

Our work, however, takes a slightly weaker stance by assuming Lipschitz continuity of the value functions, with knowledge of the constant. Notably, this assumption is implied by the Lipschitz continuity of the transition function, see [GKM+22b, Appendix B.2.1]. There also exist literature such as [HJS+20; SJJ20] which assume Lipschitz continuity but without knowledge of the constant. These works offer convergence in the limit or "probably approximately correct" results but again, they do not provide a bound on the error since it would require knowledge of the constant.

Guarantees. Many of the techniques mentioned above rely on discretization [GHK04; SA11; Ber75; TMK+13; AKL+10]. However, they also necessitate some form of continuity assumptions to establish an error bound [AAP+07]. Also, none of them are anytime algorithms, that keeps on improving the result and can be stopped at anytime to get the current value.

3.2 Contribution: Verification of Uncountable MDPs

Since we focus on robust guarantees, our main goal here is to identify a set of necessary assumptions while trying to keep them as minimal and weak as possible. Under these assumptions, it becomes possible to lift the iterative algorithms *Value Iteration* and *BRTDP* from the finite case to this setting. Because of the general nature of our assumptions, these algorithms are more like templates that allow for a variety of heuristic in contrast to e.g., a discretization based algorithm.

We begin with stating the assumptions required for the VI algorithm and then explain the algorithm itself. Later, we do the same for the BRTDP algorithm.

3.2.1 Assumptions: Value Iteration

Recall that the central idea of asynchronous[1] VI is to start with a lower bound for all states, sample some states, and apply Bellman updates to these states to improve their values.

Apart from the continuity assumptions as discussed in Section 3.1, we also need some notions of computability and structural properties for the MDP. The computability assumptions are required to be able to do any computation with the model and are implicitly present in all of the previous works. For other assumptions, we briefly explain what they intuitively mean and how restrictive they are. For a detailed discussion on this, see [GKM+22a, Section 3.1].

Assumptions **A1**-**A4** defined below correspond to each part of the uncountable MDP; **A1** for $S \times Act$, **A2** for Av, **A3** for Δ, and **A4** for T. Since in any iterative algorithm, it is only possible to update values of finitely many states, therefore, we require Lipschitz continuity assumption **C**. It is used to extrapolate information from these sampled points to the whole state and action spaces.

Assumption A1. S and Act are metric spaces with computable metrics d_S and d_{Act} respectively, and $d_\times$ is a *compatible* metric on the product $S \times Act$.

[1]we only use asynchronous version here since synchronous VI requires updating values for an infinite number of states which can become infeasible to work with

Algorithm 4 Value Iteration for Uncountable MDPs

```
Input: ApproxLower query with threshold $\zeta$, satisfying A1–A4, B.VI and C.
1: Sampled $\leftarrow \emptyset$, $t \leftarrow 1$
2: while APPROX$_{\leq}(L(s_0)$, PRECISION$(t)) \leq \zeta$ do
3:     $(s, a) \leftarrow$ GETPAIR
4:     if $s \in T$ then $\widehat{L}(s, \cdot) \leftarrow 1$
5:     else $\widehat{L}(s, a) \leftarrow$ APPROX$_{\leq}(\Delta(s, a)\langle L \rangle$, PRECISION$(t))$
6:     Sampled $\leftarrow$ Sampled $\cup \{(s, a)\}$, $t \leftarrow t + 1$
   return yes
```

Assumption A2. For each state s, and a computable Lipschitz continuous function $f : Av(s) \to [0, 1]$, the value $\max_{a \in Av(s)} f(a)$ can be under approximated to an arbitrary precision. We use APPROX$_{\leq}(\max a \in Av(s) f(a), \epsilon)$ to denote this under-approximation with precision ϵ.

Assumption A3. For each state-action pair (s, a), and a Lipschitz continuous function $g : S \to [0, 1]$ which can be under approximated to an arbitrary precision, the successor expectation $\Delta(s, a)\langle g \rangle$ can be under approximated to an arbitrary precision. We abuse the notation and use APPROX$_{\leq}(\Delta(s, a)\langle g \rangle, \epsilon)$ to denote this under-approximation with ϵ precision.

Assumption A4. The target set of states T is decidable i.e. we are given a computable predicate which can decide whether a state $s \in T$ or not.

Assumption B.VI. Let $S^{\diamond} = \{last(\varrho) \mid \varrho \in \mathtt{FPaths}_{\mathcal{M},s}\}$ be the set of all reachable states from s. For any $\epsilon > 0$, $s \in S^{\diamond}$ and $a \in Av(s)$, GETPAIR eventually returns a pair (s', a') such that $d_{\times}((s, a), (s', a')) < \epsilon$ and $\delta_{TV}(\Delta(s, a), \Delta(s', a')) < \epsilon$.

Assumption C. The value functions $v(s)$ and $v(s, a)$ are Lipschitz continuous with constants C_S and $C_{\times}$ respectively.

Assumptions **A1** can be satisfied quite easily since for practical purposes most domains are metric spaces. Assumptions **A2** and **A3** can also be satisfied if you can sample densely from $Av(s)$ (e.g. random sampling) and can approximate $\Delta(s, a)$. Assumption **B.VI** essentially ensures the fairness of the GETPAIR procedure w.r.t reachable states, i.e. the GETPAIR provides a way to "exhaustively" generate all behaviours of the system up to a precision ϵ.

3.2.2 Value Iteration for Uncountable MDPs

The pseudo-code for VI is quite similar to the finite VI we saw in 2.3.1, and is shown in Algorithm 4. It begins with initializing the sampled points to empty set in line 1. Then, from line 2, the iterations start, with sampling a new state-action pair. If the sampled state is one of the target states, it's lower bound is set to 1 (line 4), else, it is updated by computing the value according to the previous lower bounds (line 5). At the end of loop, line 6 updates the sampled state-action pair using the new ones.

Theorem 1 states the correctness of the VI algorithm and for a detailed proof, we refer the reader to [GKM+22b, Appendix E.1].

Theorem 1. *Algorithm 4 is correct under Assumptions **A1–A4**, **B.VI**, and **C**, i.e. it outputs yes iff $V(s) > \zeta$.*

3.2.3 Assumptions: Bounded Real-Time Dynamic Programming

Since in BRTDP, we have both lower and upper bounds, we need extra assumptions. Assumptions **A5** and **A6** are the upper bound counterparts of **A2** and **A3** respectively. Assumption **B.BRTDP** is slightly weaker than **B.VI** since it doesn't consider all the states that are reachable from the initial state. Assumption **D** is probably the strongest one here but it is equivalent to the EC-free assumption which was also required for finite BRTDP initially.

Assumption A5. For each state s, and a computable Lipschitz continuous function $f : Av(s) \to [0,1]$, the value $\max_{a \in Av(s)} f(a)$ can be over approximated to an arbitrary precision. We use $\textsc{Approx}_{\geq}(\max a \in Av(s) f(a), \epsilon)$ to denote this over-approximation with precision ϵ.

Assumption A6. For each state-action pair (s,a), and a Lipschitz continuous function $g : S \to [0,1]$ which can be over approximated to an arbitrary precision, the successor expectation $\Delta(s,a)\langle g \rangle$ can be over approximated to an arbitrary precision. We again abuse the notation and use $\textsc{Approx}_{\geq}(\Delta(s,a)\langle g \rangle, \epsilon)$ to denote this over-approximation with ϵ precision.

Assumption B.BRTDP. Let $S^{\diamond}_{\mathcal{F}}$ be the set of all states reachable by using actions which are arbitrarily close to the optimal actions (see [GKM+22a] for a detailed explanation). The procedure $\textsc{GetPair}$ is *fair* w.r.t $S^{\diamond}_{\mathcal{F}}$ i.e. for any $\epsilon > 0$, $s \in S^{\diamond}_{\mathcal{F}}$, and $a \in Act(s)$, $\textsc{GetPair}$ a.s. eventually yields a pair (s', a') such that $d_{\times}((s,a),(s',a')) < \epsilon$ and $\delta_{TV}(\Delta(s,a), \Delta(s',a')) < \epsilon$.

Assumption D. Along with T, there also exists a decidable set R, called sink, such that $V(s) = 0$ for all $s \in R$. Moreover, for any $s \in S$ and strategy π we have $Pr^{\pi}_{\mathcal{M},s}[\diamond(T \cup R)] = 1$.

Assumptions **A5**, **A6** and **B.BRTDP** are again easy to satisfy as before. Assumption **D** requires the system to eventually reach a target or sink state from every state making it the strongest assumption here.

Algorithm 5 BRTDP for Uncountable MDPs

```
Input: ApproxBounds query with precision $\epsilon$, satisfying A1–A6, B.BRTDP, C and D.
1: Sampled $\leftarrow \emptyset$, $t \leftarrow 1$
2: while $\text{APPROX}_{\geq}(U(s_0), \text{PRECISION}(t))$ - $\text{APPROX}_{\geq}(U(s_0), \text{PRECISION}(t)) \geq \epsilon$ do
3:     $(s,a) \leftarrow$ GETPAIR
4:     if $s \in T$ then $\widehat{L}(s,\cdot) \leftarrow 1$
5:     else if $s \in R$ then $\widehat{U}(s,\cdot) \leftarrow 0$
6:     else
7:         $\widehat{U}(s,a) \leftarrow \text{APPROX}_{\geq}(\Delta(s,a)\langle U\rangle, \text{PRECISION}(t))$
8:         $\widehat{L}(s,a) \leftarrow \text{APPROX}_{\leq}(\Delta(s,a)\langle L\rangle, \text{PRECISION}(t))$
9:     Sampled $\leftarrow$ Sampled $\cup\{(s,a)\}$, $t \leftarrow t+1$
   return $(L(s_0), U(s_0))$
```

3.2.4 BRTDP for Uncountable MDPs

We are now ready to describe the pseudo-code, shown in Algorithm 5. It is again, quite similar to the finite state setting. It starts with initializing the set of sampled points in line 1. While the difference between current lower and upper bound is greater than a given threshold, the loop in lines 2-9 is executed. A state-action pair is sampled using the GETPAIR method (line 3). If the state is in either target or sink, the lower or upper bound is updated accordingly (line 4, 5). Otherwise, lines 7 and 8 compute new lower and upper bounds for the sampled state-action pair and store them. Once the lower and upper bound for the given state are close enough, the loop is terminated and returns the bounds. The correctness of Algorithm 5 is shown in Theorem 2 and for its detailed proof, we suggest to look at [GKM+22b, Appendix E.2].

Theorem 2. *Algorithm 5 is correct under Assumptions **A1–A6**, **B.BRTDP**, **C** and **D**, and terminates with probability 1.*

3.3 Outlook

We extended the infamous VI algorithm alongside the BRTDP algorithm to the uncountable setting under a few assumptions. They can specially be helpful in situations where strong guarantees are required. Their current state might be far from a phase that can be applied in real-world scenarios but this is a step in the right direction.

We now discuss the possibilities of improving the algorithm, relaxing the EC-free assumption, and extending to LTL properties.

Using Heuristics. To enhance performance while maintaining strong guarantees, some heuristics and learning techniques can be effectively employed. One approach involves adapting Lipschitz constants for different sections of the domain, as opposed to using a universal constant. This optimizes computations on parts with low Lipschitz constants, resulting in faster processing.

Another strategy uses domain knowledge to sample points more frequently from important areas, reducing the overall number of required samples and thereby improving performance. An alternative sampling method follows a path-sampling method, similar to [BCC+14], which automatically ensures that more probable points are sampled more frequently. Finally, one can employ the difference between the lower and upper bounds as a heuristic and increase sampling in areas where this difference is substantial.

However, it's crucial to ensure that these heuristics adhere to the corresponding assumption (**B.VI** or **B.BRTDP**), e.g., it can be achieved by sampling random points with a certain probability p and using the heuristic for sampling with a probability of $1 - p$.

Extending to LTL. Extending this algorithm to handle Linear Temporal Logic (LTL) formulas presents several challenges. Even for repeated reachability properties i.e. repeatedly reaching a set of states infinitely often, significant complications arise due to the inability to identify end components. Furthermore, for safety specifications like ensuring an agent remains within a safe region indefinitely (**G** `safe_region`), relying solely on sampling is not possible since, even for a single execution, it's not feasible to determine if the agent will exit the safe region after a finite amount of sampling. For a more in-depth exploration of why relaxing assumption D and extending the algorithm to LTL scenarios is a challenging task, we refer to [GKM+22b, Section 4.1.2, Appendix C.3].

4 Applications of Markov Decision Process Verification

Verification techniques are being used more and more across diverse fields, including robotics [LWA+10; ZRF+19], security [BCM18], aerospace [FSP+16; HAS+14], and more [CZ11; EAA10]. However, there are instances where employing these techniques right out of the box can prove to be challenging. This may be attributed to factors like the substantial size of the model, making it difficult to be handled by a model checker, or the unique nature of the problem at hand, which doesn't precisely align with the abilities of a model checker. In such scenarios, there arises a need to tailor the verification method to the specific requirements of the application.

We found two such challenges in the fields of robotics and aerospace. To address these challenges, we used the existing verification methods combined with new ways to improve their efficacy.

4.1 State of the art: Task Planning With Recovery

Our first application [KGA+22] is from robotics, where the objective is to devise a high-level *plan* for a robot that satisfies some specific requirements, typically reachability. In the context of robotics, this challenge is often referred to as *task planning* [GNT04].

In real-world scenarios, these plans are susceptible to errors, which can arise due to factors such as environmental changes during execution, inaccurate data generated by the sensors, or imprecise robot control. These errors can lead to the system reaching undesired states, rendering the computed plan ineffective. In such cases, most traditional planners resort to computing a new plan. In addition to the challenge of replanning, it is also possible that the system reaches a state from which it is not possible to reach the goal state using any sequence of the defined actions. We call these states *deadlock* states. This can happen if the engineer responsible for modeling the system did not anticipate its possibility.

Task Planning Coupled With Motion Planning. Task planning is often combined with motion planning, to generate a plan that can be executed by the robot. In the research by Kaelbling et al. [KL11], they introduce a hierarchical planning approach that addresses both task planning and motion planning as a unified problem. Here, graph algorithms like A^* handle the task planning aspect, while a separate layer manages the motion planning.

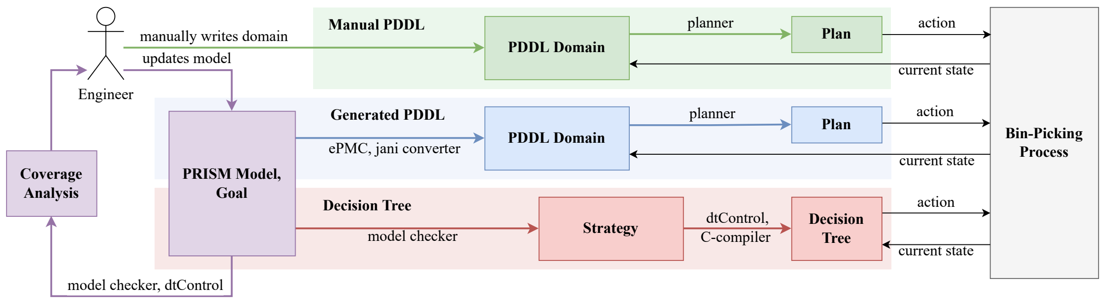


Figure 4.1: Framework for the task planning solution with a feedback loop for improving the model

Planning Domain. Traditionally, task planning has been solved by planners such as STRIPS which employ Planning Domain Definition Language (PDDL) to define the model. They solve the planning problem by employing different algorithms and find a suitable plan. In [SFR+14] Srivastava et.al. suggests a framework where PDDL is used to define the task planning problem, which can then be solved by any planner.

Encode as a Different Problem. Apart from PDDL, it is also possible to use ASP to encode the task planning problem. For an in-depth comparison between PDDL and ASP, see [JZK+19]. More approaches include encoding task planning as a Boolean satisfiability problem [KS92; Rin12; HHP+13] or an SMT instance [NPM+14] which can then be solved by respective solvers.

Model Checking. Model checking has also been used to solve task planning problems, which usually generates a *controller* satisfying the specification. One such example is presented in [JK11] where the sensors could be erroneous and probabilistic verification techniques are used to compute the probability of satisfying the goal under a given plan.
None of the approaches describe above give a "universal plan" which suggests an action to play in all of the states. This also helps us in identifying the deadlock states which can be used to improve the model by adding recovery actions to those states.

4.2 Contribution: Task Planning With Recovery

The overview of our approach is depicted in the bottom pipeline of Figure 4.1 (labeled as **Decision Tree**), while the top pipeline (**Manual PDDL**) shows the traditional approach by writing PDDL domain manually and middle pipeline (**Generated PDDL**) represents an intermediate approach.

In our approach, during the modeling phase, the engineer can receive information about these bad states that don't reach the target with a given probability threshold using coverage analysis. They are provided in a systematic manner, where similar

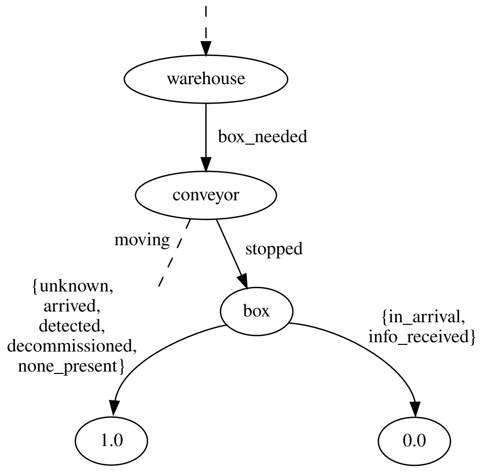

warehouse
box_needed
conveyor
moving
stopped
{unknown, arrived, detected, decommissioned, none_present}
box
{in_arrival, info_received}
1.0
0.0

Figure 4.2: Part of a decision tree representing the values

states are combined into one branch of a decision tree. The engineer can then pick a branch, e.g. with most states, and add the appropriate action and repeat the process. Once a model with no deadlock states is created, it can be sent to a model checker like PRISM which returns the optimal strategy. This strategy can also be converted to a decision-tree (discussed in Section 2.3.3) resulting in a succinct universal plan, making it easy to understand, and fast to execute.

4.2.1 Our Approach

We show the details on our approach on a case-study in which we modeled a Franka-Emika robotic arm using an MDP. It was placed in a warehouse and its task was to pick an item from a box and place them on a sorter tray.

Model. We used 14 variables to represent the states of the MDP, resulting in more than a million states. We defined the actions that the robotic arm can perform from a given state. These actions had some preconditions to ensure that they are available in the current state and can be taken safely. The transitions define how the state changes after taking an action. However, as discussed before, the execution of these actions are not perfect and the system can end up in an unexpected state. This behavior can be modeled completely using MDPs if all the possible outcomes are known with their respective probabilities of occurring. But, only so much of these unpredictable outcomes can be incorporated during the modeling phase. Therefore, to obtain a universal plan, we defined all possible states as initial. The target states are the ones for which the bin picking cycle is finished.

Model Checking. We feed this MDP to a model checker like PRISM or STORM which can solve the reachability problem using algorithms like VI. They can output the prob-

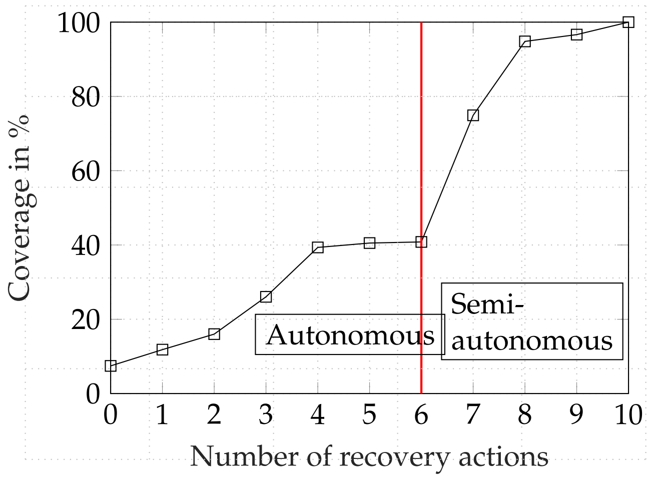


Figure 4.3: Percentage of initial states which can reach a target state. Actions 1–6 are autonomous, meaning the robot can perform them on its own, and actions 7–10 are semi-autonomous, requiring human intervention during execution.

ability of reaching the target (recall that it is called value) for each state or the strategy as a lookup table depending on what is required.

Improving the Model. For coverage analysis, we get the values of all the states as a lookup table that the model checker outputs. These lookup tables are converted to decision trees using dtControl. Figure 4.2 shows an example of such a tree. These decision trees can now help identify the deadlock states e.g., by looking at Figure 4.2, we can conclude that the set of states with `conveyer=stopped` and `box=in_arrival,info_received` cannot reach the target states since their value is 0. Now, a recovery action can be added to these state to improve their value. This process can be repeated multiple times until there are no deadlock states.

Final Strategy. Once all the recovery actions have been added, the model checker is sent to the model checker one more time and a strategy is extracted this time instead of the values of the states. This strategy is also converted to a decision tree which can then be used by the robotic arm.

4.2.2 Experimental Results

First, we show how coverage analysis looked for our model of the robotic arm. Figure 4.3 shows the percentage of states that reach a target with the current strategy and how it evolves by adding more recovery actions. Since autonomous actions are preferred, they are added first. We ran the final controller generated by our method on a real-world Franka-Emika arm working in a warehouse. Figure 4.4 show how many recovery actions were needed during 134 bin-picking cycles.

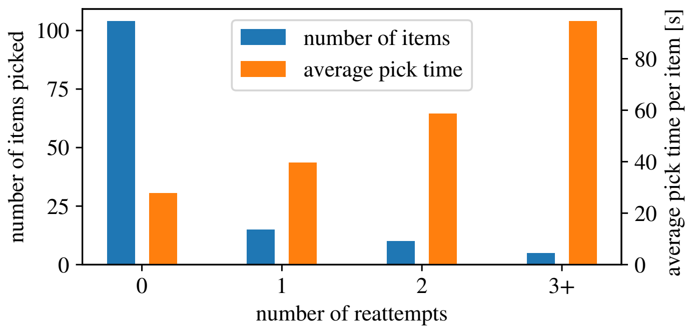


Figure 4.4: Number of reattempts required by 134 bin-picking cycles

Second, we show how our strategy performed in the real world when the robot was working in a warehouse. Figure 4.4 shows, it completed 134 cycles and a few of them required reattempts. These reattempts were suggested by the recovery actions and all 134 of the bin-picking cycles were successful eventually.

Although, it is not evident from the experimental results but finding a recovery action was not difficult because of the explainability of the controller. When a branch is picked, finding which variables needs fixing becomes quite easy and a lot of states with the same issue can be fixed simultaneously.

4.3 State of the art: Fault Isolation for Satellites

Space missions have little room for human intervention, therefore they are required to be fault tolerant, usually handled by Fault Detection, Isolation and Recovery (FDIR) concepts. Many methods have been developed under this framework such as Failure Modes and Effects Analysis (FMEA) [Fak21] and Fault Tree Analysis (FTA) [MMG+20]. One way to improve the fault tolerance is by adding redundant parts for each component but it is not economical since adding even a tiny weight on space missions results in a much increased cost. Therefore, they are required to be as minimal as possible. Here, we focus on making the systems 1-fault tolerant i.e. if one fault occurs, the satellite can still work perfectly.

Failure Modes and Effects Analysis. In FMEA, engineers analyze consequences for every fault possible in a bottom up fashion and changes the model as required. An improvement to FMEA was introduced in [BBC+14] called FAME which introduces timed fault propagation graphs. Here, when a fault occurs, it is propagated through several modes which are monitored. This helps pin-pointing where the fault occurred and figure out a way to handle it. Redundancy Verification Analysis (RVA) is another technique to identify interactions and failure scenarios specific to redundant configurations. RVA goes into more detail than a system-level FMEA, here signal levels are tracked throughout the system and block diagrams are generated to express the tolerance of the design to single failures.

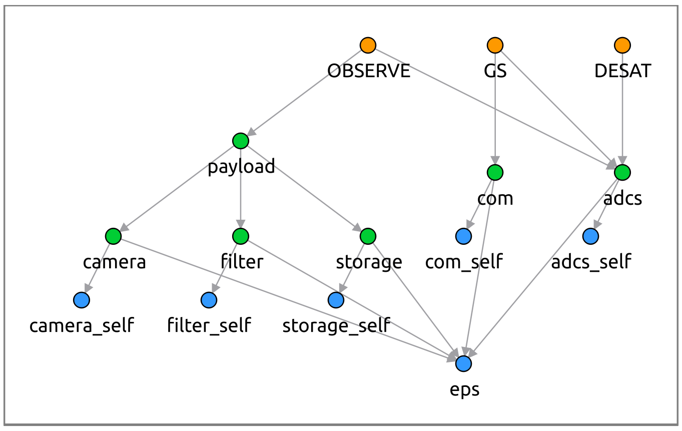


Figure 4.5: Example of an architecture graph

Fault Tree Analysis. On the other hand, FTA is a top down approach where the input is a fault-tree which looks similar to an architecture graph that we use as the input. In [MMG+20; MGN], a recovery strategy is synthesized to handle faults. But here, the information about where the fault occurred is assumed to be known. We, however, focus more on how to figure out where exactly does the fault occurred. Also, we can also figure out if the system has enough redundancies to make it one fault tolerant.

Model Checking. A model checking based method is described in [CPC03], where they also try to figure out a fault based on observables. The investigated models however use highly accurate continuous domain models in the background that are discretized by the Livingstone framework while we focus on the architecture of spacecraft in the design phase where no

4.4 Contribution: Fault Isolation for Satellites

Architecture Graph. The input of our problem is a directed graph we call *architecture graph* where the nodes are either basic components or assemblies which are functional units that depend on smaller assemblies or components. The dependency relation between nodes is defined using directed edges. We call the assemblies which can be observed as *modes* which can be used to isolate faults. Each equipment has a probability of failing, which introduces stochasticity in the model. Each mode is also associated with a cost which defines how expensive testing them is. Figure 4.5 shows an example of a dependency graph for a satellite.

Problem Statement. To isolate a fault, some modes need to be observed with which either the components used in this modes can be ruled out of suspicion or the the ones not in this mode can be ruled out of suspicion. Although, all of the modes can be tested in order to isolate the fault, we want to spend minimal effort in doing so, since resources are extra valuable for space missions.

Modeling the System as an MDP. It is possible to convert this problem into a reachability problem for MDPs, the states of which keep track of the suspicious components. The actions are testing the modes with corresponding cost associated with them. The transition probabilities depends on the probabilities of the suspicious equipments. Each mode correspond to an initial state since a fault can be encountered in them but it is also possible to make all components suspicious. The set of target states are where it is not possible to rule out any more suspicious components by checking any mode. The fault is isolated completely if there is only one suspicious component therefore the target.

4.4.1 Pruning the model using Monte Carlo Tree Search

Similar to the task planning application, we give this MDP to a model checker which generates a strategy. This method works in theory but becomes intractable as the number of components increase. To solve this issue, we suggested using Monte Carlo Tree Search (MCTS) to prune the model such that only the best actions are kept in every state based on simulations. Solving the reachability problem for the smaller MDP can now become feasible. After getting the strategy, as always, we use dtControlto convert it into a decision tree for the same reasons as before.

Algorithm 6 outlines our MCTS pruning approach. We initiate the process by creating a new MDP containing only the initial state and initialize a set of explored states as empty set (line 1, 2). A loop is started in line 3 which runs until line 8. Within the loop, following the MCTS approach, an unexplored state is chosen from the current MDP (line 4). The MDP is expanded from this node by adding all of its children from the original MDP (line 5). A few paths are simulated from this selected state in the original MDP which also accumulate the cost along these paths (line 6). They provide an estimated expected cost for that state, and the PRUNE procedure retains only the top k successors, where k can be an input parameter (line 7). Finally, the selected state in current iteration is added to the set of explored states (line 8). This process repeats until all states within the current MDP have been explored.

Algorithm 6 MCTS based pruning

```
1: $\hat{\mathcal{M}} \leftarrow s_0$
2: $V \leftarrow \emptyset$
3: **while** $V \neq \hat{S}$ **do**
4:     $s \leftarrow$ SELECT($\hat{S} \setminus V$)
5:     $\hat{\mathcal{M}} \leftarrow$ EXPAND($s$)
6:     SIMULATE($G, s$)
7:     PRUNE($\mathcal{M}, s, a$)
8:     $V \leftarrow V \cup s$
```

4.4.2 Implementation and Experiments

We've developed a user-friendly tool featuring a graphical user interface (GUI) to assist engineers in the early satellite development phase. The primary input for this tool includes an architecture graph in `dot` format, fault probabilities for each component, and the cost associated with executing each mode. Beyond the fault isolation method discussed previously, the tool offers additional functionalities that we describe next.

Weakness Report. An initial analysis of the architecture graph generates a weakness report. This report highlights components that may not be entirely isolable, and also shows the probabilities of each mode failing. Engineers can leverage this report to enhance fault tolerance by introducing additional modes or redundancies.

Fault Isolation Strategy. In scenarios where the architecture is single fault-tolerant, the tool can generate and export a fault isolation strategy as a decision tree. If the size of the MDP generated becomes too large, MCTS pruning can be used. For that, a user can specify how many children to retain for each node.

Recovery. The recovery part handles several tasks. By looking at the faulty components, it evaluates which modes are currently available to use. It also selects which modes to use for a given task. Additionally, it determines which components to use while using a mode since there are several configurations to choose from.

4.5 Outlook

Throughout this chapter, we explored how task planning and fault isolation can take advantage of probabilistic verification methods, in particular verification of MDPs. Additionally, we saw how the result of these verification techniques can be used to improve the system. Moreover, in scenarios where absolute optimality is not a strict requirement, we leveraged learning algorithms like MCTS to achieve scalability within the verification process, even if it came at a minor cost to optimality.

The practical application of these approaches on real-world scenarios serves as a strong motivation to utilize them even more.

5 Motion Planning in Unknown Environments

As we briefly mentioned in the previous chapter, *task planning* is the problem of finding a high level plan. *Motion planning*, on the other hand, deals with finding low level plans which incorporate the dynamics of the system and interact with the physical world directly. In this chapter, we discuss the motion planning problem for scLTL objectives in unknown environments and discuss our algorithm given in [GBT+21].

5.1 State of the art

Several studies have tackled the motion planning problem for temporal specifications, including formulas from LTL [BKV10; AAB13; KZ19; VB13], and μ-calculus [KF09].

Syntactic Co-safe LTL. We show the workings of our algorithm for a simplified form of LTL known as the ”syntactic co-safe fragment”. This scLTL fragment is frequently used for motion planning problems [BKV10; AAB13] since they can express a wide spectrum of high-level robotic missions but also belong to the class of formulas that are monitorable [BLS07].

Abstraction. In most sampling-based methods, a common approach involves creating an abstraction of the environment. Additionally, the LTL specification is transformed into an automaton that helps in keeping track of satisfying the specification by constructing a product automaton with the abstraction. In the work by Bhatia et. al. [BKV10], an environmental decomposition based on geometry serves as the abstraction. A high-level plan is derived from this abstraction, which then guides the low-level planner. In contrast, [VB13] constructs the abstraction using the RRG algorithm and dynamically updates the product automaton as new edges are added in the RRG graph.

Multiple RRTs. Another approach, as described in [KZ19], addresses LTL specifications by constructing the RRT graph until an accepting state is reached in the automaton. Then, a new RRT run begins from this accepting point until a cycle is discovered. Continuing in this direction, a bias using the automaton is introduced in [KZ20]. A similar bias is also proposed by Luo et. al. in [LKZ19].

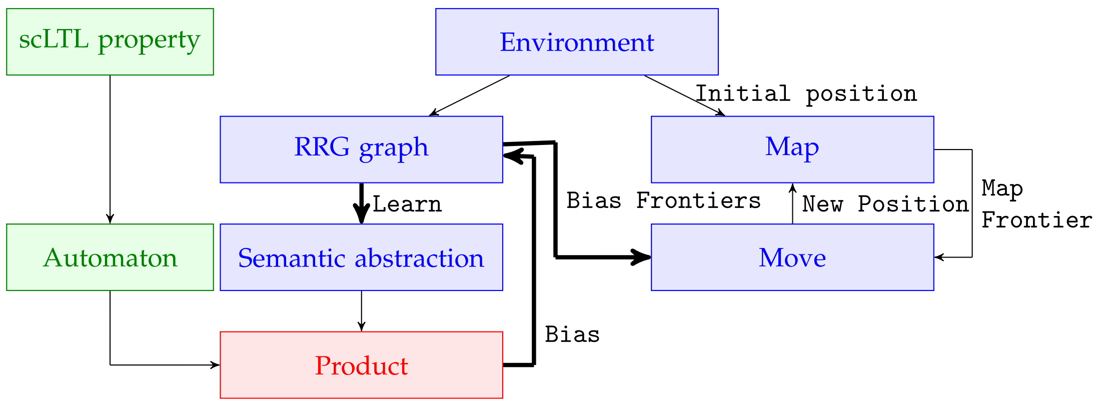


Figure 5.1: Overview of the SAG-RRG algorithm

Unknown Environments. It's important to note that all these approaches assume knowledge of the whole environment. With respect to unknown environments, only the obstacles are unknown beforehand in [KMK+20]. Another paper closely related to our contribution is [AAB13], where information about obstacles and labeling is not known in advance. However, none of these papers tackle the problem where both, obstacles and labels are unknown.

5.2 Contribution: SAG-RRG Algorithm

One naive way to approach this problem would be to explore the whole environment first and then solve LTL motion planning in known environment setting. We, however, want something smarter, where it is possible to also make progress towards satisfying the temporal specification while exploring the environment.

Before we start describing our approach, we need labels, which describe regions of interest in the environment. These labels will then be used in the LTL formula to describe the goal of the robot.

Labeling. Let Σ denote a set of labels (or atomic propositions). A labeling function $\mathcal{L} : \mathcal{X} \rightarrow 2^{\Sigma}$ maps each point in the environment to a set of atomic propositions that hold there. This labeling helps in describing the LTL specification.

5.2.1 Overview of Our Approach

Overview of our approach is highlighted in Figure 5.1. We describe each part of the Figure separately to explain it better.

Automaton. As in any other model-checking inspired approach, we start by converting the scLTL formula to an automaton.

RRG Graph. We start a run of the RRG algorithm which is then used to build the abstraction. Instead of adding one point in one iteration, we add a bunch of points to the graph. For computing *bias frontiers*, see the paragraph **Bias Frontier**.

Abstraction. The abstraction represents the semantic relations present in the environment which means that the set of states is 2^{Σ} It is built using the RRG graph as our foundation. It involves adding transitions that correspond to the edges observed in the environment. To illustrate, for each edge $x_1 \to x_2$ that we incorporate into the graph, we include a corresponding transition $\mathcal{L}(x_1) \to \mathcal{L}(x_2)$ in our abstraction. Additionally, we add supplementary transitions that are *similar* to these original ones (see next paragraph **Learn**). To categorize and specify these different types of transitions within a transition system, we require *Multi-Modal Transition System (MMTS)*. Here, the transitions can be of different modalities. In our particular case, we only require two modalities: *must* and *may*. All the transition described before are added as *must* transitions, since they have been sampled from the environment so they must be present.

Learn. In the abstraction, some *may* transitions are added which are learned from the ones seen in the environment. To elaborate, for each *must* transition of the form $s_1 \to s_2$ in the abstraction, we define the set of *domain of changes*. This domain is defined as follows:

$$DOC(s_1 \to s_2) := L(s_1) \oplus L(s_2)$$

Now, to identify similar transitions to $s_1 \to s_2$, we search for transitions that share an identical *domain of changes*. This set of similar transitions is defined as:

$$\{s_3 \to s_4 \mid DOC(s_3 \to s_4) = DOC(s_1 \to s_2)\}$$

Then all these these similar transitions are added into the abstraction as *may* transitions.

Map. Simultaneously, we build a map of the environment that tracks which parts of the map have been explored. At the beginning, it updates the map by sensing area surrounding the initial position. Later on, the map is updated as the robot moves in the environment.

Map Frontiers. A frontier exploration algorithm (similar to Section 2.5.2) is implemented using the map. Recall that information gain for a map frontier was defined as

$$IG_{map} = size \times f(d)$$

where d is the distance and *size* is the size of the frontier. The best frontier is sent to the MOVE procedure whenever robot wants to move.

Algorithm 7 SAG-RRG

```
Input: Initial point $x_0$, a Büchi automaton $\mathcal{A}$ and step size $\delta$
 1: Initialize semantic abstraction
 2: $V \leftarrow x_0$; $E \leftarrow \emptyset$
 3: curr_pos$\leftarrow x_0$; seen_st$\leftarrow s(x_0)$
 4: while $\neg$AcceptingPath do
 5:     UpdateMap(curr_pos, $r_s$)
 6:     $bias \leftarrow$ Bias(seen_st)
 7:     $t_{symb} \leftarrow \emptyset$; $i \leftarrow 0$
 8:     while $i <$batch_size do
 9:         $x_s, x_{near} \leftarrow$SampleAndExtend($\chi_{free}, V$)
10:         if CollisionFree($x_{near}, x_s$) then
11:             if $(s(x_{near}), s(x_s)) \in bias$ then
12:                 add $x_s$ to bias frontier
13:             else
14:                 continue to next iteration with probability $p$
15:             $E \leftarrow E \cup (x_{near}, x_s)$; $V \leftarrow V \cup x_s$
16:             $t_{symb} \leftarrow t_{symb} \cup (s(x_{near}), s(x_s))$
17:             seen_st $\leftarrow$ seen_st $\cup s(x_s)$
18:             $i \leftarrow i + 1$
19:         for $x \in$ Near($x_s$) do
20:             if CollisionFree($x, x_s$) then
21:                 $E \leftarrow E \cup (x, x_s)$; $V \leftarrow V \cup x$
22:                 $t_{symb} \leftarrow t_{symb} \cup (s(x), s(x_s))$
23:                 if $(s(x), s(x_s)) \in bias$ then
24:                     add $x_s$ to bias frontier
25:     Learn($t_{symb}$)
26:     curr_pos $\leftarrow$ Move
    return accepting path
```

Product. After an iteration of adding edges to the RRG graph and transitions to the abstraction, the product automaton of the abstraction with the property is created. By looking at the accepting runs in the product, new biases and bias frontiers are computed as described next.

Bias. In the product automaton, we find accepting runs using any kind of transitions (*must* or *may*). Note that even if there is no accepting path using the RRG roadmap yet, there can be accepting paths in the abstraction using the *may* transitions. In these accepting paths, the *may* transitions might be a good idea to start looking for something. Also, the closer they are to an accepting state, the better they are. These sets of *may* transitions are therefore returned as an ordered list based on how far they are from an accepting state. Which call this object *bias*, which will also help in figuring out the bias frontiers.

Bias Frontiers. Whenever a transition is sampled from the *bias* list during the RRG iterations, it is added to the set of *bias frontiers*. We define *rank* of a bias frontier as `index` + 1 where `index` is the index of that particular transition in list, e.g. first element of *bias*, which corresponds to the closest transitions to an accepting state has rank 1. We define it's information gain as:

$$IG_{bias} = g(r, d)$$

where d is the distance to the current location and g is some function such that both $g(r, \cdot)$ and $g(\cdot, d)$ are strictly decreasing functions. The best frontier is then sent to the MOVE procedure with the information gain.

Move. After every few iterations of adding points to the RRG graph, the robot finds a new position to go to which maximizes the information gain among all frontiers.

5.2.2 Algorithm

We explain the pseudo code of our algorithm here, as presented in [GBT+21]. Apart from the main procedure, we have LEARN, BIAS, and MOVE procedures which deals with the different aspects of the algorithm as described before.

SAG-RRG. The main procedure shown in Algorithm 7, starts by initializing the abstraction, the RRG graph (V, E), current position, and the set of visited states in the abstraction (line 1-3). A loop (line 4-26) is started in which runs until an accepting path is found. Inside the loop, the map used to store the information about the explored area is updated with respect to the current position and the sensing radius r_s (line 5). In line 6, the bias function is called, which returns an ordered list of transitions which should be treated as advice while sampling points for RRG. Another while loop starts in line 8 for "batch_size" many iterations, where each iteration of this loop is similar to an iteration of the RRG algorithm. It starts by finding a new point x_s and the closest point to it in the graph x_{near}(line 9). Now, if the line joining them is collision free and it's a transition that *bias* suggested, it is added to the set of bias frontiers (line 10-12). Else, with probability $0 < p < 1$, this point is discarded since it was not in the *bias*. Here p depends on how much biasing is required. In the next lines (15-24), it follows the RRG algorithm of adding that edge to the graph and looking in the neighbors to add more edges. Here, simultaneously transitions corresponding to the new edges are added to the set t_{symb} and if a transition from bias in encountered, it is added to the set of bias frontiers

Algorithm 8 Learn

```
1: function LEARN($t_{symb}$)
2:     ADDTOPRODUCT($t_{symb}$, must)
3:     $t_{sim}$ ←FINDSIMILAR($t_{symb}$)
4:     ADDTOPRODUCT($t_{sim}$, may)
```

Learn. The LEARN procedure as shown in Algorithm 8 first adds all new sampled transitions to the product automaton as *must* transitions. Then it computes the set of transitions similar to any transition in t_{symb} and adds them to the product automaton as *may* transitions.

Algorithm 9 Bias

```
1: function BIAS(seen_st)
2:     $bias[0]$ ← transitions ending in accepting states acc_st
3:     reached[0] ← acc_st
4:     reached[1] ← PREIMG(acc_st)
5:     all_reached ← reached[0] ∪ reached[1]
6:     $i \leftarrow 1$
7:     while PREIMG(reached[i]) ⊆ all_reached do
8:         useful_pre ← PREIMG(reached[i]) ∩ seen_st
9:         useful_post ← POSTIMG(useful_pre)∩ reached[i]
10:        $bias[i]$ ← (useful_pre, useful_post) reached[i + 1] ← PREIMG(reached[i])
11:        all_reached ← all_reached ∪ reached[i + 1]
12:        $i \leftarrow i + 1$
    return $bias$
```

Bias. The BIAS procedure (Algorithm 9) starts by adding the set of transitions that end in an accepting state as the first element of bias.

Algorithm 10 Move

```
1: function MOVE
2:     $p_1$ ← FINDBESTMAPFRONTIER
3:     $p_2$ ← FINDBESTBIASFRONTIER
4:     $v$ BEST($p_1, p_2$)
5:     GOTOVERTEX($v$) return $v$
```

Move. Algorithm 10 shows the MOVE procedure, which initially computes the information gain of all map frontiers and bias frontiers and finds the best among them in line 2, 3. Line 4 then finds the vertex v closest to the center of the best frontier. The robot then finds the shortest path to this vertex using a graph search algorithm and

Table 5.1: Mean and standard deviation of different values for 100 randomly generated environments. Each approach ran 3 times for every environment.

	Explore, then plan	**Simultaneous**	**Simult. biased**
Total length	79.1 (7.1)	62.9 (16.5)	32.3 (11.8)
Exploration length	57.8 (4.9)	44.4 (16.6)	31.3 (12.1)
Remaining plan l.	21.3 (5.1)	18.5 (3.4)	1.1 (1.8)
Total Time	9.6 (2.5)	8.3 (3.2)	9.1 (2.4)
RRG size	2313.8 (550.9)	1868.7 (498.2)	1901.4 (301.2)

moves there (line 5).

To show the correctness of our algorithm, we need to prove that it is sound and complete.

Theorem 3 (Soundness)**.** *Algorithm 7 is sound, i.e. any trajectory returned by SAG-RRG satisfies the given scLTL formula ϕ.*

Proof. (Sketch) If an accepting path is found in the product automaton, it can be projected onto the abstraction. Finding a path in the RRG graph corresponding to this abstract path can also be done. This results in a path which satisfies the given specification. □

Theorem 4 (Completeness)**.** *SAG-RRG is asymptotically complete.*

Proof. (Sketch) It follows directly from the convergence and completeness properties of the original RRG and the fact that biasing here is probabilistic therefore allowing it to sample points from everywhere. □

5.2.3 Experimental Results

Our experiments showed significant improvement over the naive approach of first exploring the whole map and then planning in known environment. Our environments were like an office area, where there is a corridor in the middle and 3 rooms on each side of the corridor. Inside each room, a table and a bin is generated randomly and the task of the robot was to reach the bins in all of the rooms. Table 5.1 shows part of our results from [GBT+21]. Here, we see total length, along with the length of path traversed until an accepting path is found (exploration length), and remaining path length required to satisfy the formula. For the first column, the robot explored the whole environment first without actively trying to satisfy the specification. Second column shows the results when robot actively tries to satisfy the specification but does not use our learning and bias. Third and the final column shows results for our algorithm. It is quite evident from the table that in this setting our algorithm performs a lot better than the naive approaches on these environments.

5.3 Outlook

We gave a brand new algorithm for motion planning in an environment with no prior knowledge of obstacles or labels. It learns and exploits similar semantic relations present in the environment. We identify several directions this work can be extended to.

Linear Temporal Logic. First direction for expansion could be solving for the entirety of LTL. The key challenge here is ensuring the robot's movements align with the specified properties. For simple properties like safety it is easy to ensure that the robot doesn't go to unsafe regions, however, monitoring becomes more complex for intricate properties. Techniques from [BLS11] could be useful here for monitoring LTL formulas.

Dynamic Environments. Currently, we only consider stationary obstacles, but in reality, some obstacles move, such as humans and other robots. There have been significant developments in this area e.g. [SLS+07; FPM+23], and it's logical to integrate our methods with these advancements.

On-the-fly Labeling. We also assume that the number of labels is known in advance. However, in unknown environments, this information may be missing. Identifying objects and labels using image recognition is possible, but adding too many labels would result in the state explosion problem. Therefore, it would be crucial to identify the useful labels which can help in satisfying the specification. It's worth noting that relying solely on the labels in the specification may not promote the learning of semantic relationships.

Bibliography

[AAB13] A. I. Medina Ayala, S. B. Andersson, and C. Belta. "Temporal logic motion planning in unknown environments". In: *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2013, pp. 5279–5284. DOI: 10.1109/IROS.2013.6697120 (cited on pages 30, 31).

[AAP+07] Alessandro Abate, Saurabh Amin, Maria Prandini, John Lygeros, and Shankar Sastry. "Computational Approaches to Reachability Analysis of Stochastic Hybrid Systems". In: *Hybrid Systems: Computation and Control*. Ed. by Alberto Bemporad, Antonio Bicchi, and Giorgio Buttazzo. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 4–17 (cited on page 17).

[AJJ+20] Pranav Ashok, Mathias Jackermeier, Pushpak Jagtap, Jan Křetínský, Maximilian Weininger, and Majid Zamani. "DtControl: Decision Tree Learning Algorithms for Controller Representation". In: *Proceedings of the 23rd International Conference on Hybrid Systems: Computation and Control*. HSCC '20. Sydney, New South Wales, Australia: Association for Computing Machinery, 2020. DOI: 10.1145/3365365.3382220 (cited on page 11).

[AKL+10] Alessandro Abate, Joost-Pieter Katoen, John Lygeros, and Maria Prandini. "Approximate Model Checking of Stochastic Hybrid Systems". In: *European Journal of Control* 16.6 (2010), pp. 624–641. DOI: https://doi.org/10.3166/ejc.16.624-641 (cited on pages 16, 17).

[APL+08] Alessandro Abate, Maria Prandini, John Lygeros, and Shankar Sastry. "Probabilistic reachability and safety for controlled discrete time stochastic hybrid systems". In: *Automatica* 44.11 (2008), pp. 2724–2734. DOI: https://doi.org/10.1016/j.automatica.2008.03.027 (cited on page 16).

[BBC+14] Benjamin Bittner, Marco Bozzano, Alessandro Cimatti, Regis De Ferluc, Marco Gario, Andrea Guiotto, and Yuri Yushtein. "An Integrated Process for FDIR Design in Aerospace". In: *Model-Based Safety and Assessment*. Ed. by Frank Ortmeier and Antoine Rauzy. Cham: Springer International Publishing, 2014, pp. 82–95 (cited on page 26).

[BCC+14] Tomáš Brázdil, Krishnendu Chatterjee, Martin Chmelík, Vojtěch Forejt, Jan Křetínský, Marta Kwiatkowska, David Parker, and Mateusz Ujma. "Verification of Markov Decision Processes Using Learning Algorithms". In: *Automated Technology for Verification and Analysis*. Ed. by Franck Cassez and Jean-François Raskin. Cham: Springer International Publishing, 2014, pp. 98–114 (cited on pages 10, 21).

[BCC+15] Tomáš Brázdil, Krishnendu Chatterjee, Martin Chmelík, Andreas Fellner, and Jan Křetínský. "Counterexample Explanation by Learning Small Strategies in Markov Decision Processes". In: *Computer Aided Verification*. Ed. by Daniel Kroening and Corina S. Păsăreanu. Cham: Springer International Publishing, 2015, pp. 158–177 (cited on page 11).

[BCM18] David Basin, Cas Cremers, and Catherine Meadows. "Model Checking Security Protocols". In: *Handbook of Model Checking*. Ed. by Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem. Cham: Springer International Publishing, 2018, pp. 727–762. DOI: 10.1007/978-3-319-10575-8_22 (cited on page 22).

[Ber75] D. Bertsekas. "Convergence of discretization procedures in dynamic programming". In: *IEEE Transactions on Automatic Control* 20.3 (1975), pp. 415–419. DOI: 10.1109/TAC.1975.1100984 (cited on pages 16, 17).

[BFG+93] R.I. Bahar, E.A. Frohm, C.M. Gaona, G.D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. "Algebraic decision diagrams and their applications". In: *Proceedings of 1993 International Conference on Computer Aided Design (ICCAD)*. 1993, pp. 188–191. DOI: 10.1109/ICCAD.1993.580054 (cited on page 11).

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008 (cited on pages 1, 8, 12).

[BKV10] Amit Bhatia, Lydia E. Kavraki, and Moshe Y. Vardi. "Sampling-based motion planning with temporal goals". In: *2010 IEEE International Conference on Robotics and Automation*. 2010, pp. 2689–2696. DOI: 10.1109/ROBOT.2010.5509503 (cited on page 30).

[BLS07] Andreas Bauer, Martin Leucker, and Christian Schallhart. "The Good, the Bad, and the Ugly, But How Ugly Is Ugly?" In: *Runtime Verification*. Ed. by Oleg Sokolsky and Serdar Taşıran. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 126–138 (cited on page 30).

[BLS11] Andreas Bauer, Martin Leucker, and Christian Schallhart. "Runtime Verification for LTL and TLTL". In: *ACM Trans. Softw. Eng. Methodol.* 20.4 (Sept. 2011). DOI: 10.1145/2000799.2000800 (cited on page 37).

[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. DOI: 10.1007/978-3-642-35746-6_1 (cited on page 2).

[Cou07] Rémi Coulom. “Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search”. In: *Computers and Games*. Ed. by H. Jaap van den Herik, Paolo Ciancarini, and H. H. L. M. (Jeroen) Donkers. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 72–83 (cited on page 2).

[CPC03] Alessandro Cimatti, Charles Pecheur, and Roberto Cavada. “Formal Verification of Diagnosability via Symbolic Model Checking”. In: Jan. 2003, pp. 363–369 (cited on page 27).

[CZ11] Edmund M. Clarke and Paolo Zuliani. “Statistical Model Checking for Cyber-Physical Systems”. In: *Automated Technology for Verification and Analysis*. Ed. by Tevfik Bultan and Pao-Ann Hsiung. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 1–12 (cited on page 22).

[DH03] Daisy Dobrijevic and Elizabeth Howell. “Columbia Disaster: What happened and what NASA learned”. In: *Space.com* (Feb. 1, 2003) (cited on page 1).

[Dij59] E. W. Dijkstra. “A note on two problems in connexion with graphs”. In: *Numerische Mathematik* 1.1 (Dec. 1959), pp. 269–271. DOI: `10.1007/BF01386390` (cited on page 13).

[DJK+17] Christian Dehnert, Sebastian Junges, Joost-Pieter Katoen, and Matthias Volk. “A Storm is Coming: A Modern Probabilistic Model Checker”. In: *Computer Aided Verification*. Ed. by Rupak Majumdar and Viktor Kunčak. Cham: Springer International Publishing, 2017, pp. 592–600 (cited on page 11).

[EAA10] Mohamed Elboukhari, Mostafa Azizi, and Abdelmalek Azizi. “Verification of Quantum Cryptography Protocols by Model Checking”. In: *International Journal of Network Security & Its Applications* 2 (Oct. 2010). DOI: `10.5121/ijnsa.2010.2404` (cited on page 22).

[Fak21] Hengameh Fakhravar. “Application of Failure Modes and Effects Analysis in the Engineering Design Process”. In: *CoRR* abs/2101.05444 (2021) (cited on page 26).

[Far02] Berndt Farwer. “ω-Automata”. In: *Automata Logics, and Infinite Games: A Guide to Current Research*. Ed. by Erich Grädel, Wolfgang Thomas, and Thomas Wilke. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 3–21. DOI: `10.1007/3-540-36387-4_1` (cited on page 12).

[FC19] Hongfei Fu and Krishnendu Chatterjee. “Termination of Nondeterministic Probabilistic Programs”. In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Constantin Enea and Ruzica Piskac. Cham: Springer International Publishing, 2019, pp. 468–490 (cited on page 16).

[FKN+11] Vojtěch Forejt, Marta Kwiatkowska, Gethin Norman, and David Parker. "Automated Verification Techniques for Probabilistic Systems". In: *Formal Methods for Eternal Networked Software Systems: 11th International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM 2011, Bertinoro, Italy, June 13-18, 2011. Advanced Lectures*. Ed. by Marco Bernardo and Valérie Issarny. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 53–113. DOI: 10.1007/978-3-642-21455-4_3 (cited on page 9).

[FPM+23] Mark Nicholas Finean, Luka Petrović, Wolfgang Merkt, Ivan Marković, and Ioannis Havoutis. "Motion planning in dynamic environments using context-aware human trajectory prediction". In: *Robotics and Autonomous Systems* 166 (2023), p. 104450. DOI: https://doi.org/10.1016/j.robot.2023.104450 (cited on page 37).

[FSP+16] Orlando Ferrante, Eelco Scholte, Claudio Pinello, Alberto Ferrari, Leonardo Mangeruca, Cong Liu, and Christos Sofronis. "A Methodology for Increasing the Efficiency and Coverage of Model Checking and its Application to Aerospace Systems". In: *SAE International Journal of Aerospace* 9 (Sept. 2016), pp. 140–150. DOI: 10.4271/2016-01-2053 (cited on page 22).

[GBT+21] Kush Grover, Fernando S Barbosa, Jana Tumova, and Jan Křetínský. "Semantic Abstraction-Guided Motion Planning for scLTL Missions in Unknown Environments". In: *Robotics: Science and Systems XVII, Virtual Event, July 12-16, 2021*. RSS Foundation-Robotics Science & Systems Foundation. Virtual, July 2021. DOI: 10.15607/RSS.2021.XVII.090 (cited on pages 4, 5, 13, 30, 34, 36, 71).

[GHK04] Carlos Guestrin, Milos Hauskrecht, and Branislav Kveton. "Solving Factored MDPs with Continuous and Discrete Variables". In: *Proceedings of the 20th Conference on Uncertainty in Artificial Intelligence*. UAI '04. Banff, Canada: AUAI Press, 2004, pp. 235–242 (cited on pages 16, 17).

[GKM+22a] Kush Grover, Jan Křetínský, Tobias Meggendorfer, and Maximilian Weininger. "Anytime Guarantees for Reachability in Uncountable Markov Decision Processes". In: *33rd International Conference on Concurrency Theory, CONCUR 2022, September 12-16, 2022, Warsaw, Poland*. Ed. by Bartek Klin, Slawomir Lasota, and Anca Muscholl. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Sept. 2022, 11:1–11:20. DOI: 10.4230/LIPIcs.CONCUR.2022.11 (cited on pages 3–5, 15, 17, 19, 49).

[GKM+22b] Kush Grover, Jan Křetínský, Tobias Meggendorfer, and Maximilian Weininger. *Anytime Guarantees for Reachability in Uncountable Markov Decision Processes*. 2022. DOI: 10.48550/arXiv.2008.04824 (cited on pages 16, 19–21).

[GNT04] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: Theory and Practice*. The Morgan Kaufmann Series in Artificial Intelligence. Amsterdam: Morgan Kaufmann, 2004 (cited on page 22).

[HAS+14] Khaza Anuarul Hoque, Otmane Ait Mohamed, Yvon Savaria, and Claude Thibeault. "Early Analysis of Soft Error Effects for Aerospace Applications Using Probabilistic Model Checking". In: *Formal Techniques for Safety-Critical Systems*. Ed. by Cyrille Artho and Peter Csaba Ölveczky. Cham: Springer International Publishing, 2014, pp. 54–70 (cited on page 22).

[Has12] Hado van Hasselt. "Reinforcement Learning in Continuous State and Action Spaces". In: *Reinforcement Learning: State-of-the-Art*. Ed. by Marco Wiering and Martijn van Otterlo. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 207–251. DOI: 10.1007/978-3-642-27645-3_7 (cited on page 16).

[Hen96] T.A. Henzinger. "The theory of hybrid automata". In: *Proceedings 11th Annual IEEE Symposium on Logic in Computer Science*. 1996, pp. 278–292. DOI: 10.1109/LICS.1996.561342 (cited on page 1).

[HHP+13] Giray Havur, Kadir Haspalamutgil, Can Palaz, Esra Erdem, and Volkan Patoglu. "A case study on the Tower of Hanoi challenge: Representation, reasoning and execution". In: *2013 IEEE International Conference on Robotics and Automation*. 2013, pp. 4552–4559. DOI: 10.1109/ICRA.2013.6631224 (cited on page 23).

[HJS+20] William B. Haskell, Rahul Jain, Hiteshi Sharma, and Pengqian Yu. "A Universal Empirical Dynamic Programming Algorithm for Continuous State MDPs". In: *IEEE Transactions on Automatic Control* 65.1 (2020), pp. 115–129. DOI: 10.1109/TAC.2019.2907414 (cited on page 16).

[HKP+98] Thomas A. Henzinger, Peter W. Kopke, Anuj Puri, and Pravin Varaiya. "What's Decidable about Hybrid Automata?" In: *Journal of Computer and System Sciences* 57.1 (1998), pp. 94–124. DOI: https://doi.org/10.1006/jcss.1998.1581 (cited on page 16).

[HNR68] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". In: *IEEE Transactions on Systems Science and Cybernetics* 4.2 (1968), pp. 100–107. DOI: 10.1109/TSSC.1968.300136 (cited on page 13).

[Jau01] Luc Jaulin. "Path Planning Using Intervals and Graphs". In: *Reliable Computing* 7 (Feb. 2001), pp. 1–15. DOI: 10.1023/A:1011400431065 (cited on pages 2, 13).

[JJG+19] Manfred Jaeger, Peter Gjøl Jensen, Kim Guldstrand Larsen, Axel Legay, Sean Sedwards, and Jakob Haahr Taankvist. "Teaching Stratego to Play Ball: Optimal Synthesis for Continuous Space MDPs". In: *Automated Technology for Verification and Analysis*. Ed. by Yu-Fang Chen, Chih-Hong Cheng, and Javier Esparza. Cham: Springer International Publishing, 2019, pp. 81–97 (cited on page 16).

[JK11] Benjamin Johnson and Hadas Kress-Gazit. “Probabilistic Analysis of Correctness of High-Level Robot Behavior with Sensor Error”. In: *Robotics: Science and Systems*. 2011 (cited on page 23).

[JZK+19] Yu-qian Jiang, Shi-qi Zhang, Piyush Khandelwal, and Peter Stone. “Task planning in robotics: an empirical comparison of PDDL- and ASP-based systems”. In: *Frontiers of Information Technology & Electronic Engineering* 20 (Mar. 2019), pp. 363–373. DOI: `10.1631/FITEE.1800514` (cited on pages 2, 23).

[KF09] Sertac Karaman and Emilio Frazzoli. “Sampling-based motion planning with deterministic μ-calculus specifications”. In: *Proceedings of the 48h IEEE Conference on Decision and Control (CDC) held jointly with 2009 28th Chinese Control Conference*. 2009, pp. 2222–2229. DOI: `10.1109/CDC.2009.5400278` (cited on pages 2, 30).

[KF11] Sertac Karaman and Emilio Frazzoli. “Incremental Sampling-based Algorithms for Optimal Motion Planning”. In: *Robotics: Science and Systems VI*. The MIT Press, Aug. 2011. DOI: `10.7551/mitpress/9123.003.0038` (cited on pages 2, 13).

[KGA+22] Jonis Kiesbye, Kush Grover, Pranav Ashok, and Jan Křetínský. “Planning via model checking with decision-tree controllers”. In: *2022 International Conference on Robotics and Automation, ICRA 2022, Philadelphia, PA, USA, May 23-27, 2022*. IEEE, May 2022, pp. 4347–4354. DOI: `10.1109/ICRA46639.2022.9811980` (cited on pages 3–5, 22, 81).

[KGK23] Jonis Kiesbye, Kush Grover, and Jan Křetínský. “Model Checking for Proving and Improving Fault Tolerance of Satellites”. In: *2023 IEEE Aerospace Conference*. 2023, pp. 1–9. DOI: `10.1109/AERO55745.2023.10115801` (cited on pages 3, 5, 91).

[KL11] Leslie Pack Kaelbling and Tomás Lozano-Pérez. “Hierarchical task and motion planning in the now”. In: *2011 IEEE International Conference on Robotics and Automation*. 2011, pp. 1470–1477. DOI: `10.1109/ICRA.2011.5980391` (cited on page 22).

[KMK+20] Yiannis Kantaros, Matthew Malencia, Vijay Kumar, and George J. Pappas. “Reactive Temporal Logic Planning for Multiple Robots in Unknown Environments”. In: *2020 IEEE International Conference on Robotics and Automation (ICRA)*. 2020, pp. 11479–11485. DOI: `10.1109/ICRA40945.2020.9197570` (cited on page 31).

[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 (cited on pages 2, 11).

[KS92] Henry Kautz and Bart Selman. "Planning as Satisfiability". In: *Proceedings of the 10th European Conference on Artificial Intelligence*. ECAI '92. Vienna, Austria: John Wiley & Sons, Inc., 1992, pp. 359–363 (cited on page 23).

[KZ19] Yiannis Kantaros and Michael M. Zavlanos. "Sampling-Based Optimal Control Synthesis for Multirobot Systems Under Global Temporal Tasks". In: *IEEE Transactions on Automatic Control* 64.5 (2019), pp. 1916–1931. DOI: 10.1109/TAC.2018.2853558 (cited on page 30).

[KZ20] Yiannis Kantaros and Michael M Zavlanos. "STyLuS*: A Temporal Logic Optimal Control Synthesis Algorithm for Large-Scale Multi-Robot Systems". In: *The International Journal of Robotics Research* 39.7 (2020), pp. 812–836. DOI: 10.1177/0278364920913922 (cited on page 30).

[LaV98] Steven M. LaValle. "Rapidly-exploring random trees : a new tool for path planning". In: *The annual research report* (1998) (cited on pages 2, 13).

[Lee59] C. Y. Lee. "Representation of switching circuits by binary-decision programs". In: *The Bell System Technical Journal* 38.4 (1959), pp. 985–999. DOI: 10.1002/j.1538-7305.1959.tb01585.x (cited on page 11).

[LKZ19] Xusheng Luo, Yiannis Kantaros, and Michael M. Zavlanos. "An Abstraction-Free Method for Multirobot Temporal Logic Optimal Control Synthesis". In: *IEEE Transactions on Robotics* 37 (2019), pp. 1487–1507 (cited on page 30).

[LL05] Lihong Li and Michael L. Littman. "Lazy Approximation for Solving Continuous Finite-Horizon MDPs". In: *Proceedings of the 20th National Conference on Artificial Intelligence - Volume 3*. AAAI'05. Pittsburgh, Pennsylvania: AAAI Press, 2005, pp. 1175–1180 (cited on page 16).

[LWA+10] M. Lahijanian, J. Wasniewski, S. B. Andersson, and C. Belta. "Motion planning and control from temporal logic specifications with probabilistic satisfaction guarantees". In: *2010 IEEE International Conference on Robotics and Automation*. 2010, pp. 3227–3232. DOI: 10.1109/ROBOT.2010.5509686 (cited on page 22).

[Mal11] Tariq Malik. "Rocket Carrying New NASA Climate Satellite Likely Crashed Into Pacific Ocean". In: *Space.com* (Mar. 4, 2011) (cited on page 1).

[MGN] Sascha Müller, Andreas Gerndt, and Thomas Noll. "Synthesizing FDIR Recovery Strategies from Non-Deterministic Dynamic Fault Trees". In: *AIAA SPACE and Astronautics Forum and Exposition*. DOI: 10.2514/6.2017-5163 (cited on page 27).

[MLG05] H. Brendan McMahan, Maxim Likhachev, and Geoffrey J. Gordon. "Bounded Real-Time Dynamic Programming: RTDP with Monotone Upper Bounds and Performance Guarantees". In: *Proceedings of the 22nd International Conference on Machine Learning*. ICML '05. Bonn, Germany: Association for Computing Machinery, 2005, pp. 569–576. DOI: 10.1145/1102351.1102423 (cited on page 10).

[MMG+20] Sascha Müller, Liana Mikaelyan, Andreas Gerndt, and Thomas Noll. "Synthesizing and optimizing FDIR recovery strategies from fault trees". In: *Science of Computer Programming* 196 (2020), p. 102478. DOI: https://doi.org/10.1016/j.scico.2020.102478 (cited on pages 2, 26, 27).

[MMR08] Francisco S. Melo, Sean P. Meyn, and M. Isabel Ribeiro. "An Analysis of Reinforcement Learning with Function Approximation". In: *Proceedings of the 25th International Conference on Machine Learning*. ICML '08. Helsinki, Finland: Association for Computing Machinery, 2008, pp. 664–671. DOI: 10.1145/1390156.1390240 (cited on page 16).

[Mul63] David E. Muller. "Infinite sequences and finite machines". In: *Proceedings of the Fourth Annual Symposium on Switching Circuit Theory and Logical Design (swct 1963)*. 1963, pp. 3–16. DOI: 10.1109/SWCT.1963.8 (cited on page 12).

[NPM+14] Srinivas Nedunuri, Sailesh Prabhu, Mark Moll, Swarat Chaudhuri, and Lydia E. Kavraki. "SMT-based synthesis of integrated task and motion plans from plan outlines". In: *2014 IEEE International Conference on Robotics and Automation (ICRA)*. 2014, pp. 655–662. DOI: 10.1109/ICRA.2014.6906924 (cited on page 23).

[Pnu77] Amir Pnueli. "The temporal logic of programs". In: *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*. 1977, pp. 46–57. DOI: 10.1109/SFCS.1977.32 (cited on pages 1, 11).

[Pri18] Rob Price. "A self-driving Uber car hit and killed a woman in the first known autonomous-vehicle death". In: *Business Insider* (Mar. 19, 2018) (cited on page 1).

[Put94] Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. 1st. USA: John Wiley & Sons, Inc., 1994 (cited on pages 1, 6, 9).

[Rin12] Jussi Rintanen. "Engineering Efficient Planners with SAT". In: *Proceedings of the 20th European Conference on Artificial Intelligence*. ECAI'12. Montpellier, France: IOS Press, 2012, pp. 684–689 (cited on page 23).

[SA11] Sadegh Esmaeil Zadeh Soudjani and Alessandro Abate. "Adaptive Gridding for Abstraction and Verification of Stochastic Hybrid Systems". In: *2011 Eighth International Conference on Quantitative Evaluation of SysTems*. 2011, pp. 59–68. DOI: 10.1109/QEST.2011.16 (cited on pages 16, 17).

[SEJ+16] Salomon Sickert, Javier Esparza, Stefan Jaax, and Jan Křetínský. "Limit-Deterministic Büchi Automata for Linear Temporal Logic". In: *Computer Aided Verification*. Ed. by Swarat Chaudhuri and Azadeh Farzan. Cham: Springer International Publishing, 2016, pp. 312–332 (cited on page 12).

[SFR+14] Siddharth Srivastava, Eugene Fang, Lorenzo Riano, Rohan Chitnis, Stuart Russell, and Pieter Abbeel. "Combined task and motion planning through an extensible planner-independent interface layer". In: *2014 IEEE International Conference on Robotics and Automation (ICRA)*. 2014, pp. 639–646. DOI: 10.1109/ICRA.2014.6906922 (cited on page 23).

[SJJ20] Hiteshi Sharma, Mehdi Jafarnia-Jahromi, and Rahul Jain. "Approximate Relative Value Learning for Average-reward Continuous State MDPs". In: *Proceedings of The 35th Uncertainty in Artificial Intelligence Conference*. Ed. by Ryan P. Adams and Vibhav Gogate. Vol. 115. Proceedings of Machine Learning Research. PMLR, July 2020, pp. 956–964 (cited on page 16).

[SLS+07] Zvi Shiller, Frederic Large, Sepanta Sekhavat, and Christian Laugier. "Motion Planning in Dynamic Environments". In: *Autonomous Navigation in Dynamic Environments*. Ed. by Christian Laugier and Raja Chatila. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 107–119. DOI: 10.1007/978-3-540-73422-2_5 (cited on page 37).

[TMK+13] Ilya Tkachev, Alexandru Mereacre, Joost-Pieter Katoen, and Alessandro Abate. "Quantitative Automata-Based Controller Synthesis for Non-Autonomous Stochastic Hybrid Systems". In: *Proceedings of the 16th International Conference on Hybrid Systems: Computation and Control*. HSCC '13. Philadelphia, Pennsylvania, USA: Association for Computing Machinery, 2013, pp. 293–302. DOI: 10.1145/2461328.2461373 (cited on pages 16, 17).

[VB13] Cristian Ioan Vasile and Calin Belta. "Sampling-based temporal logic path planning". In: *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2013, pp. 4817–4822. DOI: 10.1109/IROS.2013.6697051 (cited on pages 2, 30).

[WB93] Ronald J. Williams and Leemon C. Baird. "Analysis of Some Incremental Variants of Policy Iteration: First Steps Toward Understanding Actor-Cr". In: 1993 (cited on page 9).

[WGM+21] Maximilian Weininger, Kush Grover, Shruti Misra, and Jan Křetínský. "Guaranteed Trade-Offs in Dynamic Information Flow Tracking Games". In: *2021 60th IEEE Conference on Decision and Control (CDC), Austin, TX, USA, December 14-17, 2021*. IEEE, Dec. 2021, pp. 3786–3793. DOI: 10.1109/CDC45484.2021.9683447 (cited on page 5).

[Yam97] B. Yamauchi. "A frontier-based approach for autonomous exploration". In: *Proceedings 1997 IEEE International Symposium on Computational Intelligence in Robotics and Automation CIRA'97. 'Towards New Computational Principles for Robotics and Automation'*. 1997, pp. 146–151. DOI: 10.1109/CIRA.1997.613851 (cited on page 13).

[ZRF+19] Xingyu Zhao, Valentin Robu, David Flynn, Fateme Dinmohammadi, Michael Fisher, and Matt Webster. “Probabilistic Model Checking of Robots Deployed in Extreme Environments”. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 33.01 (July 2019), pp. 8066–8074. DOI: 10.1609/aaai.v33i01.33018066 (cited on page 22).

Part I

First Author Publications

A Anytime Guarantees for Reachability in Uncountable Markov Decision Processes

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

Kush Grover, Jan Křetínský, Tobias Meggendorfer, and Maximilian Weininger. "Anytime Guarantees for Reachability in Uncountable Markov Decision Processes". In: *33rd International Conference on Concurrency Theory, CONCUR 2022, September 12-16, 2022, Warsaw, Poland*. Ed. by Bartek Klin, Slawomir Lasota, and Anca Muscholl. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Sept. 2022, 11:1–11:20. DOI: 10.4230/LIPIcs.CONCUR.2022.11

Summary. In this work, we introduce two anytime algorithms designed to solve the reachability problem in MDPs with uncountable state and action spaces. The general reachability problem is undecidable however by imposing certain assumptions, we can extend existing iterative methods designed for finite MDPs to work here. Our primary contribution in this paper is the identification of these necessary assumptions, striving to keep them as minimal as possible, thus pushing the boundaries of systems that can be reliably analyzed.

We present two algorithms in this regard, extensions of *Value Iteration* and *Bounded Real-time Dynamic Programming*. The value iteration algorithm provides a lower bound on the value, requiring fewer assumptions in comparison to previous approaches which give guarantees. In contrast, the BRTDP algorithm provides both lower and upper bounds but demands more assumptions due to the inclusion of the additional upper bound. Importantly, both of these algorithms are *anytime* algorithms, continuously improving their estimations and ultimately converging to the true value in the limit.

Contributions of thesis author. The author played an important role in the composition and revision of the manuscript. He actively participated in joint discussions and contributed significantly to the creation of techniques presented in the paper. Noteworthy individual contributions include figuring out some of the assumptions required for the algorithm, along with the help in proving the correctness of the algorithm.

Anytime Guarantees for Reachability in Uncountable Markov Decision Processes

Kush Grover
Technische Universität München, Germany

Jan Křetínský
Technische Universität München, Germany

Tobias Meggendorfer
Institute of Science and Technology Austria, Wien, Austria

Maximilian Weininger
Technische Universität München, Germany

Abstract

We consider the problem of approximating the reachability probabilities in Markov decision processes (MDP) with uncountable (continuous) state and action spaces. While there are algorithms that, for special classes of such MDP, provide a sequence of approximations converging to the true value in the limit, our aim is to obtain an algorithm with guarantees on the precision of the approximation.

As this problem is undecidable in general, assumptions on the MDP are necessary. Our main contribution is to identify sufficient assumptions that are as weak as possible, thus approaching the "boundary" of which systems can be correctly and reliably analyzed. To this end, we also argue why each of our assumptions is necessary for algorithms based on processing finitely many observations.

We present two solution variants. The first one provides converging lower bounds under weaker assumptions than typical ones from previous works concerned with guarantees. The second one then utilizes stronger assumptions to additionally provide converging upper bounds. Altogether, we obtain an *anytime* algorithm, i.e. yielding a sequence of approximants with known and iteratively improving precision, converging to the true value in the limit. Besides, due to the generality of our assumptions, our algorithms are very general templates, readily allowing for various heuristics from literature in contrast to, e.g., a specific discretization algorithm. Our theoretical contribution thus paves the way for future practical improvements without sacrificing correctness guarantees.

2012 ACM Subject Classification Mathematics of computing → Markov processes; Mathematics of computing → Continuous mathematics; Computing methodologies → Continuous models

Keywords and phrases Uncountable system, Markov decision process, discrete-time Markov control process, probabilistic verification, anytime guarantee

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2022.11

Related Version *Full Version*: `https://arxiv.org/abs/2008.04824` [17]

Funding *Kush Grover*: The author has been supported by the DFG research training group GRK 2428 *ConVeY*.
Maximilian Weininger: The author has been partially supported by DFG projects 383882557 *Statistical Unbounded Verification (SUV)* and 427755713 *Group-By Objectives in Probabilistic Verification (GOPro)*.

1 Introduction

The standard formalism for modelling systems with both non-deterministic and probabilistic behaviour are Markov decision processes (MDP) [43]. In the context of many applications such as cyber-physical systems, states and actions are used to model real-valued phenomena like position or throttle. Consequently, the state space and the action space may be uncountably infinite. For example, the intervals $[a, b] \times [c, d] \subseteq \mathbb{R}^2$ can model a safe area for a robot to move in or a set of available control inputs such as acceleration and steering angle. This gives rise to MDP with uncountable state- and action-spaces (sometimes called controlled discrete-time Markov process [51, 52] or discrete-time Markov control process [11, 28]), with applications ranging from modelling a Mars rover [10, 24], over water reservoir control [36] and warehouse storage management [38], to energy control [51], and many more [41].

Although systems modelled by MDP are often safety-critical, the analysis of uncountable systems is so complex that practical approaches for verification and controller synthesis are usually based on "best effort" learning techniques, for example *reinforcement learning.* While efficient in practice, these methods guarantee, even in the best case, convergence to the true result only in the limit, e.g. [40], or for increasingly precise discretization, e.g. [51, 32]. In line with the tradition of learning and to make the analysis more feasible, the typical objectives considered for MDP are either finite-horizon [37, 3] or discounted properties [18, 53, 25], together with restrictive assumptions. Note that when it comes to approximation, discounted properties effectively are finite-horizon. In contrast, ensuring safety of a reactive system or a certain probability to satisfy its mission goals requires an *unbounded* horizon and reduces to optimizing the reachability probabilities. Moreover, the safety-critical context requires *reliable* bounds on the probability, not an approximation with *unknown* precision.

In this paper, we provide the first provably correct anytime algorithm for (unbounded) reachability in uncountable MDP. As an *anytime* algorithm, it can at every step of the execution return correct lower and upper bounds on the true value. Moreover, these bounds gradually converge to the true value, allowing approximation up to an arbitrary precision. Since the problem is undecidable, the core of our contribution is identifying sufficient conditions on the uncountable MDP to allow for approximation.

Our *primary goal* is to provide *conditions as weak as possible*, thereby pushing towards the boundary of which systems can be analyzed provably correctly. To this end, we do not rely on any particular representation of the system. Nonetheless, for classical scenarios, and, in particular, for finite MDP, our conditions are mostly satisfied trivially.

Our *secondary goal* is to derive the respective algorithms as an *extension of value iteration* (VI) [29, 43], while *avoiding drawbacks of discretization*-based approaches. VI is a de facto standard method for numerical analysis of finite MDP, in particular with reachability objectives, regarded as practically efficient and allowing for heuristics avoiding the exploration of the complete state space, e.g. [9]. Interestingly, even for finite MDP, anytime VI algorithms with precision guarantees are quite recent [9, 19, 4, 44, 22]. Previous to that, the most used model checkers could return arbitrarily wrong results [19]. Providing VI with precision guarantees for general uncountable MDP is thus worthwhile on its own. Finally, while discretization is conceptually simple, we prefer to provide a solution that avoids the need to introduce arbitrary boundaries through gridding the whole state space and, moreover, instead utilizes information from one "cell" of the grid in other places, too.

To summarize, while algorithmic aspects form an important motivation, our primary contribution is theoretical: an explicit and complete set of generic assumptions allowing for guarantees, disregarding practical efficiency at this point. Consequently, while our approach lays foundations for further, more tailored approaches, it is not to be seen as a competitor to the existing practical, best-effort techniques, as these aim for a completely different goal.

Our Contribution. In this work, we provide the following:

Section 3: A set of assumptions that allow for computing converging *lower* bounds on the reachability probability in MDP with uncountable state and action spaces. We discuss in detail why they are weaker than usual, necessary, and applicable to typically considered systems. With these assumptions, we extend the standard (convergent but precision-ignorant) VI to this general setting.

Section 4: An additional set of assumptions that yield the first *anytime* algorithm, i.e. with provable bounds on the precision/error of the result, converging to 0. We combine the preceding algorithm with the technique of *bounded real-time dynamic programming (BRTDP)* [39] and provide also converging *upper* bounds on the reachability probability.

Section 5: A discussion of theoretical extensions and practical applications.

Related work. For detailed *theoretical* treatment of reachability and related problems on uncountable MDP, see e.g. [52, 11]. Reachability on uncountable MDP generalizes numerous problems known to be undecidable. For example, we can encode the halting problem of (probabilistic) Turing machines by encoding the tape content as real value. Similarly, almost-sure termination of probabilistic programs (undecidable [33]) is a special case of reachability on general uncountable MDP (see e.g. [16]). As precise reachability analysis is undecidable even for non-stochastic linear hybrid systems [26], many works turn their attention to more relaxed notions such as δ-reachability, e.g. [48], and/or employ many assumptions.

In order to obtain precision bounds, we assume that the *value* function, mapping states to their reachability probability, is *Lipschitz continuous* (and that we know the Lipschitz constant). This is slightly *weaker* than the classical approach of assuming Lipschitz continuity of the *transition* function (and knowledge of the constant), e.g. [2, 49]. In particular, these assumptions (i) imply our assumption (as we show in [17, App. B.2.1]) and (ii) are used even in the simpler settings of finite-horizon and discounted reward scenarios [5, 2, 49, 51] or even more restricted settings to obtain practical efficiency, e.g. [35]. In contrast to our approach, they are not anytime algorithms and require treatment of the whole state space.

To provide context, we outline how continuity is used (explicitly or implicitly) in related work and mention their respective results. Firstly, [25, 47] assume Lipschitz continuity, *but not* explicit knowledge of the constant. In essence, these approaches solve the problem by successively increasing internal parameters.The parameters then eventually cross a bound implied by the Lipschitz constant, yielding an "eventual correctness". In particular, they provide "convergence in the limit" or "probably approximately correct" results, but no bounds on the error or the convergence rate; these would depend on knowledge of the constant.

Secondly, [18, 40, 2, 49, 51] (and our work) assume Lipschitz continuity *and* knowledge of the constant. Relying on the constant being provided externally, these works derive guarantees. Previously, the guarantees given are weaker than our convergent anytime bounds: Either convergence in the limit [40] or a bound on a discretization error, relativized to sub-optimal strategies [18] or bounded horizon [2, 49, 51].

Several of the above mentioned works employ *discretization* [18, 2, 49, 51]. This method is quite general, but obtaining any bounds on the error requires continuity assumptions [1]. Further, there are works that use other assumptions: [23, 24] use *reinforcement learning* methods to tackle reachability and more general problems, without any continuity assumption. However, they do not provide any guarantees. See [53] for a detailed exposition of similar approaches. Assuming an abstraction is given, *abstraction and bisimulation* approaches, e.g. [21, 20], provide guarantees, but only on the lower bounds. With significant assumptions on the system's structure, *symbolic* approaches [37, 54, 45, 14] may even obtain exact solutions.

2 Preliminaries

In this section, we recall basics of probabilistic systems and set up the notation. As usual, $\mathbb{N}$ and $\mathbb{R}$ refer to the (positive) natural numbers and real numbers, respectively. For a set S, $\mathbb{1}_S$ denotes its *characteristic function*, i.e. $\mathbb{1}_S(x) = 1$ if $x \in S$ and 0 otherwise. We write $S^\star$ and S^ω to refer to the set of finite and infinite sequences comprising elements of S, respectively.

We assume familiarity with basic notions of measure theory, e.g. *measurable set* or *measurable function*, as well as probability theory, e.g. *probability spaces* and *measures* [8]. For a measure space X with sigma-algebra Σ_X, $\Pi(X)$ denotes the set of all probability measures on X. For a measure $\mu \in \Pi(X)$, we write $\mu(Y) = \int \mathbb{1}_Y \, d\mu$ to denote the mass of a measurable set $Y \in \Sigma_X$ (also called *event*). For two probability measures μ and ν, the *total variation distance* is defined as $\delta_{TV}(\mu, \nu) := 2 \cdot \sup_{Y \in \Sigma_X} |\mu(Y) - \nu(Y)|$. Some event happens *almost surely* (a.s.) w.r.t. some measure μ if it happens with probability 1. We write $\operatorname{supp}(\mu)$ to denote the *support* of the probability measure μ.

▶ Remark 1. It is surprisingly difficult to give a well-defined notion of support for measures in general. Intuitively, $\operatorname{supp}(\mu)$ describes the "smallest" set which μ assigns a value of 1. However, this is not well-defined for general measures. We discuss these issues and a proper definition in [17, App. E]. Throughout this work, similar subtle issues related to measure theory arise. For the sake of readability, these are mostly delegated to footnotes or the appendix of the full version [17], and readers may safely skip over these points.

We work with *Markov decision processes* (MDP) [43], a widely used model to capture both non-determinism and probability. We consider uncountable state and action spaces.

▶ **Definition 2.** *A* (continuous-space, discrete-time) Markov decision process (MDP) *is a tuple $\mathcal{M} = (S, Act, Av, \Delta)$, where S is a compact set of* states *(with topology $\mathcal{T}_S$ and Borel σ-algebra $\Sigma_S = \mathfrak{B}(\mathcal{T}_S)$), Act is a compact set of* actions *(with topology $\mathcal{T}_{Act}$ and Borel σ-algebra $\Sigma_{Act} = \mathfrak{B}(\mathcal{T}_{Act})$), $Av \colon S \to \Sigma_{Act} \setminus \{\emptyset\}$ assigns to every state a non-empty, measurable, and compact set of* available actions*, and $\Delta \colon S \times Act \to \Pi(S)$ is a* transition function *that for each state s and (available) action $a \in Av(s)$ yields a probability measure over successor states (i.e. a Markov Kernel). An MDP is called* finite *if $|S| < \infty$ and $|Act| < \infty$.*

See [43, Sec. 2.3] and [6, Chp. 9] for a more detailed discussion on the technical considerations arising from uncountable state and action spaces. Note that we assume the set of available actions to be non-empty. This means that the system can never get "stuck" in a degenerate state without successors. *Markov chains* are a special case of MDP where $|Av(s)| = 1$ for all $s \in S$, i.e. a completely probabilistic system without any non-determinism. Our presented methods thus are directly applicable to Markov chains as well.

Given a measure $\mu \in \Pi(X)$ and a measurable function $f \colon X \to \mathbb{R}$ mapping elements of a set X to real numbers, we write $\mu\langle f \rangle := \int f(x) \, d\mu(x)$ to denote the integral of f with respect to μ. For example, $\Delta(s,a)\langle f \rangle$ denotes the expected value $\mathbb{E}_{s' \sim \Delta(s,a)} f(s')$ of $f \colon S \to \mathbb{R}$ over the successors of s under action a. Moreover, abusing notation, for some set of state $S' \subseteq S$ and function $Av' \colon S' \to Act$, we write $S' \times Av' = \{(s,a) \mid s \in S', a \in Av'(s)\}$ to denote the set of state-action pairs with states from S' under Av'.

An *infinite path* in an MDP is some infinite sequence $\rho = s_1 a_1 s_2 a_2 \cdots \in (S \times Av)^\omega$, such that for every $i \in \mathbb{N}$ we have $s_{i+1} \in \operatorname{supp}(\Delta(s_i, a_i))$. A *finite path* (or *history*) $\varrho = s_1 a_1 s_2 a_2 \ldots s_n \in (S \times Av)^\star \times S$ is a non-empty, finite prefix of an infinite path of length $|\varrho| = n$, ending in state s_n, denoted by $last(\varrho)$. We use $\rho(i)$ and $\varrho(i)$ to refer to the i-th state in an (in)finite path. We refer to the set of finite (infinite) paths of an MDP $\mathcal{M}$ by $\mathsf{FPaths}_\mathcal{M}$ ($\mathsf{Paths}_\mathcal{M}$). Analogously, we write $\mathsf{FPaths}_{\mathcal{M},s}$ ($\mathsf{Paths}_{\mathcal{M},s}$) for all (in)finite paths starting in s.

In order to obtain a probability measure, we first need to eliminate the non-determinism. This is done by a so-called *strategy* (also called *policy*, *controller*, or *scheduler*). A strategy on an MDP $\mathcal{M} = (S, Act, Av, \Delta)$ is a function $\pi \colon \mathsf{FPaths}_{\mathcal{M}} \to \Pi(Act)$, s.t. $\mathrm{supp}(\pi(\varrho)) \subseteq Av(last(\varrho))$. The set of all strategies is denoted by $\Pi_{\mathcal{M}}$. Intuitively, a strategy is a "recipe" describing which step to take in the current state, given the evolution of the system so far.

Given an MDP $\mathcal{M}$, a strategy $\pi \in \Pi_{\mathcal{M}}$, and an initial state s_0, we obtain a measure on the set of infinite paths $\mathsf{Paths}_{\mathcal{M}}$, which we denote as $\mathsf{Pr}^{\pi}_{\mathcal{M},s_0}$. See [43, Sec. 2] for further details. Thus, given a measurable set $A \subseteq \mathsf{Paths}_{\mathcal{M}}$, we can define its maximal probability starting from state s_0 under any strategy by $\mathsf{Pr}^{\sup}_{\mathcal{M},s_0}[A] := \sup_{\pi \in \Pi_{\mathcal{M}}} \mathsf{Pr}^{\pi}_{\mathcal{M},s_0}[A]$. Depending on the structure of A it may be the case that no optimal strategy exists and we have to resort to the supremum instead of the maximum. This may already arise for finite MDP, see [12].

For an MDP $\mathcal{M} = (S, Act, Av, \Delta)$ and a set of *target states* $T \subseteq S$, *(unbounded) reachability* refers to the set $\Diamond T = \{\rho \in \mathsf{Paths}_{\mathcal{M}} \mid \exists i \in \mathbb{N}.\ \rho(i) \in T\}$, i.e. all paths which eventually reach T. The set $\Diamond T$ is measurable if T is measurable [51, Sec. 3.1], [52, Sec. 2].

Now, it is straightforward to define the *maximal reachability problem* of a given set of states. Given an MDP $\mathcal{M}$, target set T, and state s_0, we are interested in computing the maximal probability of eventually reaching T, starting in state s_0. Formally, we want to compute the *value* of the state s_0, defined as $\mathcal{V}(s_0) := \mathsf{Pr}^{\sup}_{\mathcal{M},s_0}[\Diamond T] = \sup_{\pi \in \Pi_{\mathcal{M}}} \mathsf{Pr}^{\pi}_{\mathcal{M},s_0}[\Diamond T]$. This state value function satisfies a straightforward fixed point equation, namely

$$\mathcal{V}(s) = 1 \quad \text{if } s \in T \qquad \mathcal{V}(s) = \sup\nolimits_{a \in Av(s)} \Delta(s,a)\langle \mathcal{V} \rangle \qquad \text{otherwise.} \tag{1}$$

Moreover, $\mathcal{V}$ is the *smallest* fixed point of this equation [6, Prop. 9.8, 9.10], [52, Thm. 3]. In our approach, we also deal with values of state-action pairs $(s,a) \in S \times Av$, where $\mathcal{V}(s,a) := \Delta(s,a)\langle \mathcal{V} \rangle$. Intuitively, this represents the value achieved by choosing action a in state s and then moving optimally. Clearly, we have that $\mathcal{V}(s) = \sup_{a \in Av(s)} \mathcal{V}(s,a)$. See [15, Sec. 4] for a discussion of reachability on finite MDP and [52] for the general case.

In this work, we are interested in *approximate solutions* due to the following two reasons. Firstly, obtaining precise solutions for MDP is difficult already under strict assumptions and undecidable in our general setting.(1) We thus resort to approximation, allowing for much lighter assumptions. Secondly, by considering approximation we are able to apply many different optimization techniques, potentially leading to algorithms which are able to handle real-world systems, which are out of reach for precise algorithms even for finite MDP [9].

We are interested in two types of approximations. Firstly, we consider approximating the value function in the limit, without knowledge about how close we are to the true value. This is captured by a semi-decision procedure for queries of the form $\mathsf{Pr}^{\sup}_{\mathcal{M},s}[\Diamond T] > \xi$ for a threshold $\xi \in [0,1]$. We call this problem $\mathsf{ApproxLower}$. Secondly, we consider the variant where we are given a precision requirement $\varepsilon > 0$ and obtain ε-optimal values (l,u), i.e. values with $\mathcal{V}(s_0) \in [l,u]$ and $0 \leq u - l < \varepsilon$. We refer to this variant as $\mathsf{ApproxBounds}$.

3 Converging Lower Bounds

In this section, we present the first set of assumptions, enabling us to compute *converging lower bounds* on the true value, solving the $\mathsf{ApproxLower}$ problem. In Section 3.1, we discuss each assumption in detail and argue on an intuitive level why it is necessary by means of

(1)For example, one can encode the tape of a Turing machine into the binary representation of a real number and reduce the halting problem to a reachability query.

counterexamples. With the assumptions in place, in Section 3.2 we then present our first algorithm, also introducing several ideas we employ again in the following section.

Our assumptions and algorithms are motivated by *value iteration* (VI) [29], which we briefly outline. In a nutshell, VI boils down to repeatedly applying an iteration operator to a value vector v_n. For example, the canonical value iteration for reachability on finite MDP starts with $v_0(s) = 1$ for all $s \in T$ and 0 otherwise and then iterates

$$v_{n+1}(s) = \max_{a \in Act(s)} \sum_{s' \in S} \Delta(s, a, s') \cdot v_n(s') \tag{2}$$

for all $s \notin T$. The vector v_n converges monotonically from below to the true value for all states. We mention two important points. Firstly, the iteration can be applied "asynchronously". Instead of updating all states in every iteration, we can pick a single state and only update its value. The values v_n still converge to the correct value as long as all states are updated infinitely often. Secondly, instead of storing a value per state, we can store a value for each state-action pair and obtain the state value as the maximum of these values. Both points are a technical detail for finite MDP, however they play an essential role in our uncountable variant. See [17, App. A.1] for more details on VI for finite MDP.

In the uncountable variant of Equation (2), v is a function, $Act(s)$ is potentially uncountable, and the sum is replaced by integration. As in this setting the problem is undecidable, naturally we have to employ some assumptions. Our goal is to sufficiently imitate the essence of Equation (2), obtaining convergence without being overly restrictive. In particular, we want to (i) represent (an approximation of) v_n using finite memory, (ii) safely approximate the maximum and integration, and (iii) select appropriate points to update v_n.

3.1 Assumptions

Before discussing each assumption in detail, we first put them into context. As we argue in the following, most of our assumptions typically hold implicitly. Still, by stating even basic computability assumptions in a form as weak as possible, we avoid "hidden" assumptions, e.g. by assuming that the state space is a subset of $\mathbb{R}^d$. Two of our assumptions are more restrictive, namely **Assumption C**: **Value Lipschitz Continuity** (Section 3.1.3) and, introduced later, **Assumption D**: **Absorption** (Section 4.1.2). However, they are also often used in related works, as we detail in the respective sections. Moreover, in light of previous results, the necessity of restrictive assumptions is to be expected: Computing bounds is hard or even undecidable already for very restricted classes. Aside from the discussion in the introduction, we additionally mention two further cases. In the setting of probabilistic programs (which are a very special case of uncountable MDP), deciding almost sure termination for a fixed initial state (which is a severely restricted subclass of reachability on uncountable MDP without non-determinism) is an actively researched topic with recent advances, see e.g. [30, 31], and shown to be Π_2^0-complete [33], i.e. highly undecidable. In [27] and the references therein, the authors present (un-)decidability results for hybrid automata, which are a special case of uncountable MDP *without any stochastic dynamics* (flow transitions can be modelled as actions indicating the delay). As such, it is to be expected that the general class of models we consider has to be pruned very strictly in order to hope for any decidability results.

▶ Remark 3. As already mentioned, we want to provide assumptions which are as general as possible. Importantly, we avoid (unnecessarily) assuming any particular representation of the system. Our motivation is to ultimately identify the boundary of what is necessary to derive guarantees. While our assumptions are motivated by VI and built around Equation (2), we note that being able to represent the state values and evaluate (some aspect of) the transition dynamics intuitively are a necessity for *any* method dealing with such systems. We do not claim that our framework of assumptions is the only way to approach the problem, instead we provide arguments why it is a sensible way to do so.

3.1.1 A: Basic Assumptions (Asm. A1-A4)

We first present a set of basic computability assumptions (**A1**-**A4**). These are essential, since for uncountable systems even the simplest computations are intractable without any assumptions. More specifically, such systems cannot be given explicitly (due to their infinite size), but instead have to be described symbolically by, e.g., differential equations. Thus, we necessarily require some notion of computability and structural properties for each part of this symbolic description. And indeed, each assumption essentially corresponds to one part of the MDP description (**Metric Space** to $S \times \mathit{Act}$, **Maximum Approximation** to Av, **Transition Approximation** to Δ, and **Target Computability** to T). They are weak and hold on practically all commonly considered systems (see [17, App. B.1]). In particular, finite MDP and discrete components are trivially subsumed by considering the discrete metric.

A1: Metric Space S and Act are metric spaces with (computable) metrics d_S and $\mathrm{d}_{\mathit{Act}}$, respectively, and $\mathrm{d}_\times$ is a compatible[(2)] metric on the space of state-action pairs $S \times \mathit{Av}$,

A2: Maximum Approximation For each state s and computable Lipschitz $f : \mathit{Av}(s) \to [0,1]$, the value $\max_{a \in \mathit{Av}(s)} f(a)$ can be under-approximated to arbitrary precision.

A3: Transition Approximation For each state-action pair (s,a) and Lipschitz $g : S \to [0,1]$ which can be under-approximated to arbitrary precision, the successor expectation $\Delta(s,a)\langle g \rangle$ can be under-approximated to arbitrary precision.

A4: Target Computability The target set T is decidable, i.e. we are given a computable predicate which, given a state s, decides whether $s \in T$.

We denote the approximations for **A2** and **A3** by $\textsc{Approx}_{\leq}$, i.e. given a pair (s,a) and functions f, g as in the assumptions, we write (abusing notation) $\textsc{Approx}_{\leq}(\max_{a \in \mathit{Av}(s)} f(a), \varepsilon)$ and $\textsc{Approx}_{\leq}(\Delta(s,a)\langle g \rangle, \varepsilon)$ for approximation of the respective values up to precision ε, i.e. $0 \leq \max_{a \in \mathit{Av}(s)} f(a) - \textsc{Approx}_{\leq}(\max_{a \in \mathit{Av}(s)} f(a), \varepsilon) \leq \varepsilon$ and analogous for $\Delta(s,a)\langle g \rangle$. Note that **A2** and **A3** are satisfied if we can sample densely in $\mathit{Av}(s)$ and approximate $\Delta(s,a)$.

3.1.2 B: Sampling (Asm. B.VI)

As there are uncountably many states, we are unable to explicitly update all of them at once and instead update values asynchronously. Moreover, as there may also be uncountably many actions, we instead store and update the values of state-action pairs. Together, we need to pick state-action pairs to update. We delegate this choice to a selection mechanism $\textsc{GetPair}$, an oracle for state-action pairs. We allow for $\textsc{GetPair}$ to be "stateful", i.e. the sampled state-action pair may depend on previously returned pairs. This is required in, for example, round-robin or simulation-based approaches. We only require a basic notion of fairness in order to guarantee that we do not miss out on any information. Note the additional identifier **.VI** (*value iteration*) on the assumption name; later on, a similar, but weaker variant (**B.BRTDP**) is introduced.

(2) For two pairs (s,a) and (s,a') we have that $k \cdot \mathrm{d}_{\mathit{Act}}(a,a') \leq \mathrm{d}_\times((s,a),(s,a')) \leq K \cdot \mathrm{d}_{\mathit{Act}}(a,a')$ for some constants $k, K \geq 0$, analogous for d_S, achieved by, e.g. $\mathrm{d}_\times((s,a),(s',a')) := \mathrm{d}_S(s,s') + \mathrm{d}_{\mathit{Act}}(a,a')$.

B.VI: State-Action Sampling Let $S^{\Diamond} = \{last(\varrho) \mid \varrho \in \mathsf{FPaths}_{\mathcal{M},s}\}$ the set of all reachable states. Then, for any $\varepsilon > 0$, $s \in S^{\Diamond}$, and $a \in Av(s)$ we have that GetPair eventually yields a pair (s', a') with $\mathrm{d}_\times((s,a),(s',a')) < \varepsilon$ and $\delta_{TV}(\Delta(s,a), \Delta(s',a')) < \varepsilon$ a.s.[3]

Essentially, this means that GetPair provides a way to "exhaustively" generate all behaviours of the system up to a precision of ε. This fairness assumption is easily satisfied under usual conditions. For example, if $S \times Av$ is a bounded subset of $\mathbb{R}^d$, we can randomly sample points in that space or consider increasingly dense grids. Alternatively, if we can sample from the set of actions and from the distributions of Δ, GetPair can be implemented by sampling paths of random length, following random actions. Note that we can view the procedure as a "template": Instead of requiring a concrete method to acquire pairs to update, we leave this open for generality; we discuss implications of this in Sections 5.1 and 5.3.

The requirement on total variation may seem unnecessary, especially given that we will also assume continuity. However, otherwise we could, for example, miss out on solitary actions which are the "witnesses" for a state's value: suppose that $Av(s) = [0,1]$ and $\Delta(s,0)$ moves to the goal, while $\Delta(s,a)$ just loops back to s. Only selecting actions close to $a = 0$ w.r.t. the product metric is not sufficient to observe that we can move to the goal. Note that this would not be necessary if we assumed continuity of the transition function – selecting "nearby" actions then also yields "similar" behaviour.

3.1.3 C: Lipschitz Continuity

Finally, we present our already advertised continuity assumption. For simplicity, we give it in its strict form and discuss relaxations later in Section 5.2. Intuitively, Lipschitz continuity allows us to extrapolate the behaviour of the system from a single state to its surroundings.

C: Value Lipschitz Continuity The value functions $\mathcal{V}(s)$ and $\mathcal{V}(s,a)$ are Lipschitz continuous with *known* constants C_S and $C_\times$, i.e. for all $s, s' \in S$ and $a \in Av(s), a' \in Av(s')$ we have

$$|\mathcal{V}(s) - \mathcal{V}(s')| \leq C_S \cdot \mathrm{d}_S(s,s') \qquad |\mathcal{V}(s,a) - \mathcal{V}(s',a')| \leq C_\times \cdot \mathrm{d}_\times((s,a),(s',a'))$$

This requirement may seem quite restrictive at first glance. Indeed, it is the only one in this section to not usually hold on "standard" systems. However, in order to obtain any kind of (provably correct) bounds, some notion of continuity is elementary, since otherwise we cannot safely extrapolate from finitely many observations to an uncountable set. The immediately arising questions are (i) why *Lipschitz* continuity is necessary compared to, e.g., regular or uniform continuity, and (ii) why *knowledge of the Lipschitz constant* is required. For the first point, note that we want to be able to extrapolate from values assigned to a single state to its immediate surroundings. While continuity means that the values in the surroundings do not "jump", it does not give us any way of bounding the rate of change, and this rate may grow arbitrarily (for example, consider the continuous but not Lipschitz function $\sin(\frac{1}{x})$ for $x > 0$). So, also relating to the second point, without knowledge of the Lipschitz constant, regular continuity and Lipschitz continuity are (mostly) equivalent from a computational perspective: The function does not have discontinuities, but we cannot safely estimate the rate of change in general. To illustrate this point further, we give an intuitive example.

[3] Technically, it is sufficient to satisfy this property on any subset of $S^{\Diamond}$ which only differs from it up to measure 0. More precisely, we only require that this assumption holds for $S^{\Diamond} = \mathrm{supp}(\mathsf{Pr}^{\sup}_{\mathcal{M},s})$, i.e. the set of all reachable paths with non-zero measure. We omit this rather technical notion and the discussion it entails in order to avoid distracting from the central results of this work.

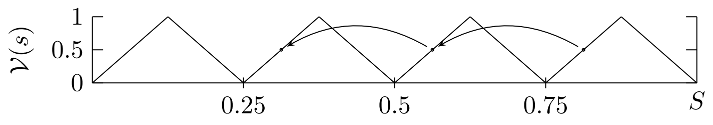


Figure 1 The value function of Example 4, showing that knowledge of the constant is important.

▶ **Example 4.** We construct an MDP with a periodic, Lipschitz continuous value function, as illustrated in Figure 1 and formally defined below. Intuitively, for a given period width w (e.g. 0.25) and a periodic function f (e.g. a triangle function), a state s between 0 and w moves to a target or sink with probability $f(s)$. All larger states $s \geq w$ transition to $s - w$ with probability 1. The value function thus is periodic and Lipschitz continuous, see Figure 1 for a possible value function and [17, App. B.2.3] for a formal definition.

For a finite number of samples, we can choose f and w such that all samples achieve a value of 1. Nevertheless, we cannot conclude anything about states we have not sampled yet: *Without knowledge of the constant, we cannot extrapolate from samples.*

We note the underlying connection to the Nyquist-Shannon sampling theorem [46, Thm. 1]. Intuitively, the theorem states that, for a function that contains no frequencies higher than W, it is completely determined by giving its ordinates at a series of points spaced $0.5 \cdot W$ apart. If we know the Lipschitz constant, this gives us a way of bounding the "frequency" of the value function, and thus allows us to determine it by sampling a finite number of points. On the other hand, without the Lipschitz constant, we do not know the frequency and cannot judge whether we are "undersampling".

Since we do not assume any particular representation of the transition system, we cannot derive such constants in general. Instead, these would need to be obtained by, e.g., domain knowledge, or tailored algorithms. As in previous approaches [18, 40, 2, 49, 51], we thus resort to assuming that we are given this constant, offloading this (highly non-trivial) step. Recall that Lipschitz continuity of the transition function implies Lipschitz continuity of the value function (see [17, App. B.2.1]), but can potentially be checked more easily.

3.2 Assumptions Applied: Value Iteration Algorithm

Before we present our new algorithm, we explain how our assumptions allow us to lift VI to the uncountable domain. Contrary to the finite state setting, we are unable to store precise values for each state explicitly, since there are uncountably many states. Hence, the algorithm exploits the Lipschitz-continuity of the value function as follows. Assume that we know that the value of a state s is bounded from below by a value l, i.e. $\mathcal{V}(s) \geq l$. Then, by Lipschitz-continuity of $\mathcal{V}$, we know that the value of a state s' is bounded by $l - \mathrm{d}_S(s, s') \cdot C_S$. More generally, if we are given a finite set of states $\mathsf{Sampled}$ with correct lower bounds $\widehat{\mathsf{L}} \colon \mathsf{Sampled} \to [0, 1]$, we can safely extend these values to the whole state space by

$$\mathsf{L}(s) := \max_{s' \in \mathsf{Sampled}} \left(\widehat{\mathsf{L}}(s') - C_S \cdot \mathrm{d}_S(s, s') \right).$$

Since $\mathcal{V}(s) \geq \widehat{\mathsf{L}}(s)$ for all $s \in \mathsf{Sampled}$, we have $\mathcal{V}(s) \geq \mathsf{L}(s)$ for all $s \in S$, i.e. $\mathsf{L}(\cdot)$ is a valid lower bound. We thus obtain a lower bound for all of the uncountably many states, described symbolically as a combination of finitely many samples. See Figure 2 for an illustration.

This is sufficient to deal with Markov chains, but for MDPs we additionally need to take care of the (potentially uncountably many) actions. Recall that value iteration updates state values with the maximum over available actions, $v_{n+1}(s) = \max_{a \in Av(s)} \Delta(s, a)\langle v_n \rangle$.

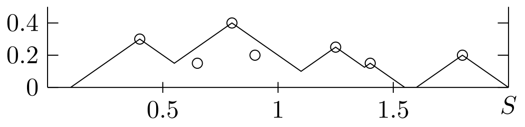


Figure 2 Example of the function extension on the set $[0, 2]$ with a Lipschitz constant of $C_S = 1$. Dots represent stored values in $\widehat{\mathsf{L}}$, while the solid line represents the extrapolated function L. Note that it is possible to have $\widehat{\mathsf{L}}(s) < \mathsf{L}(s)$, as seen in the graph.

Algorithm 1 The Value Iteration (VI) Algorithm for MDPs with general state- and action-spaces.

```
Input: ApproxLower query with threshold $\xi$, satisfying A1–A4, B.VI and C.
Output: yes, if $\mathcal{V}(s_0) > \xi$.
1: Sampled $\leftarrow \emptyset$, t $\leftarrow 1$                                  ▷ Initialize
2: while $\text{APPROX}_{\leq}(\mathsf{L}(s_0), \text{PRECISION}(\mathsf{t})) \leq \xi$ do
3:     $(s, a) \leftarrow$ GETPAIR                                  ▷ Sample state-action pair
4:     if $s \in T$ then   $\widehat{\mathsf{L}}(s, \cdot) \leftarrow 1$                    ▷ Handle target states
5:     else   $\widehat{\mathsf{L}}(s, a) \leftarrow \text{APPROX}_{\leq}(\Delta(s, a)\langle \mathsf{L} \rangle, \text{PRECISION}(\mathsf{t}))$      ▷ Update $\widehat{\mathsf{L}}$
6:     Sampled $\leftarrow$ Sampled $\cup \{(s, a)\}$, t $\leftarrow$ t $+ 1$
7: return yes
```

This is straightforward to compute when there are only finitely many actions, but in the uncountable case obtaining $\mathsf{L}(s) = \sup_{a \in Av(s)} \mathsf{L}(s, a)$ is much more involved. We apply the idea of Lipschitz continuity again, storing values for a set Sampled of state-action pairs instead of only states. We bound the value of every state-action pair by

$$\mathsf{L}(s, a) := \max_{(s', a') \in \mathsf{Sampled}} \left(\widehat{\mathsf{L}}(s', a') - \mathrm{d}_\times((s, a), (s', a')) \cdot C_\times \right) \tag{3}$$

Observe that $\mathsf{L}(s, a)$ is computable and Lipschitz-continuous as well, so by **Maximum Approximation** we can approximate the bound of any state, i.e. $\mathsf{L}(s) = \max_{a \in Av(s)} \mathsf{L}(s, a)$, based on such a finite set of values assigned to state-action pairs. (Recall that $Av(s)$ is compact and $\mathsf{L}(s, a)$ continuous, hence the maximum is attained.) Consequently, we can also under-approximate $\Delta(s, a)\langle \mathsf{L} \rangle$ by **Transition Approximation**. To avoid clutter, we omit the following two special cases in the definition of $\mathsf{L}(s, a)$: Firstly, if $\mathsf{Sampled} = \emptyset$, we naturally set $\mathsf{L}(s, a) = 0$. Secondly, if all pairs (s', a') are too far away for a sensible estimate, i.e. if Equation (3) was yielding $\mathsf{L}(s, a) < 0$, we also set $\mathsf{L}(s, a)$ to 0.

We present VI for MDPs with general state- and action-spaces in Algorithm 1. It depends on $\text{PRECISION}(\mathsf{t})$, a sequence of precisions converging to zero in the limit, e.g. $\text{PRECISION}(\mathsf{t}) = \frac{1}{\mathsf{t}}$. The algorithm executes the main loop until the current approximation of the lower bound of the initial state $\mathsf{L}(s_0) = \max_{a \in Av(s_0)} \mathsf{L}(s_0, a)$ exceeds the given threshold ξ. Inside the loop, the algorithm updates state-action pairs yielded by GETPAIR. For target states, the lower bound is set to 1. Otherwise, we set the bound of the selected pair to an approximation of the expected value of L under the corresponding transition. Here is the crucial difference to VI in the finite setting: Instead of using Equation (2), we have to use Equation (3) and $\text{APPROX}_{\leq}$, the approximations that exist by assumption, see Section 3.1.1. Since $\text{PRECISION}(\mathsf{t})$ converges to zero, the approximations eventually get arbitrarily fine. The procedure $\text{PRECISION}(\mathsf{t})$ may be adapted heuristically in order to speed up computation. For example, it may be beneficial to only approximate up to 0.01 precision at first to quickly get a rough overview. We show that Algorithm 1 is correct, i.e. the stored values (i) are lower bounds and (ii) converge to the true values in [17, App. E.1]. Here, we only provide a sketch, illustrating the main steps.

▶ **Theorem 5.** *Algorithm 1 is correct under Assumptions* ***A1*–*A4*, *B.VI*,** *and* ***C***, *i.e. it outputs* `yes` *iff* $\mathcal{V}(s) > \xi$.

Proof sketch. First, we show that $\mathsf{L}_\mathsf{t}(s) \leq \mathsf{L}_{\mathsf{t}+1}(s) \leq \mathcal{V}(s)$ by simple induction on the step. Initially, we have $\mathsf{L}_1(s) = 0$, obviously satisfying the condition. The updates in Lines 4 and 5 both keep correctness, i.e. $\mathsf{L}_{\mathsf{t}+1}(s) \leq \mathcal{V}(s)$, proving the claim.

Since L_t is monotone as argued above, its limit for $\mathsf{t} \to \infty$ is well defined, denoted by L_∞. By **State-Action Sampling**, the set of accumulation points of s_t contains all reachable states $S^\Diamond$. We then prove that L_∞ satisfies the fixed point equation Equation (1). For this, we use the second part of the assumption on GetPair, namely that for every $(s, a) \in S^\Diamond \times Av$ we get a converging subsequence $(s_{\mathsf{t}_k}, a_{\mathsf{t}_k})$ where additionally $\Delta(s_{\mathsf{t}_k}, a_{\mathsf{t}_k})$ converges to $\Delta(s, a)$ in total variation. Intuitively, since infinitely many updates occur infinitely close to (s, a), its limit lower bound $\mathsf{L}_\infty(s, a)$ agrees with the limit of the updates values $\lim_{k\to\infty} \Delta(s_{\mathsf{t}_k}, a_{\mathsf{t}_k})\langle \mathsf{L}_{\mathsf{t}_k} \rangle$. Since L_∞ satisfies the fixed point equation and is less or equal to the value function $\mathcal{V}$, we get the result, since $\mathcal{V}$ is the smallest fixed point. ◀

4 Converging Upper Bounds

In this section, we present the second set of assumptions, allowing us to additionally compute converging *upper* bounds. With both lower and upper bounds, we can quantify the progress of the algorithm and, in particular, terminate the computation once the bounds are sufficiently close. Therefore, instead of only providing a semi-decision procedure for reachability, this algorithm is able to determine the maximal reachability probability up to a given precision. Thus, we obtain the first algorithm able to handle such general systems with guarantees on its result. We again present our assumptions together with a discussion of their necessity (Section 4.1), and then introduce the subsequent algorithm and prove its correctness (Section 4.2). As expected, obtaining this additional information also requires additional assumptions. On the other hand, quite surprisingly, we can use the additional information of upper bounds to actually *speed up* the computation, as discussed in Section 5.3.

As before, our approach is inspired by algorithms for finite MDP, in this case by *Bounded Real-Time Dynamic Programming* (BRTDP) [39, 9]. BRTDP uses the same update equations as VI, but iterates both lower and upper bounds. A major contribution of [9] was to solve the long standing open problem of how to deal with *end components*. These parts of the state space prevent convergence of the upper bounds by introducing additional fixpoints of Equation (1). We direct the interested reader to [17, App. A.2] for further details on BRTDP and insights on the issue of end components. In the uncountable setting, these issues arise as well alongside several other, related problems, which we discuss in Section 4.1.2.

4.1 Assumptions

The basic assumptions **A1**–**A4** as well as **Lipschitz continuity** (Assumption **C**) remain unchanged. For **Maximum Approximation** (**A2**) and **Transition Approximation** (**A3**), we additionally require that we are able to *over*-approximate the respective results. The respective assumptions are denoted by **A5** and **A6**, respectively, and both over-approximations by Approx$_\geq$. Further, we only require a weakened variant of **State-Action Sampling**, now called Assumption **B.BRTDP** instead of Assumption **B.VI**. Finally, there is the new Assumption **D** called **Absorption**, addressing the aforementioned issue of end components.

4.1.1 B: Weaker Sampling (Asm. B.BRTDP)

We again assume a GetPair oracle, but, perhaps surprisingly, with *weaker* assumptions. Instead of requiring it to return "all" actions, we only require it to yield "optimal" actions, respective to a given state-action value function. We first introduce some notation. Intuitively, we want GetPair to yield actions which are optimal with respect to the *upper bounds* computed by the algorithm. However, these upper bounds potentially change after each update. Thus, assume that $f_n \colon S \times Av \to [0, 1]$ is an arbitrary sequence of computable, Lipschitz continuous, (point-wise) monotone decreasing functions, assigning a value to each state-action pair, and set $\mathcal{F} = (f_1, f_2, \dots)$. For each state $s \in S$, set

$$Av_{\mathcal{F}}(s) := \{a \in Av(s) \mid \forall \varepsilon > 0.\ \forall N \in \mathbb{N}.\ \exists n > N.\ \max_{a' \in Av(s)} f_n(s, a') - f_n(s, a) < \varepsilon\},$$

i.e. actions that infinitely often achieve values arbitrarily close to the optimum of f_n. Let $S^{\Diamond}_{\mathcal{F}} = \{last(\varrho) \mid \varrho \in \mathsf{FPaths}_{\mathcal{M},s_0} \cap (S \times Av_{\mathcal{F}})^* \times S\}$ be the set of all states reachable using these optimal actions.(4) Essentially, we require that GetPair samples densely in $S^{\Diamond}_{\mathcal{F}} \times Av_{\mathcal{F}}$.

B.BRTDP: State-Action Sampling For any $\varepsilon > 0$, $\mathcal{F}$ as above, $s \in S^{\Diamond}_{\mathcal{F}}$, and $a \in Av_{\mathcal{F}}(s)$ we have that GetPair a.s. eventually yields a pair (s', a') with $\mathrm{d}_\times((s, a), (s', a')) < \varepsilon$ and $\delta_{TV}(\Delta(s, a), \Delta(s', a')) < \varepsilon$.

While this new variant may seem much more involved, it is *weaker* than its previous variant, since $Av_{\mathcal{F}}(s) \subseteq Av(s)$ for each $s \in S$ and thus also $S^{\Diamond}_{\mathcal{F}} \subseteq S^{\Diamond}$. As such, it also allows for more practical optimizations, which we briefly discuss in Section 5.3.

4.1.2 D: Absorption

We present our most specific assumption. While it is *not needed* for correctness, we require it for convergence of the upper bounds to the value and thus for termination of the algorithm.

D: Absorption There exists a *known and decidable* set R (called *sink*) such that $\mathcal{V}(s) = 0$ for all $s \in R$. Moreover, for any $s \in S$ and strategy π we have $\mathsf{Pr}^{\pi}_{\mathcal{M},s}[\Diamond(T \cup R)] = 1$.

Intuitively, the assumption requires that for all strategies, the system will eventually reach a target or a goal state; in other words: It is not possible to avoid both target and sink infinitely long. Variants of this assumption are used in numerous settings: On MDP, it is similar to the contraction assumption, e.g. [6, Chp. 4]; in stochastic game theory (a two-player extension of MDP) it is called *stopping*, e.g. [13]; and, using terms from the theory of the stochastic shortest path problem, we require all strategies to be *proper*, see e.g. [7].

This assumption already is important in the finite setting: There, **Absorption** is equivalent to the absence of *end components*, which introduce multiple solutions of Equation (1). Then, a VI algorithm computing upper bounds can be "stuck" at a greater fixpoint than the value and thus does not converge [9, 19]. *Any* procedure using value iteration thus either needs to exclude such cases or detect and treat them. Aside from end components, which are the only issue in the finite setting, uncountable systems may feature other complex behaviour, such as Zeno-like approaching the target closer and closer without reaching it.

Unfortunately, even just detecting these problems already is difficult. For the mentioned, restricted setting of probabilistic programs, almost sure termination is Π^2_0-complete [33]. Yet, universal termination with goal set $T \cup R$ is exactly what we require for **Absorption**. So, already on a restricted setting (together with a *given* guess for R), we cannot decide whether the assumption holds, let alone treat the underlying problems. Thus, we decide to exclude this issue and delegate treatment to specialized approaches.

(4) As in Section 3, we simplify the definition of $S^{\Diamond}_{\mathcal{F}}$ slightly in order to avoid technical details.

Algorithm 2 The BRTDP algorithm for MDPs with general state- and action-spaces.

```
Input: ApproxBounds query with precision $\varepsilon$, satisfying A1–A6, B.BRTDP, C and D.
Output: $\varepsilon$-optimal values $(l, u)$.
1: $\mathsf{Sampled} \leftarrow \emptyset, \mathsf{t} \leftarrow 1$                                   ▷ Initialize
2: while $\textsc{Approx}_{\geq}\Big(\mathsf{U}(s_0), \textsc{Precision}(\mathsf{t})\Big) - \textsc{Approx}_{\leq}\Big(\mathsf{L}(s_0), \textsc{Precision}(\mathsf{t})\Big) \geq \varepsilon$ do
3:     $s, a \leftarrow \textsc{GetPair}$                                          ▷ Sample stat-action pair
4:     if $s \in T$ then   $\widehat{\mathsf{L}}(s, \cdot) \leftarrow 1$                          ▷ Handle special cases
5:     else if $s \in R$ then   $\widehat{\mathsf{U}}(s, \cdot) \leftarrow 0$
6:     else                                                ▷ Update upper and lower bounds
7:         $\widehat{\mathsf{U}}(s, a) \leftarrow \textsc{Approx}_{\geq}(\Delta(s, a)\langle\mathsf{U}\rangle, \textsc{Precision}(\mathsf{t}))$
8:         $\widehat{\mathsf{L}}(s, a) \leftarrow \textsc{Approx}_{\leq}(\Delta(s, a)\langle\mathsf{L}\rangle, \textsc{Precision}(\mathsf{t}))$
9:     $\mathsf{Sampled} \leftarrow \mathsf{Sampled} \cup \{(s, a)\}$, $\mathsf{t} \leftarrow \mathsf{t} + 1$
10: return $(\mathsf{L}(s_0), \mathsf{U}(s_0))$
```

In summary, while this assumption is indeed restrictive, it is the key point that allows us to obtain convergent upper bounds and thus an anytime algorithm. As argued above, an assumption of this kind seems to be *necessary* to obtain such an algorithm in this generality.

▶ Remark 6. These problems do not occur when considering *finite horizon* or *discounted* properties, which are frequently used in practice. For details on treating finite horizon objectives, see [17, App. C.1]. Discounted reachability with a factor of $\gamma < 1$ is equivalent to normal reachability where at each step the system moves into a sink state with probability $(1 - \gamma)$. **Absorption** is trivially satisfied and our methods are directly applicable.

4.2 Assumptions Applied: The Convergent Anytime Algorithm

With our assumptions in place, we are ready to present our adaptation of BRTDP to the uncountable setting. Compared to VI, we now also store upper bounds, again using Lipschitz-continuity to extrapolate the stored values. In particular, together with the definitions of Equation (3) we additionally set

$$\mathsf{U}(s, a) = \min_{(s', a') \in \mathsf{Sampled}} \left(\widehat{\mathsf{U}}(s', a') + \mathrm{d}_{\times}((s, a), (s', a')) \cdot C_{\times}\right).$$

We also set $\mathsf{U}(s, a) = 1$ if either $\mathsf{Sampled} = \emptyset$ or the above equation would yield $\mathsf{U}(s, a) > 1$.

We present BRTDP in Algorithm 2. It is structurally similar to BRTDP in the finite setting (see [17, App. A.2]). The major difference is given by the storage tables $\widehat{\mathsf{U}}$ and $\widehat{\mathsf{L}}$ used to compute the current bounds U and L, again exploiting Lipschitz continuity. As before, the central idea is to repeatedly update state-action pairs given $\textsc{GetPair}$. If $\textsc{GetPair}$ yields a state of the terminal sets T and R, we update the stored values directly. Otherwise, we back-propagate the value of the selected pair by computing the expected value under this transition. Moreover, we again require that $\textsc{Precision}(\mathsf{t})$ converges to zero. Note that the algorithm can easily be supplied with a-priori knowledge by initializing the upper and lower bounds to non-trivial values. Moreover, in contrast to VI, this algorithm is an *anytime* algorithm, i.e. it can at any time provide an approximate solution together with its precision.

Despite the algorithm being structurally similar to the finite variant of [9], the proof of correctness unsurprisingly is more intricate due to the uncountable sets. We again provide both a simplified proof sketch here and the full technical proof in [17, App. E.2].

▶ **Theorem 7.** *Algorithm 2 is correct under Assumptions **A1**–**A6**, **B.BRTDP**, **C** and **D**, and terminates with probability 1.*

Proof sketch. We again obtain monotonicity of the bounds, i.e. $\mathsf{L}_\mathsf{t}(s,a) \leq \mathsf{L}_{\mathsf{t}+1}(s,a) \leq \mathcal{V}(s,a) \leq \mathsf{U}_{\mathsf{t}+1}(s,a) \leq \mathsf{U}_\mathsf{t}(s,a)$ by induction on t, using completely analogous arguments.

By monotonicity, we also obtain well defined limits U_∞ and L_∞. Further, we define the difference function $\mathrm{Diff}_\mathsf{t}(s,a) = \mathsf{U}_\mathsf{t}(s,a) - \mathsf{L}_\mathsf{t}(s,a)$ together with its state based counterpart $\mathrm{Diff}_\mathsf{t}(s)$ and its limit $\mathrm{Diff}_\infty(s)$. We show that $\mathrm{Diff}_\infty(s_0) = 0$, proving convergence. To this end, similar to the previous proof, we prove that Diff_∞ satisfies a fixed point equation on $S^\Diamond_+$ (see **B.BRTDP**), namely $\mathrm{Diff}_\infty(s) = \Delta(s,a(s))\langle\mathrm{Diff}_\infty\rangle$ where $a(s)$ is a specially chosen "optimal" action for each state satisfying $\mathrm{Diff}_\infty(s,a(s)) = \mathrm{Diff}_\infty(s)$. Now, set $\mathrm{Diff}_* = \max_{s\in S^\Diamond_+} \mathrm{Diff}_\infty(s)$ the maximal difference on $S^\Diamond_+$ and let $S^\Diamond_*$ be the set of witnesses obtaining Diff_*. Then, $\Delta(s,a(s),S^\Diamond_*) = 1$: If a part of the transition's probability mass would move to a region with smaller difference, an appropriate update of a pair close to $(s,a(s))$ would reduce its difference. Hence, the set of states $S^\Diamond_*$ is a "stable" subset of the system when following the actions $a(s)$. By **Absorption**, we eventually have to reach either the target T or the sink R starting from any state in $S^\Diamond_*$. Since $\mathrm{Diff}_\infty(s) = 0$ for all (sampled) states in $T \cup R$ and Diff_∞ satisfies the fixed point equation, we get that $\mathrm{Diff}_\infty(s) = 0$ for all states $S^\Diamond_*$ and consequently $\mathrm{Diff}_\infty(s_0) = 0$. ◀

5 Discussion

5.1 Relation to Algorithms for Finite Systems and Discretization

Our algorithm directly generalizes the classical value iteration as well as BRTDP for finite MDP by an appropriate choice of GetPair. In value iteration, it proceeds in round-robin fashion, enumerating all state-action pairs. Note that the algorithm immediately uses the results of previous updates, corresponding to the *Gauß-Seidel variant* of VI; to exactly obtain synchronous value iteration, we would have to slightly modify the structure for saving the values. In BRTDP, GetPair simulates paths through the MDP and we update only those states encountered during the simulation.

Approaches based on discretization through, e.g., grids with increasing precision, essentially reduce the uncountable state space to a finite one. This is also encompassed by GetPair, e.g. by selecting the grid points in round robin or randomized fashion. However, our algorithm has the following key advantages when compared to classical discretization. Firstly, it avoids the need to grid the whole state space (typically into cells of regular sizes). Secondly, in discretization, updating the value of one cell does not directly affect the value in other cells; in contrast in our algorithm, knowledge about a state fluently propagates to other areas (by using Equation (3)) without being hindered by (arbitrarily chosen) cell boundaries.

5.2 Extensions

We outline possible extensions and augmentations of our approach to showcase its versatility.

Discontinuities. Our Lipschitz assumption **C** actually is slightly stronger than required. We first give an example of a system exhibiting discontinuities and then describe how our approach can be modified to deal with it. More details are in [17, App. C.2].

▶ **Example 8.** Consider a robot navigating a terrain with cliffs, where falling down a cliff immediately makes it impossible to reach the target. There, states which are barely on the edge may still reach the goal with significant probability, while a small step to the side results in falling down the cliff and zero probability of reaching the goal.

To solve this example, one could model the cliff as a steep but continuous slope, which would make our approach still possible. Unfortunately, this might not be very practical, since the Lipschitz constant then is quite large.

However, if we know of discontinuities, e.g. the location of cliffs in the terrain the robot navigates, both our algorithms can be extended as follows: Instead of requiring $\mathcal{V}$ to be continuous on the whole domain, we may assume that we are given a (finite, decidable) partitioning of the state set S into several sets S_i. We allow the value function to be discontinuous along the boundaries of S_i (the cliffs), as long as it remains Lipschitz-continuous inside each S_i. We only need to slightly modify the assumption on GetPair by requiring that for any state-action pair (s, a) with $s \in S_i$ we eventually get a nearby, similarly behaving state-action pair (s', a') *of the same region*, i.e. $s' \in S_i$. While computing the bounds of a particular state-action pair, e.g. $\mathsf{U}(s, a)$, we first determine which partition S_i the state s belongs to and then only consider the stored values of states inside the region S_i.

Linear Temporal Logic. In [9], the authors extend BRTDP to LTL queries [42]. Several difficulties arise in the uncountable setting. For example, in order to prove *liveness* conditions, we need to solve the *repeated reachability* problem, i.e. whether a particular set of states is reached infinitely often. This is difficult even for restricted classes of uncountable systems, and impossible in the general case. In particular, [9] relies on analysing end components, which we already identified as an unresolved problem. We provide further insight in [17, App. C.3]. Nevertheless, there is a straightforward extension of our approach to the subclass of *reach-avoid* problems [50] (or *constrained reachability* [52]), see [17, App. C.4].

5.3 Implementation and Heuristics

For completeness, we implemented a prototype of our BRTDP algorithm to demonstrate its effectiveness. See [17, App. D] for details and an evaluation on both a one- and two-dimensional navigation model. Our implementation is barely optimized, with no delegation to high-performance libraries. Yet, these non-trivial models are solved in reasonable time. However, since we aim for assumptions that are as general as possible, one cannot expect our generic approach perform on par with highly optimized tools. Our prototype serves as a proof-of-concept and does not aim to be competitive with specialized approaches. We highlight again that the goal of our paper is *not* to be practically efficient in a particular, restricted setting, but rather to provide general assumptions and theoretical algorithms applicable to all kinds of uncountable systems.

Aside from several possible optimizations concerning the concrete implementation, we suggest two more general directions for heuristics:

Adaptive Lipschitz constants. As an example, suppose that a robot is navigating mostly flat land close to its home, but more hilly terrain further away. The flat land has a smaller Lipschitz constant than the hilly terrain, and thus here we can infer tighter bounds. More generally, given a partitioning of the state space and local Lipschitz constants for every subset, we use this local knowledge when computing $\widehat{\mathsf{L}}$ and $\widehat{\mathsf{U}}$ instead of using the global Lipschitz constant, which is the maximum of all local ones. See [17, App. C.2] for details.

GetPair-heuristics. In Section 3.1.2, we mentioned two simple implementations of GetPair. Firstly, we can discretize both state and action space, yielding each state-action pair in the discretization for a finite number of iterations, choosing a finer discretization constant, and repeating the process until convergence. Assuming that we can sample all state-action pairs in the discretization, this method eventually samples arbitrarily close to *any* state-action pair in $S \times Av$ and thus trivially satisfies the sampling assumption. This intuitively corresponds to executing interval iteration [19] on the (increasingly refined) discretized systems. Note that this approach completely disregards the reachability probability of certain states and invests the same computational effort for all of them. In particular, it invests the same amount of computational effort into regions which are only reached with probability 10^{-100} as in regions around the initial state s_0.

Thus, a second approach is to sample a path through the system at random, following random actions. This approach updates states roughly proportional to the probability of being reached, which already in the finite setting yields dramatic speed-ups [34].

However, we can also use further information provided by the algorithm, namely the upper bounds. As mentioned in [9], following "promising" actions with a large upper bound proves to be beneficial, since actions with small upper bound likely are suboptimal. To extend this idea to the general domain, we need to apply a bit of care. In particular, it might be difficult to select exactly from the optimal set of actions, since already $\arg\max_{a \in Av(s)} \mathsf{U}(s, a)$ might be very difficult to compute. Yet, it is sufficient to choose some constant $\xi > 0$ and over-approximate the set of ξ-optimal actions in a given state, randomly selecting from this set. This over-approximation can easily be performed by, for example, randomly sampling the set of available actions $Av(s)$ until we encounter an action close to the optimum (which can approximate due to our assumptions). By generating paths only using these actions, we combine the previous idea of focussing on "important" states (in terms of reachability) with an additional focus on "promising" states (in terms of upper bounds). This way, the algorithm *learns* from its experiences, using it as a guidance for future explorations.

More generally, we can easily apply more sophisticated learning approaches by interleaving it with one of the above methods. For example, by following the learning approach with probability ν and a "safe" method with probability $1 - \nu$ we still obtain a safe heuristic, since the assumption only requires limit behaviour. As such, we can combine our approach with existing, learning based algorithm by following their suggested heuristic and interleave it with some sampling runs guided by the above ideas. In other words, this means that the learning algorithm can focus on finding a reasonable solution quickly, which is then subsequently *verified* by our approach, potentially *improving* the solution in areas where the learner is performing suboptimally. On top, the (guaranteed) bounds identified by our algorithm can be used as feedback to the learning algorithm, creating a positive feedback loop, where both components improve each other's behaviour and performance.

6 Conclusion

In this work, we have presented the first anytime algorithm to tackle the reachability problem for MDP with uncountable state- and action-spaces, giving both correctness and termination guarantees under general assumptions. The experimental evaluation of our prototype implementation shows both promising results and room for improvements.

On the theoretical side, we conjecture that **Assumption D: Absorption** can be weakened if we complement it with an automatic procedure that finds and treats problematic parts of the state space of a certain kind, similar to the collapsing approach on finite MDP

[19, 9]. Note that as the general problem is undecidable, some form of **Absorption** will remain necessary. On the practical side, we aim for a more sophisticated tool, applying our theoretical foundation to the full range of MDP, including discrete discontinuities. Moreover, we want to combine the tool with existing ways of identifying the Lipschitz constant.

References

1 Alessandro Abate, Saurabh Amin, Maria Prandini, John Lygeros, and Shankar Sastry. Computational approaches to reachability analysis of stochastic hybrid systems. In *HSCC*, volume 4416 of *Lecture Notes in Computer Science*, pages 4–17. Springer, 2007. doi:10.1007/978-3-540-71493-4_4.

2 Alessandro Abate, Joost-Pieter Katoen, John Lygeros, and Maria Prandini. Approximate model checking of stochastic hybrid systems. *Eur. J. Control*, 16(6):624–641, 2010. doi:10.3166/ejc.16.624-641.

3 Alessandro Abate, Maria Prandini, John Lygeros, and Shankar Sastry. Probabilistic reachability and safety for controlled discrete time stochastic hybrid systems. *Automatica*, 44(11):2724–2734, 2008. doi:10.1016/j.automatica.2008.03.027.

4 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 *CAV (1)*, volume 10426 of *Lecture Notes in Computer Science*, pages 160–180. Springer, 2017.

5 Dimitri Bertsekas. Convergence of discretization procedures in dynamic programming. *IEEE Transactions on Automatic Control*, 20(3):415–419, 1975.

6 Dimitri P Bertsekas and Steven Shreve. Stochastic optimal control: the discrete-time case, 1978.

7 Dimitri P. Bertsekas and John N. Tsitsiklis. An analysis of stochastic shortest path problems. *Math. Oper. Res.*, 16(3):580–595, 1991. doi:10.1287/moor.16.3.580.

8 Patrick Billingsley. *Probability and Measure*, volume 939. John Wiley & Sons, 2012.

9 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 *ATVA*, volume 8837 of *Lecture Notes in Computer Science*, pages 98–114. Springer, 2014. doi:10.1007/978-3-319-11936-6_8.

10 John L. Bresina, Richard Dearden, Nicolas Meuleau, Sailesh Ramakrishnan, David E. Smith, and Richard Washington. Planning under continuous time and resource uncertainty: A challenge for AI. *CoRR*, abs/1301.0559, 2013. arXiv:1301.0559.

11 Debasish Chatterjee, Eugenio Cinquemani, and John Lygeros. Maximizing the probability of attaining a target prior to extinction. *Nonlinear Analysis: Hybrid Systems*, 5(2):367–381, 2011.

12 Krishnendu Chatterjee, Zuzana Kretínská, and Jan Kretínský. Unifying two views on multiple mean-payoff objectives in Markov decision processes. *Logical Methods in Computer Science*, 13(2), 2017. doi:10.23638/LMCS-13(2:15)2017.

13 Anne Condon. The complexity of stochastic games. *Inf. Comput.*, 96(2):203–224, 1992. doi:10.1016/0890-5401(92)90048-K.

14 Zhengzhu Feng, Richard Dearden, Nicolas Meuleau, and Richard Washington. Dynamic programming for structured continuous Markov decision problems. In *UAI*, pages 154–161. AUAI Press, 2004. URL: https://dslpitt.org/uai/displayArticleDetails.jsp?mmnu=1&smnu=2&article_id=1102&proceeding_id=20.

15 Vojtech Forejt, Marta Z. Kwiatkowska, Gethin Norman, and David Parker. Automated verification techniques for probabilistic systems. In *SFM*, volume 6659 of *Lecture Notes in Computer Science*, pages 53–113. Springer, 2011. doi:10.1007/978-3-642-21455-4_3.

16 Hongfei Fu and Krishnendu Chatterjee. Termination of nondeterministic probabilistic programs. In *VMCAI*, volume 11388 of *Lecture Notes in Computer Science*, pages 468–490. Springer, 2019. doi:10.1007/978-3-030-11245-5_22.

17 Kush Grover, Jan Kretínský, Tobias Meggendorfer, and Maximilian Weininger. Anytime guarantees for reachability in uncountable markov decision processes. *CoRR*, abs/2008.04824, 2020. arXiv:2008.04824.

18 Carlos Guestrin, Milos Hauskrecht, and Branislav Kveton. Solving factored MDPs with continuous and discrete variables. In *UAI*, pages 235–242. AUAI Press, 2004. URL: https://dslpitt.org/uai/displayArticleDetails.jsp?mmnu=1&smnu=2&article_id=1113&proceeding_id=20.

19 Serge Haddad and Benjamin Monmege. Interval iteration algorithm for MDPs and IMDPs. *Theor. Comput. Sci.*, 735:111–131, 2018. doi:10.1016/j.tcs.2016.12.003.

20 Sofie Haesaert, Sadegh Soudjani, and Alessandro Abate. Temporal logic control of general Markov decision processes by approximate policy refinement. In *ADHS*, volume 51(16) of *IFAC-PapersOnLine*, pages 73–78. Elsevier, 2018. doi:10.1016/j.ifacol.2018.08.013.

21 Sofie Haesaert, Sadegh Esmaeil Zadeh Soudjani, and Alessandro Abate. Verification of general Markov decision processes by approximate similarity relations and policy refinement. *SIAM J. Control and Optimization*, 55(4):2333–2367, 2017. doi:10.1137/16M1079397.

22 Arnd Hartmanns and Benjamin Lucien Kaminski. Optimistic value iteration. In *CAV (2)*, volume 12225 of *Lecture Notes in Computer Science*, pages 488–511. Springer, 2020.

23 Mohammadhosein Hasanbeig, Alessandro Abate, and Daniel Kroening. Logically-constrained neural fitted q-iteration. In *AAMAS*, pages 2012–2014. International Foundation for Autonomous Agents and Multiagent Systems, 2019. URL: http://dl.acm.org/citation.cfm?id=3331994.

24 Mohammadhosein Hasanbeig, Alessandro Abate, and Daniel Kroening. Certified reinforcement learning with logic guidance. *CoRR*, abs/1902.00778, 2019. arXiv:1902.00778.

25 William B. Haskell, Rahul Jain, Hiteshi Sharma, and Pengqian Yu. A universal empirical dynamic programming algorithm for continuous state MDPs. *IEEE Trans. Automat. Contr.*, 65(1):115–129, 2020. doi:10.1109/TAC.2019.2907414.

26 Thomas A. Henzinger, Peter W. Kopke, Anuj Puri, and Pravin Varaiya. What's decidable about hybrid automata? In *STOC*, pages 373–382. ACM, 1995.

27 Thomas A. Henzinger, Peter W. Kopke, Anuj Puri, and Pravin Varaiya. What's decidable about hybrid automata? *J. Comput. Syst. Sci.*, 57(1):94–124, 1998. doi:10.1006/jcss.1998.1581.

28 Onésimo Hernández-Lerma and Jean B Lasserre. *Discrete-time Markov control processes: basic optimality criteria*, volume 30. Springer Science & Business Media, 2012.

29 Ronald A Howard. *Dynamic programming and Markov processes.* John Wiley, 1960.

30 Mingzhang Huang, Hongfei Fu, and Krishnendu Chatterjee. New approaches for almost-sure termination of probabilistic programs. In *Program. Lang. and Sys.*, volume 11275 of *Lecture Notes in Computer Science*, pages 181–201. Springer, 2018. doi:10.1007/978-3-030-02768-1_11.

31 Mingzhang Huang, Hongfei Fu, Krishnendu Chatterjee, and Amir Kafshdar Goharshady. Modular verification for almost-sure termination of probabilistic programs. *Proc. ACM Program. Lang.*, 3(OOPSLA):129:1–129:29, 2019. doi:10.1145/3360555.

32 Manfred Jaeger, Peter Gjøl Jensen, Kim Guldstrand Larsen, Axel Legay, Sean Sedwards, and Jakob Haahr Taankvist. Teaching stratego to play ball: Optimal synthesis for continuous space MDPs. In *ATVA*, volume 11781 of *Lecture Notes in Computer Science*, pages 81–97. Springer, 2019. doi:10.1007/978-3-030-31784-3_5.

33 Benjamin Lucien Kaminski and Joost-Pieter Katoen. On the hardness of almost-sure termination. In *MFCS*, volume 9234 of *Lecture Notes in Computer Science*, pages 307–318. Springer, 2015. doi:10.1007/978-3-662-48057-1_24.

34 Jan Kretínský and Tobias Meggendorfer. Of cores: A partial-exploration framework for Markov decision processes. In *CONCUR*, volume 140 of *LIPIcs*, pages 5:1–5:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.CONCUR.2019.5.

35 Ratan Lal and Pavithra Prabhakar. Bounded verification of reachability of probabilistic hybrid systems. In *QEST*, volume 11024 of *Lecture Notes in Computer Science*, pages 240–256. Springer, 2018. `doi:10.1007/978-3-319-99154-2_15`.

36 Bernard F Lamond and Abdeslem Boukhtouta. Water reservoir applications of Markov decision processes. In *Handbook of Markov decision processes*, pages 537–558. Springer, 2002.

37 Lihong Li and Michael L. Littman. Lazy approximation for solving continuous finite-horizon MDPs. In *AAAI*, pages 1175–1180. AAAI Press / The MIT Press, 2005. URL: `http://www.aaai.org/Library/AAAI/2005/aaai05-186.php`.

38 Masoud Mahootchi. *Storage system management using reinforcement learning techniques and nonlinear models*. PhD thesis, University of Waterloo, 2009.

39 H. Brendan McMahan, Maxim Likhachev, and Geoffrey J. Gordon. Bounded real-time dynamic programming: RTDP with monotone upper bounds and performance guarantees. In *ICML*, volume 119 of *ACM International Conference Proceeding Series*, pages 569–576. ACM, 2005. `doi:10.1145/1102351.1102423`.

40 Francisco S. Melo, Sean P. Meyn, and M. Isabel Ribeiro. An analysis of reinforcement learning with function approximation. In *ICML*, volume 307 of *ACM International Conference Proceeding Series*, pages 664–671. ACM, 2008. `doi:10.1145/1390156.1390240`.

41 Goran Peskir and Albert Shiryaev. *Optimal stopping and free-boundary problems*. Springer, 2006.

42 Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 46–57. IEEE Computer Society, 1977. `doi:10.1109/SFCS.1977.32`.

43 Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley Series in Probability and Statistics. Wiley, 1994. `doi:10.1002/9780470316887`.

44 Tim Quatmann and Joost-Pieter Katoen. Sound value iteration. In *CAV (1)*, volume 10981 of *Lecture Notes in Computer Science*, pages 643–661. Springer, 2018.

45 Scott Sanner, Karina Valdivia Delgado, and Leliane Nunes de Barros. Symbolic dynamic programming for discrete and continuous state MDPs. In *UAI*, pages 643–652. AUAI Press, 2011. URL: `https://dslpitt.org/uai/displayArticleDetails.jsp?mmnu=1&smnu=2&article_id=2223&proceeding_id=27`.

46 Claude Elwood Shannon. Communication in the presence of noise. *Proceedings of the IRE*, 37(1):10–21, 1949.

47 Hiteshi Sharma, Mehdi Jafarnia-Jahromi, and Rahul Jain. Approximate relative value learning for average-reward continuous state MDPs. In *UAI*, page 341. AUAI Press, 2019. URL: `http://auai.org/uai2019/proceedings/papers/341.pdf`.

48 Fedor Shmarov and Paolo Zuliani. Probreach: verified probabilistic delta-reachability for stochastic hybrid systems. In *HSCC*, pages 134–139. ACM, 2015.

49 Sadegh Esmaeil Zadeh Soudjani and Alessandro Abate. Adaptive gridding for abstraction and verification of stochastic hybrid systems. In *QEST*, pages 59–68. IEEE Computer Society, 2011. `doi:10.1109/QEST.2011.16`.

50 Sean Summers and John Lygeros. Verification of discrete time stochastic hybrid systems: A stochastic reach-avoid decision problem. *Automatica*, 46(12):1951–1961, 2010. `doi:10.1016/j.automatica.2010.08.006`.

51 Ilya Tkachev, Alexandru Mereacre, Joost-Pieter Katoen, and Alessandro Abate. Quantitative automata-based controller synthesis for non-autonomous stochastic hybrid systems. In *HSCC*, pages 293–302. ACM, 2013. `doi:10.1145/2461328.2461373`.

52 Ilya Tkachev, Alexandru Mereacre, Joost-Pieter Katoen, and Alessandro Abate. Quantitative model-checking of controlled discrete-time Markov processes. *Inf. Comput.*, 253:1–35, 2017. `doi:10.1016/j.ic.2016.11.006`.

53 Hado van Hasselt. Reinforcement learning in continuous state and action spaces. In *Reinforcement Learning*, volume 12 of *Adaptation, Learning, and Optimization*, pages 207–251. Springer, 2012. `doi:10.1007/978-3-642-27645-3_7`.

54 Luis Gustavo Rocha Vianna, Scott Sanner, and Leliane Nunes de Barros. Continuous real time dynamic programming for discrete and continuous state MDPs. In *2014 Brazilian Conference on Intelligent Systems, BRACIS 2014, Sao Paulo, Brazil, October 18-22, 2014*, pages 134–139. IEEE Computer Society, 2014. `doi:10.1109/BRACIS.2014.34`.

B Semantic Abstraction-Guided Motion Planning for scLTL Missions in Unknown Environments

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

Kush Grover, Fernando S Barbosa, Jana Tumova, and Jan Křetínský. "Semantic Abstraction-Guided Motion Planning for scLTL Missions in Unknown Environments". In: *Robotics: Science and Systems XVII, Virtual Event, July 12-16, 2021*. RSS Foundation-Robotics Science & Systems Foundation. Virtual, July 2021. DOI: 10.15607/RSS.2021.XVII.090

Summary. This paper deals with motion planning in unknown environments for temporal specifications. Unknown environment means that the robot could only perceive obstacle and labeling within a certain radius around it. We gave an algorithm which learns the semantic relations present in the environment to figure out similar transitions it should look for, in the future, to satisfy the formula as soon as possible. Additionally, to maintain record of explored and unexplored area we employed a frontier-based approach which also suggests directions with most unexplored area to guide the robot's movements. We combined the two biases and incorporated them in an RRG style algorithm.

Contributions of thesis author. The author played a pivotal role in the composition and revision of the manuscript. He actively participated in joint discussions and was a major contributor to the development of the results presented in the paper. Noteworthy individual contributions include designing of the algorithm along with its complete implementation. The experiments presented were also done by the him.

Robotics: Science and Systems 2021
Held Virtually, July 12–16, 2021

Semantic Abstraction-Guided Motion Planning for scLTL Missions in Unknown Environments

Kush Grover*, Fernando S. Barbosa†, Jana Tumova† and Jan Křetínský*
*Technical University of Munich, Germany. Emails: {kush.grover, jan.kretinsky}@tum.de
†KTH Royal Institute of Technology, Stockholm, Sweden. Emails: {fdsb, tumova}@kth.se

***Abstract*—Complex mission specifications can be often specified through temporal logics, such as Linear Temporal Logic and its syntactically co-safe fragment, scLTL. Finding trajectories that satisfy such specifications becomes hard if the robot is to fulfil the mission in an initially unknown environment, where neither locations of regions or objects of interest in the environment nor the obstacle space are known a priori. We propose an algorithm that, while exploring the environment, learns important semantic dependencies in the form of a semantic abstraction, and uses it to bias the growth of an Rapidly-exploring random graph towards faster mission completion. Our approach leads to finding trajectories that are much shorter than those found by the sequential approach, which first explores and then plans. Simulations comparing our solution to the sequential approach, carried out in 100 randomized office-like environments, show more than 50% reduction in the trajectory length.**

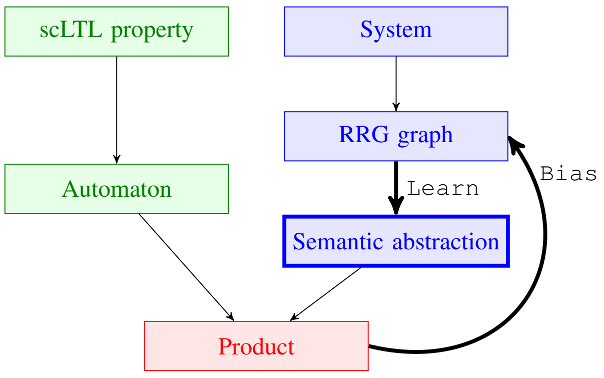


Figure 1. Scheme of our model-checking-inspired approach with novel elements drawn thickly.

I. Introduction

Motion planning with Linear Temporal Logic (LTL) mission specifications aims for consideration of richer objectives than the traditional A-to-B motion planning. Examples of such objectives include periodic surveillance, request-response, or sequencing. Successful approaches to the problem range from using various cell decomposition techniques, to creating roadmaps abstracting the environment and to sampling-based motion planning. Motion planning with LTL missions is, however, much more challenging in *a priori* unknown environments: efficient treatment of LTL specifications may require exploiting semantic and spatio-temporal dependencies between features of the environment, which are typically unknown beforehand. As an example, consider that we would like a robot to check all waste bins in all offices in an office environment. When finding the first bin, the robot may realize it was next to a desk. While looking for the bin in the next office, it is most natural that the robot starts exploring again next to the desk. At the same time, due to the potential complexity of the environment, it is not desirable to stick fully to all of the observed semantic and spatio-temporal correlations as not all of them are relevant for the specification satisfaction.

In this paper, we focus on sampling-based motion planning with missions specified with the syntactically co-safe fragment of LTL (scLTL), and with the robot deployed in *a priori* unknown environments. The key idea of our approach is, on the conceptual level, to make the sampling *guided* by a semantic abstraction of the system and by the specification. The overview of our algorithm is depicted in Fig. 1. We extend the Rapidly-exploring Random Graphs (RRG) algorithm with learning and biasing; we iteratively learn a semantic abstraction of the system from the gradually growing RRG graph and compose it with an automaton representation of the specification into a so-called product. The product is used to bias sampling in RRG, i.e. to exploit the semantic and spatio-temporal dependencies of features in the environment as well as their relation to satisfying the desired specification.

Compared to the naive two-step approach, which first explores the environment and then plans a trajectory that satisfies the mission, our approach (i) performs both tasks at once and, moreover, (ii) allows mutual exchange of information between the two tasks. We show that these two improvements shorten the length of the executed path significantly. We achieve this while maintaining similar computation time, which will, in reality, be negligible as the robot can execute the algorithm in real-time while navigating in the environment. Our contribution can be summarized as follows:

- We propose a method to learn a semantic abstraction of the system, suitable for planning with scLTL missions.
- We exploit the learned semantic abstraction and, together with consideration of the specification, we bias the growth of the RRG graph towards promising regions (in terms of making progress towards the specification satisfaction).
- We experimentally show that the loop between sampling and learning leads to better planning in terms of shorter trajectories when compared to the naive two-step approach. The results indicate more than 50% savings.

The paper is organized as follows. Sec. I-A introduces relevant related work, and Sec. II describes preliminary tools needed for the remainder of the paper. The problem is formally defined in Sec. III, which is followed by the proposed solution and analysis in Sec. IV. Lastly, a case study is presented in Sec. V, with conclusions and future work in Sec. VI.

A. *Related Work*

One of the first works to propose the use of a sampling-based motion planning algorithm to find a trajectory that satisfies a temporal logic specification is [9]. In that work, the authors propose the Rapidly-exploring Random Graph (RRG) as an alternative to the Rapidly-exploring Random Tree (RRT) to finding cyclic trajectories that satisfy a deterministic μ-calculus specification. Another approach is presented in [4], but this time for the syntactically co-safe fragment of Linear Temporal Logic (scLTL). Following these, Vasile and Belta [17] propose improvements to [9], more specifically for dealing with full LTL and for improving scalability. None of these works, however, deals with partially-known environments, nor do they attempt to speed up the search by learning characteristics of the environment.

More recently, Kantaros and Zavlanos [6] described an approach for multi-robot systems under global temporal tasks. Instead of using an RRG, the authors propose a two-step approach using RRT*. The first step constructs a tree until an accepting state of the automaton capturing the evolution of the LTL formula is reached. The second step then grows another tree rooted at this accepting state, and attempts to find a cyclic (infinite) path that satisfies the LTL specification. The same authors then introduce in [7] sampling bias guided by the automaton capturing the LTL, something that [13] also proposes in a similar fashion. Lastly, besides proposing a heuristic to guide the search, [16] integrates feedback control laws to guarantee feasibility of plans by robots with complex, possibly non-holonomic, dynamics. Although these works propose ways of improving the time taken to find a plan, they all rely on having details of the environment a priori.

To the best of our knowledge, the two papers that are mostly related to ours are [8] and [1]. The former proposes a reactive sampling-based algorithm for path planning in unknown environments under scLTL specifications. However, differently from what we propose, only the obstacle space is initially unknown to [8], i.e. the locations of the regions of interest, therefore the labeling function, are known a priori. On the other hand, Ayala et al. [1] considers completely unknown environments, including the labeling function. However, the authors propose an approach over a discretized partitioning of the environment, performing frontier exploration [18] until a path that satisfies the scLTL specification is found. We merge benefits of both approaches by proposing a sampling-based approach on completely unknown environments; furthermore, we propose a way of learning relations between labels, together with exploiting them for guiding the path search.

When it comes to robotic deployment in unknown environments, a crucial initial step might be to efficiently create a map in an exploratory manner. A seminal work on exploration is by Yamauchi [18], in which the author proposes the method coined *frontier exploration*. Since then, several other approaches have been proposed. Among them is the Receding Horizon Next-Best-View Planner [5] and the Autonomous Exploration Planner [15], both building upon RRT*. These works, however, do not focus on capturing various dependencies and relations in the environment. In contrast, in a probabilistic approach proposed by Aydemir et al. [2], a robot uses common-sense knowledge about the relation between objects and semantic room categories. Here, the focus is however on search for objects and not satisfaction of complex LTL goals.

II. Preliminaries

Let $\mathbb{R}$ denote the set of real numbers and $\mathbb{R}^n$ the n-dimensional Euclidean space. We use Σ for the finite set of atomic propositions. For a set X, 2^X denotes its power set. A word over an alphabet Y is a sequence of elements of Y. The exclusive-or operation is denoted by $\oplus$, and the disjoint union of sets by $\uplus$.

Consider a robot deployed in an *environment* $\mathcal{X} \subset \mathbb{R}^n$ and let $x_0 \in \mathcal{X}$ be its initial state. Let $\{\mathcal{O}_1, \mathcal{O}_2, \dots \mathcal{O}_k\}$ be the set of obstacles such that $\mathcal{O}_i \subset \mathcal{X}$ for all $i \in [1, k]$, and $\mathcal{X}_{\text{free}} = \mathcal{X} \setminus \bigcup_{i=1}^{k} \mathcal{O}_i$ denotes the obstacle-free space. A trajectory in the environment $\mathcal{X}$ is defined by a continuous function $\sigma : [0,1] \to \mathcal{X}$. A trajectory is collision-free if $\sigma(t) \in \mathcal{X}_{\text{free}}, \forall t \in [0,1]$. Regions of the environment $\mathcal{X}$ are labelled with atomic propositions Σ according to a labelling function $L : \mathcal{X} \to 2^{\Sigma}$, which maps each state in the state-space to a set of atomic propositions that hold true there.

A map of the environment $\mathcal{X}$ is a partitioning into a finite number of cells of equal size with a predefined precision, which can be labeled as *free*, meaning that the cell lies in $\mathcal{X}_{\text{free}}$, *occupied*, if any point within the cell lies inside the obstacle space (corresponding to an over-approximation of the obstacle set), or *unmapped*, that highlights the cell has not been seen by the robot so far. Every cell is initialised as *unmapped*, and is updated whenever it lies in the line-of-sight of the robot. A cell is called a *frontier* cell if it is marked as *free* and has a neighbouring cell marked as *unmapped*. A *map frontier* is a connected group of frontier cells, and its size is its cardinality. This is a common approach among the 3D-exploration community, so we refer to papers such as [18, 5] for more details.

A. *RRG*

The Rapidly-exploring Random Graph [10] is an anytime[1] sampling-based motion planning algorithm that builds a connected roadmap. It incrementally builds a graph $G = (V, E)$ such that $v \in \mathcal{X}_{\text{free}}, \forall v \in V$, and an edge $e \in E$ connects two nodes $v_a, v_b \in V$ if there exists a collision-free trajectory $\sigma_{v_a}^{v_b}$ between them, with $\sigma_{v_a}^{v_b}(0) = v_a$ and $\sigma_{v_a}^{v_b}(1) = v_b$. A path

[1]An anytime algorithm returns a valid solution even if it is interrupted before termination; moreover, the longer it runs, the more its solution is improved.

over G is a sequence of nodes $p = v_0, v_1, v_2, \ldots$ such that $v_i \in V$ and $(v_i, v_{i+1}) \in E$, for all $i \geq 0$.

B. Syntactically co-safe LTL and DFA

Definition 1 ((Syntactically co-safe) Linear Temporal Logic [14, 11])**.** *A formula of* LTL *is given by the syntax:*

$$\varphi := a \mid \neg a \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \mathbf{X}\varphi \mid \varphi_1 \mathbf{U} \varphi_2 \mid \mathbf{G}\varphi$$

where, $a \in \Sigma$ is an atomic proposition, $\neg, \wedge, \vee$ are the Boolean operators 'negation', 'conjunction', and 'disjunction', respectively. **X**, **U**, **G** *denote the LTL operators 'next', 'until', and 'globally' respectively. The* syntactically co-safe fragment of LTL (scLTL) *is given by the same syntax, but prohibiting the operator* **G**.

The semantics of LTL formulas is defined on words over 2^{Σ}. The Boolean operators have usual semantics. Intuitively, $\mathbf{X}\varphi$ means that φ is true in the next time step and $\varphi_1 \mathbf{U} \varphi_2$ asserts that φ_1 will be true until φ_2 becomes true. **F** is known as the 'finally' or 'eventually' operator whose semantics asserts that the property φ becomes true at some point in the future. As such, it can be defined in terms of **U** as **F** $\varphi \equiv$ *true* **U** φ **G** is known as the 'globally' or 'always' operator with the semantics that φ is always satisfied. Since the robot moves in continuous time and **X** operator is usually defined for discrete time steps, we consider for simplicity properties without **X**. However, our approach is applicable for the whole of LTL.

Let $\mathcal{L}(\varphi)$ denote the set of words that satisfies the LTL formula φ.

Definition 2 (Deterministic Finite Automaton)**.** *A* deterministic finite automaton (DFA) *is a tuple $(2^{\Sigma}, Q, q_0, \delta, F)$ where 2^{Σ} is the alphabet, Q is a finite set of states, q_0 is an initial state, $\delta : Q \times 2^{\Sigma} \to Q$ is a transition function and $F \subseteq Q$ is the set of accepting states.*

A run over a word $w_1, \ldots, w_n$ is a sequence of states $q_0, q_1, \ldots, q_n$ such that $q_i = \delta(q_{i-1}, w_i)$ for all i. A word is accepted by the automaton if the run over the word end in F. We define the language accepted by an DFA $\mathcal{A}$ as $\mathcal{L}(\mathcal{A}) = \{w \in (2^{\Sigma})^{\omega} \mid w \text{ is accepted by } \mathcal{A}\}$. It is a standard result that for every scLTL formula φ, there exists a DFA $\mathcal{A}$ such that $\mathcal{L}(\varphi) = \mathcal{L}(\mathcal{A})$ and it is effectively constructible. Consequently, DFA can be used as a precise representation of an scLTL property. (Our approach can be extended in a straightforward way to so-called Büchi automata, which can express the whole of LTL.)

Definition 3 (State-labelled transition system)**.** *A (state-labelled) transition system (TS) is a tuple (S, s_0, Δ, L) where S is a finite set of states, s_0 is an initial state, $\Delta : S \to 2^S$ is a transition relation, and $L : S \to 2^{\Sigma}$ is the labelling function.*

A transition system (representing the real system or its abstraction) can be combined with an automaton (representing the property) into a *product*, see Fig. 1. Runs of the product are thus runs of the transition system monitored by the automaton. The automaton always reads the atomic propositions true in the current state and, on the whole, determines whether the run satisfies the property or not. This standard construction is often used in model checking [3] and we use it to improve RRG by *mutual* exchange of information between the two parts.

Definition 4 (Product)**.** *Given a TS $\mathcal{T} = (S, s_0, \Delta, L)$ and DFA $\mathcal{A} = (2^{\Sigma}, Q, q_0, \delta, F)$, the product $\mathcal{T} \times \mathcal{A}$ is the tuple $(S \times Q, \hat{s_0}, \hat{\Delta}, \hat{F})$ where*

- $\hat{s_0} = (s_0, \delta(q_0, s_0))$,
- $\hat{\Delta}((s, q)) = \{(s', \delta(q, s')) \mid s' \in \Delta(s)\}$,
- $\hat{F} = \{(s, q) \mid q \in F\}$.

III. Problem Formulation

Consider a robot deployed in an *a priori* unknown environment. We assume that the set of atomic propositions Σ (semantic labels, such as *living_room* or *wastebin*) is known beforehand, but not where they hold. In other words, the L function, as well as the obstacle-space, are unknown. Furthermore, we also assume that the robot is equipped with adequate sensors and perception modules that can identify labels and obstacles within a sensing radius r_s around its current position.

Problem 1. *Given an initial state $x_0 \in \mathcal{X}$ in an a priori unknown environment $\mathcal{X}$, and an scLTL specification Φ over the set of atomic propositions Σ, find a collision-free trajectory σ in $\mathcal{X}_{free}$ which satisfies Φ.*

Since neither obstacles nor the labeling function are known a priori, one cannot use traditional offline approaches described in Sec. I-A to solve Problem 1. The solution must be an online algorithm that learns the obstacle space and the labeling function as it moves in the environment. A straightforward way to solve this problem would be to explore the whole environment and assign labels to features in the environment first, and then use planning approaches. We propose to integrate exploration and planning. As a result, the robot attempts to make progress towards satisfying the specification while exploring, resulting in a possibly shorter travelled distance.

IV. Solution

Our solution to Problem 1 is an algorithm that learns interesting semantic dependencies and relations in the form of a *semantic abstraction* and utilizes this knowledge to bias the growth of a motion graph towards faster satisfaction of the desired LTL specification.

A. Semantic abstraction-guided RRG

The overall Semantic abstraction-guided RRG (SAG-RRG) procedure is overviewed in Alg. 1. Similarly to RRG, the procedure builds a graph $G = (V, E)$ whose vertices $v \in V$ lay within the obstacle-free space $\mathcal{X}_{\text{free}}$, and edges $e \in E$ connect two vertices if a collision-free trajectory exists. An iteration of the algorithm starts by updating the map of the environment with information of what is within the sensing radius r_s of the robot (line 5). After that, it computes the guidance according to the *Bias* function (line 6), which is described in more detail in Alg. 3 and Sec. IV-C. Then, each iteration of the internal *while* loop (lines 8-24) attempts to add one new vertex to G, in a similar way to the RRG algorithm. It samples a point from the known-space of the environment and finds its closest neighbour in the current graph (line 9). If the path connecting these two points is collision-free, the sample is considered for being added to the graph. If the symbolic counterpart of the sampled transition is in bias, the sampled point is stored as a "bias frontier"; otherwise it rejects this sample with some probability p (lines 11-14). This probability depends on how much you want to bias the sampling. The algorithm then follows the usual RRG procedure: it adds the new vertex and edge to the graph (line 15) and attempts to connect such vertex to its closest neighbours (lines 19-24), with slight modifications for checking for bias frontiers, and for keeping track of the symbolic transitions t_{symb} (lines 16 and 22) and states seen seen_st (line 17). After sampling a batch, it updates the semantic abstraction through the *Learn* procedure (line 25), which is detailed in Alg. 2 and in Sec. IV-B. The algorithm then calls the *Move* procedure (Alg. 4 and Sec. IV-D), which finds the best frontier to move to, and moves the robot to the point in G closest to it. Finally, the procedure checks if a plan that satisfies the LTL formula has been found.

Remark 1. In Alg. 1 an edge $e \in E$ is defined in a way to ensure the labels along it change only once. Formally, given an edge $e = (v_a, v_b) \in E$, there exists a state $x \in \sigma_{v_a}^{v_b}$ such that i) $L(x') = L(v_a)$, $\forall x' \in \sigma_{v_a}^{x}$, and ii) $L(x'') = L(v_b)$, $\forall x'' \in \sigma_{x+\epsilon}^{v_b}$, where $x + \epsilon$ represents a state in the neighbourhood of x.

Algorithm 1: SAG-RRG

```
Input: $\mathcal{X}, x_0, \Phi$
Output: A collision free trajectory in $\mathcal{X}$ which satisfies $\Phi$
1  Initialize semantic abstraction
2  $V \leftarrow x_0;\ E \leftarrow \emptyset$
3  curr_pos $\leftarrow x_0$; seen_st $\leftarrow s(x_0)$
4  while $\neg$AcceptingPath() do
5      UpdateMap(curr_pos, $r_s$)
6      $bias \leftarrow$ Bias(seen_st)
7      $t_{\text{symb}} \leftarrow \emptyset;\ i \leftarrow 0$
8      while $i <$ batch_size do
9          $[x_s, x_{near}] \leftarrow$ SampleAndExtend($\mathcal{X}_{free}, V$)
10         if CollisionFree($x_{near}, x_s$) then
11             if $(s(x_{near}), s(x_s)) \in bias$ then
12                 add $x_s$ to bias frontiers
13             else
14                 continue to next iteration with prob $p$
15             $E \leftarrow E \cup (x_{near}, x_s)$; $V \leftarrow V \cup x_s$
16             $t_{\text{symb}} \leftarrow t_{\text{symb}} \cup (s(x_{near}), s(x_s))$
17             seen_st $\leftarrow$ seen_st $\cup\ s(x_s)$
18             $i \leftarrow i + 1$
19         for $x \in$ Near($x_s$) do
20             if CollisionFree($x, x_s$) then
21                 $E \leftarrow E \cup (x, x_s)$; $V \leftarrow V \cup x$
22                 $t_{\text{symb}} \leftarrow t_{\text{symb}} \cup (s(x), s(x_s))$
23                 if $(s(x), s(x_s)) \in bias$ then
24                     add $x_s$ to bias frontiers
25     Learn($t_{\text{symb}}$)
26     curr_pos $\leftarrow$ Move()
27 return accepting_path
```

B. Learn

This section describes in detail the proposed approach to learning the semantic abstraction of the environment. Intuitively, we try to find transitions that are similar to the sampled ones and add them as special, potential transitions in the abstraction. Next part describes how we can accommodate these special transitions in the abstraction.

1) Semantic Abstraction: To formalize the semantic abstraction, we propose extending the state-labelled TS to a "multi-modal" transition system, our extension of modal transition systems [12]:

Definition 5 (Multi-Modal Transition System). *A tuple* $(S, s_0, \Delta, L, \mathbb{M}, \mathcal{M})$ *is called a multi-modal transition system (MM-TS), where* (S, s_0, Δ, L) *is a state-labelled transition system (Def. 3),* $\mathbb{M}$ *is a finite ordered set of modes, and* $\mathcal{M} : \Delta \to \mathbb{M}$ *is a modal marking.*

A semantic abstraction of an RRG graph is an MM-TS, where a discrete state $s \in S$ represents a set of points $x \in \mathcal{X}$ with the same labeling. With a slight abuse of notation, we use $x \in s$ to say that $L(x) = L(s)$ and $s(x)$ to denote $s \in S$, such that $x \in s$.

Intuitively, $\mathcal{M}$ assigns to each transition in the abstraction a "degree" of confidence that a corresponding transitions is present in the corresponding concrete points. We use two modes[2] in our MM-TS: *must* and *may*. The former is used for transitions that are known to exist based on samples taken from the environment while the graph is constructed; the latter is an extrapolation to which transitions might exist based on the *must* transitions. When a new edge (x, x_{new}) is added to the SAG-RRG graph G, a transition $(s(x), s(x_{new}))$ is added to the MM-TS as a *must* transition, and similar transitions (see Def. 7) are added as *may* transitions. Let us now formalize when we deem two transitions of a MM-TS *similar*.

Definition 6 (Domain of Change). *The domain of change for a transition* $(s, s') \in \Delta$ *is* $DoC(s, s') = L(s) \oplus L(s')$.

The *domain of change* is essentially the set of all atomic propositions which changed their valuation during the corresponding transition in the MM-TS. For example, given a transition (s, s') where $L(s) = \{a, b\}$ and $L(s') = \{b, c\}$, its $DoC(s, s')$ is $\{a, c\}$.

Definition 7 (Similar Transitions). *Two transitions* $(s, s'), (\bar{s}, \bar{s}') \in \Delta$ *are* similar *if and only if* $DoC(s, s') = DoC(\bar{s}, \bar{s}')$, *and* $\forall a \in DoC(s, s')$, $a \in L(s) \iff a \in L(\bar{s})$ *and* $a \in L(s') \iff a \in L(\bar{s}')$.

[2]Although we choose to use two modes in this paper for the sake of simplicity of the exposition, the approach presented throughout the paper is generic enough to use any number of modes. Besides *must* and *may*, one could also use *may not* and *must not*, for instance.

Algorithm 2: Learn

```
1 Function Learn($t_{\mathrm{symb}}$):
2     AddToProduct($t_{\mathrm{symb}}$, must)
3     $t_{sim} \leftarrow$ FindSimilar($t_{\mathrm{symb}}$)
4     AddToProduct($t_{sim}$, may)
```

Intuitively, similar transitions behave the same on their *domain of changes*. For example, a transition (s, s'), where $L(s) = \{a, b\}$ and $L(s') = \{b, c\}$, is similar to $(\bar{s}, \bar{s}')$ where $L(\bar{s}) = \{a, d\}$ and $L(\bar{s}') = \{d, c\}$. The idea is that after experiencing the transition (s, s') which leaves b untouched, we may hypothesize b is irrelevant and that the same behaviour is present also in the situation when b does not hold and when some other irrelevant proposition, e.g. d, holds. However, b still may be a precondition for the transition, hence we introduce the new transition $(\bar{s}, \bar{s}')$ only with a low "confidence".

The formal definition of similarity allows us to clearly identify when transitions in the MM-TS, i.e. the semantic abstraction of an RRG graph, should be marked as *may*.

2) Multi-modal product: The semantic abstraction captures existing and possible dependencies and relationships between labels in the environment regardless of the desired specification. We extend the definition of product (Def. 4) to incorporate the knowledge of the specification and thus enable biasing of SAG-RRG sampling to achieve faster specification satisfaction. In short, a multi-modal product (MM-P) is a product as in Def. 4 but with a MM-TS instead of a TS.

Definition 8 (Multi-modal Product)**.** *Given a MM-TS* $(S, s_0, \Delta, L, \mathbb{M}, \mathcal{M})$ *and a DFA* $\mathcal{A} = (2^{\Sigma}, Q, q_0, \delta, F)$*, their product (MM-P) is a tuple* $(S \times Q, \hat{s_0}, \hat{\Delta}, \hat{F}, \mathbb{M}, \hat{\mathcal{M}})$*, where the first four components are defined as in Def. 4 and the remaining two are the modes* $\mathbb{M}$ *and a model marking* $\hat{\mathcal{M}} : \hat{\Delta} \to \mathbb{M}$*, such that* $\hat{\mathcal{M}}((s, q), (s', q')) = \mathcal{M}(s, s')$.

Similarly to the multi-modal transition system, the product can be constructed iteratively, along with the construction of the SAG-RRG graph.

3) Learn procedure: The procedure Learn is summarized in Alg. 2. Given a set of transitions t_{symb}, this procedure adds them to the MM-TS as *must* transitions, since we know that these transitions are already there. After that, for each $t \in t_{symb}$, it computes the transitions similar to t and add them as *may* transitions in the MM-TS.

C. Bias

The *bias* procedure computes which transitions would more quickly bring the system to an accepting state of the LDBA. It returns a hierarchical list of transitions according to how far they are from an accepting state in MM-P; the closer a transition is to an accepting state, the better. These transitions can then be used to bias the construction of the motion graph for faster convergence.

The procedure, described in Alg. 3, starts by initializing the variables *bias* and reached, which store transitions and states, respectively. The first element of *bias* is the set of all transitions ending in accepting states of the MM-P (line 2). As for reached, it keeps track of all backwards-reachable states from the accepting states; hence its first element is the set of accepting states (line 3), and the second element is the pre-image of the accepting states (line 4). Then, until all the backward-reachable sets have been considered, *bias* is constructed iteratively based of the set of sampled states seen_st (lines 7-13). In the end, ith element of *bias* will be the set of states that can reach an accepting state after exactly i steps in the MM-P.

Algorithm 3: Bias

```
1  Function Bias(seen_st):
2      $bias[0] \leftarrow$ transitions ending in accepting states acc_st
3      reached[0] $\leftarrow$ acc_st
4      reached[1] $\leftarrow$ PreImg(acc_st)
5      all_reached $\leftarrow$ reached[0] $\cup$ reached[1]
6      $i \leftarrow 1$
7      while PreImg(reached[$i$]) $\not\subseteq$ all_reached do
8          useful_pre $\leftarrow$ PreImg(reached[$i$]) $\cap$ seen_st
9          useful_post $\leftarrow$ PostImg(useful_pre) $\cap$ reached[$i$]
10         $bias[i] \leftarrow$ (useful_pre, useful_post)
11         reached[$i+1$] $\leftarrow$ PreImg(reached[$i$])
12         all_reached $\leftarrow$ all_reached $\cup$ reached[$i+1$]
13         $i \leftarrow i + 1$
14     return $bias$
```

Algorithm 4: Move

```
1 Function Move:
2     $p_1 \leftarrow$ FindBestMapFrontier()
3     $p_2 \leftarrow$ FindBestBiasFrontier()
4     return Best($p_1, p_2$)
```

Learn and Bias functions work in unison and help each other improve. The more *may* transitions are learned, the better is the bias received. The better the bias, the more new transitions are learned and the faster it converges.

D. Move procedure

The idea behind the Move procedure, described in Alg. 4, is to decide where to move next: should we go towards a place that will provide more information about the map, or should we move according to the advice that has been given by Bias? In order to compare both options, we employ the concept of *information gain* (IG). Given a map frontier, its information gain is defined as $\mathrm{IG}_{\mathrm{map}} = \mathrm{size} \times f(d)$, where size is the size of the frontier and $f(d)$ is a strictly decreasing function (for $d > 0$) of the distance from the robot to the center of the frontier.

In a similar fashion, we define the information gain of a *bias* frontier. Note that *bias* frontiers were introduced in Alg. 1 (lines 12 and 24) as a means to keep track of the vertices in V that correspond to advices given by Alg. 3. Since *bias* is a list of transitions, we can associate a *rank* r with each transition from *bias* equal to $\mathrm{index} + 1$, where index is the index of the sampled transition. We define IG of these frontiers as $\mathrm{IG}_{\mathrm{bias}} = g(r, d)$, where g is some function such that both $g(r, \cdot)$ and $g(\cdot, d)$ are strictly decreasing, where d is again the distance from the robot to the frontier.

The intuition behind the IG of the *map* frontiers is to have a larger value the larger the frontier is, but penalise it according to its distance to the robot, so as to motivate exploration of smaller frontiers that are nearby. Similarly, with the IG of the *bias* frontiers, we want to motivate movement towards low-rank frontiers, since these are closer to satisfying the formula.

E. Analysis

Theorem 1. *The algorithm is sound, i.e. any trajectory returned by SAG-RRG satisfies the given scLTL formula* Φ.

Proof: (Sketch) The proposed algorithm iteratively constructs a product MM-P (Def. 4) between a semantic abstraction of the RRG graph and the automaton $\mathcal{A}$, which accepts exactly the language of the specification Φ. Paths in the product that visit accepting states project directly onto accepting runs of the automaton and runs of MM-TS, which in turn project directly onto paths in the RRG graph G and further onto trajectories of the robot in the workspace. Altogether, these trajectories necessarily satisfy Φ. ■

Theorem 2. *SAG-RRG is asymptotically complete.*

Proof: (Sketch) Follows directly from the convergence and completeness properties of the original RRG [10] and the fact that the biasing we introduced allows to eventually sample the whole space. Regardless of the scenario, including the one with no regularity in the environment that can be learned and exploited for guiding the search, the worse-case scenario will see the proposed approach perform an exhaustive search of the environment. ■

V. Implementation and Experiments

The proposed approach was implemented in Java and run on a consumer grade hardware (2.60GHz Intel i7-9750H CPU, 32 GB RAM). Binary Decision Diagrams (BDDs), which are very efficient for manipulating sets of Boolean variables, are used for storing and manipulating the product automaton, the labels of each node in the RRG, and the *bias*. We encode the RRG as an undirected graph whose nodes also store the labels that hold true at that state. JavaBDD and JGraphT are the libraries used for encoding the BDDs and the graph, respectively. We use the Java Spatial Index RTree library for spatial indexing and faster querying of nearest neighbours. We also use `owl` library for converting an LTL formula to an equivalent automaton and parse that automaton file using `jhoafparser`. The implementation currently takes three files as input describing environment, labelling and the property. There is also a command line interface with which you can configure the settings like using the bias or changing some parameters.

An example of the office-like environment used for the case study is presented in Fig. 2. In order to draw statistically-meaningful results, 100 different instantiations of the environment were randomly generated, in which the footprint (i.e. walls and doors) of the office space remains unchanged, but desks and wastebins are randomly positioned within the rooms (without blocking the door).

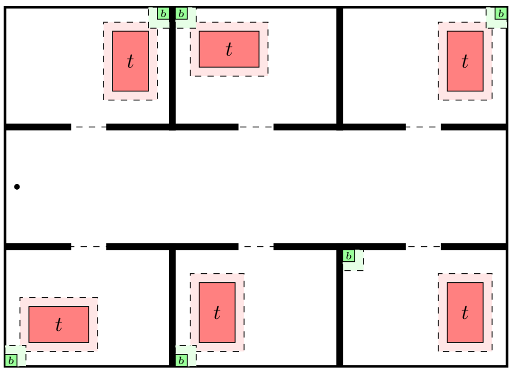


Figure 2. Example of an office-like environment used in the case study. Black solid lines represent walls. There are six rooms, each labeled with one atomic proposition r_i, for $i \in [1, 6]$, and a hallway labeled h. Each room contains a table (red) and a bin (green), labeled t and b, respectively. The labels of tables and bins hold true within the corresponding dashed and shaded areas surrounding it. The initial position of the robot is marked with a black dot on the left side of the hallway.

The scLTL specification is inspired by a realistic scenario, common in every office environment: reach a wastebin in the office rooms. We translate such a specification to the following scLTL formula:

$$\varphi = \mathbf{F}(r_1 \wedge b) \wedge \mathbf{F}(r_2 \wedge b) \wedge \ldots \mathbf{F}(r_6 \wedge b) \tag{1}$$

Note that such a specification does not impose any ordering of events.

For information gain, we use the following functions in our simulations

$$\mathrm{IG}_{\mathrm{map}}(m, p) = \frac{\mathrm{size}_m}{d_{m,p}} \tag{2}$$

$$\mathrm{IG}_{\mathrm{bias}}(x_s, p) = \frac{a}{r_{x_s}^b \, d_{x_s,p}} \tag{3}$$

where size_m is the size of frontier m, $d_{m,p}$ and $d_{x_s,p}$ are the length of the shortest path between p and m and x_s, respectively, $a, b > 0$ are user-defined parameters, and r_{x_s} is the rank of x_s. Adjusting a, b is intuitive: i) suppose m and x_s are equidistant from p; ii) fix r_{x_s} to 1 and choose a to reflect how a *bias* frontier compares to a *map* frontier; iii) now suppose $x_{s,1}, x_{s,2}$ equidistant from p, such that $r_{x_{s,1}} = 1$ and $r_{x_{s,2}} = 2$; iv) choose b as to reflect how the importance of *bias* frontier decays with its rank (e.g. linearly, quadratic). For our case study, we chose $a = 100$ and $b = 2$.

The results are presented in Table I for 100 randomly-generated environments. The solution presented in this paper can be seen as an approach that performs exploration of the environment and planning to satisfy the scLTL mission concurrently, without or with bias in building the RRG tree ('Simultaneous' and 'Simult. biased' columns in Table I). We compare this integrated solution to the trivial sequential

Table I
MEAN AND STANDARD DEVIATION OF THE TOTAL TRAJECTORY LENGTH TO SATISFY THE SCLTL MISSION (1), ALONG WITH TOTAL RUNTIME AND RRG SIZE. THE RESULTS WERE DRAWN FROM SIMULATING EACH APPROACH 3 TIMES IN EACH OF THE 100 RANDOMLY-GENERATED ENVIRONMENTS.

	See-through Desks			**Opaque Desks**		
	Explore, then plan	**Simultaneous**	**Simult. biased**	**Explore, then plan**	**Simultaneous**	**Simult. biased**
Total length	77.3 (7.5)	56.6 (8.0)	29.4 (5.0)	79.1 (7.1)	62.9 (16.5)	32.3 (11.8)
Exploration length	57.1 (3.2)	37.5 (7.1)	28.0 (4.9)	57.8 (4.9)	44.4 (16.6)	31.3 (12.1)
Remaining plan l.	20.2 (7.0)	19.1 (3.6)	1.3 (1.8)	21.3 (5.1)	18.5 (3.4)	1.1 (1.8)
Total Time	7.8 (2.0)	6.4 (2.3)	7.3 (1.9)	9.6 (2.5)	8.3 (3.2)	9.1 (2.4)
RRG size	1931.2 (460.9)	1938.6 (559.5)	1793.6 (312.1)	2313.8 (550.9)	1868.7 (498.2)	1901.4 (301.2)

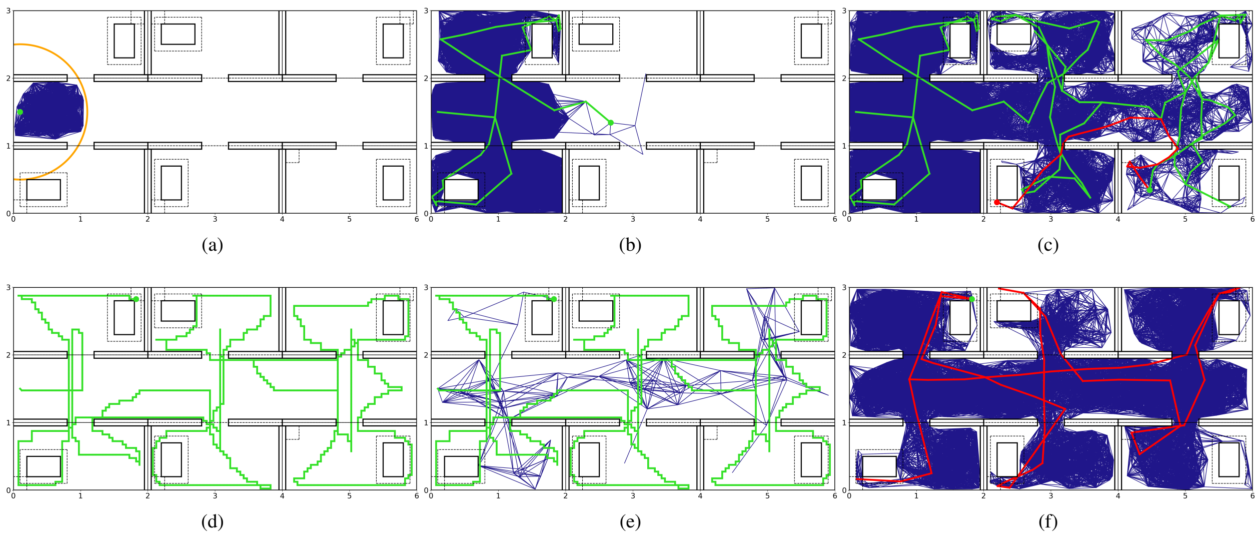


Figure 3. Snapshots of the robot navigating the office environment in the attempt to satisfy the scLTL mission (1) with two different approaches. The yellow semi-circle in (a) corresponds to the robot's sensing radius. The top-row figures (a-c) display the trajectory (green) when using the approach proposed by us (SAG-RRG), where exploration and planning are done together; the bottom-row figures (d-f) show the case where the robot first explores the environment, and only then it plans a path that satisfies the mission. The RRG graph at the time of the snapshot is shown in blue, and the path that leads to satisfaction of the mission is in red (c,f).

approach ('Explore, then plan' column in Table I), which consists of first exploring the whole environment, and then planning a trajectory that satisfies the mission. Lastly, in a more technical variant regarding the sensing capabilities of the robot, we analyse two cases, one where the robot can "see through" the desks (e.g., a flying robot), and another where they are considered to be Opaque. A few snapshots of the robot trajectory are shown in Fig. 3. Each approach is run three times in each of the 100 environments, totalling 1800 runs of the experiment.

The rows in Table I display the length ('Total length') of the trajectory traversed by the robot, from its initial state (common to all cases) until mission satisfaction, as well as the total computation time ('Total time') and number of nodes in the RRG graph ('RRG size'). Additionally, we also display the 'Exploration length' which, for the sequential approach, represents the length of the trajectory traversed only during the exploration phase, while for our approach it represents the length traversed until the system realises that a trajectory that satisfies the mission already exists. 'Remaining plan l.' is the length of the remaining trajectory that needs to be followed in order to satisfy the desired specification at the moment when exploration phase ends in the 'Explore, then plan' case, or the moment when the trajectory is found in the 'Simultaneous' and Simult. biased' cases. In Table I we see that having "see-through" desks makes the performance (both total length and time) slightly better in all the cases, which is to be expected as there are not as many occlusions in the map as with "opaque" desks. We also see that exploration and planning together performs better in general and including the bias makes it more than 2.5 times better than the naive approach in terms of the path length.

In the 'Explore, then plan' case, the robot's visits to wastebins during the exploration do not count towards the mission satisfaction, in contrast to the 'simultaneous unbiased' case. This is one of the reasons the latter performs better, as expected. The 'simultaneous biased' version performs a lot better because it was able to visit a lot of wastebins (with the help of biasing) already during the exploration.

VI. Conclusions and Future Work

We presented an online sampling-based algorithm capable of finding a trajectory in an *a priori* unknown environment that satisfies an scLTL specification. We enrich the RRG algorithm with functions that attempt to learn possible relations between labels of the environment and use such relations for biasing the search for a satisfying trajectory. The resulting paths are significantly shorter than in straightforward sequential exploration followed by planning in a known space.

A few topics to be considered for future work include extending the approach to consider probabilistic relationships in the semantic abstraction of the system, as well as the extension to multi-agent systems.

Acknowledgment

This work has been supported in part by the German Research Foundation (DFG) project 427755713 (KR 4890/3-1) *Group-By Objectives in Probabilistic Verification (GOPro)*, the research training group GRK 2428 *Continuous Verification of Cyber-Physical Systems (ConVeY)*, the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by Knut and Alice Wallenberg Foundation and the Swedish Research Council (VR). Fernando Barbosa and Jana Tumova are affiliated with the Division of Robotics, Perception and Learning and also Digital Futures.

References

[1] AI Medina Ayala, Sean B Andersson, and Calin Belta. Temporal logic motion planning in unknown environments. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5279–5284. IEEE, 2013.

[2] Alper Aydemir, Andrzej Pronobis, Moritz Göbelbecker, and Patric Jensfelt. Active visual object search in unknown environments using uncertain semantics. *IEEE Transactions on Robotics*, 29(4):986–1002, 2013.

[3] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*, volume 26202649. 01 2008. ISBN 978-0-262-02649-9.

[4] Amit Bhatia, Lydia E Kavraki, and Moshe Y Vardi. Sampling-based motion planning with temporal goals. In *2010 IEEE International Conference on Robotics and Automation*, pages 2689–2696. IEEE, 2010.

[5] Andreas Bircher, Mina Kamel, Kostas Alexis, Helen Oleynikova, and Roland Siegwart. Receding horizon "next-best-view" planner for 3D exploration. In *2016 IEEE international conference on robotics and automation (ICRA)*, pages 1462–1468. IEEE, 2016.

[6] Yiannis Kantaros and Michael M Zavlanos. Sampling-based optimal control synthesis for multirobot systems under global temporal tasks. *IEEE Transactions on Automatic Control*, 64(5):1916–1931, 2018.

[7] Yiannis Kantaros and Michael M Zavlanos. STyLuS*: A Temporal Logic Optimal Control Synthesis Algorithm for Large-Scale Multi-Robot Systems. *The International Journal of Robotics Research*, 39(7):812–836, 2020.

[8] Yiannis Kantaros, Matthew Malencia, Vijay Kumar, and George J Pappas. Reactive Temporal Logic Planning for Multiple Robots in Unknown Environments. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 11479–11485. IEEE, 2020.

[9] Sertac Karaman and Emilio Frazzoli. Sampling-based motion planning with deterministic μ-calculus specifications. In *Proceedings of the 48h IEEE Conference on Decision and Control (CDC) held jointly with 2009 28th Chinese Control Conference*, pages 2222–2229. IEEE, 2009.

[10] Sertac Karaman and Emilio Frazzoli. Sampling-based algorithms for optimal motion planning. *The International Journal of Robotics Research*, 30(7):846–894, 2011.

[11] Orna Kupferman and Moshe Y. Vardi. Model Checking of Safety Properties. In *CAV*, volume 1633 of *Lecture Notes in Computer Science*, pages 172–183. Springer, 1999.

[12] Kim Guldstrand Larsen and Bent Thomsen. A Modal Process Logic. In *LICS*, pages 203–210. IEEE Computer Society, 1988.

[13] Xusheng Luo, Yiannis Kantaros, and Michael M Zavlanos. An Abstraction-Free Method for Multi-Robot Temporal Logic Optimal Control Synthesis. *arXiv preprint arXiv:1909.00526*, 2019.

[14] Amir Pnueli. The Temporal Logic of Programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, page 4657, USA, 1977. IEEE Computer Society.

[15] Magnus Selin, Mattias Tiger, Daniel Duberg, Fredrik Heintz, and Patric Jensfelt. Efficient autonomous exploration planning of large-scale 3-D environments. *IEEE Robotics and Automation Letters*, 4(2):1699–1706, 2019.

[16] Pouria Tajvar, Fernando S Barbosa, and Jana Tumova. Safe Motion Planning for an Uncertain Non-Holonomic System with Temporal Logic Specification. In *2020 IEEE 16th International Conference on Automation Science and Engineering (CASE)*, pages 349–354. IEEE, 2020.

[17] Cristian Ioan Vasile and Calin Belta. Sampling-based temporal logic path planning. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 4817–4822. IEEE, 2013.

[18] Brian Yamauchi. A frontier-based approach for autonomous exploration. In *Proceedings 1997 IEEE International Symposium on Computational Intelligence in Robotics and Automation CIRA'97.'Towards New Computational Principles for Robotics and Automation'*, pages 146–151. IEEE, 1997.

Part II

Non-first Author Publications

C Planning via Model Checking With Decision-tree Controllers

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

Jonis Kiesbye, Kush Grover, Pranav Ashok, and Jan Křetínský. "Planning via model checking with decision-tree controllers". In: *2022 International Conference on Robotics and Automation, ICRA 2022, Philadelphia, PA, USA, May 23-27, 2022*. IEEE, May 2022, pp. 4347–4354. DOI: 10.1109/ICRA46639.2022.9811980

Summary. In the domain of robotics, *task planning* is the problem of finding a plan for a robot satisfying some given specification. In this paper, we converted it to a reachability problem for MDPs. The task at hand was for a Franka Emika robotic arm to efficiently retrieve objects from a container and place them onto a conveyor belt. During runtime, this bin-picking task can be susceptible to various faults such as environmental changes, inaccurate perception, or imprecise robot control. These faults require some kind of recovery action to be taken by the robot. We used the PRISM model checker to derive a controller that acts as a universal plan in contrast to other planners which gives a sequence of actions to execute. This universal controller usually is quite large making it an undesirable choice. We solved the problem by employing dtControl to transform it into a decision tree. The decision tree controller was significantly compact, more explainable, and orders of magnitude faster compared to the replanning approach. In addition to enhancing the controller, we generated another decision tree that pinpointed states with less likelihood of reaching the target state. This insight allowed for the incorporation of additional recovery actions, thereby enhancing the model's effectiveness.

Contributions of thesis author. The author played a significant role in the development of the theoretical ideas presented in the paper and made valuable contributions to the manuscript. In addition, he actively participated in discussions and provided feedback during the revision process.

controllers". IEEE, Conference Proceedings: 2022 International Conference on Robotics and Automation (ICRA), May/2022.

2022 IEEE International Conference on Robotics and Automation (ICRA)
May 23-27, 2022. Philadelphia, PA, USA

Planning via model checking with decision-tree controllers

Jonis Kiesbye Kush Grover Pranav Ashok Jan Křetínský

2022 IEEE International Conference on Robotics and Automation (ICRA) | 978-1-7281-9681-7/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICRA46639.2022.9811980

Abstract— **Planning problems can be solved not only by planners, but also by model checkers. While the former yield a plan that requires replanning as soon as any fault occurs, the latter provide a "universal" plan (a.k.a. strategy, policy, or controller) able to make decisions under all circumstances. One of the prohibitive aspects of the latter approach is stemming from this very advantage: since it is defined for all possible states of the system, it is typically so large that it does not fit into small memories of embedded devices. As another consequence of the size, its execution may be slow. In this paper, we provide a solution to this issue by linking the model checkers with decision-tree learners, resulting in decision-tree representations of the synthesized strategies. Not only are they dramatically smaller, but also more explainable and orders-of-magnitude faster to execute than plans with replanning. In addition, we describe a method for model validation and debugging via the model checker and the decision-tree learner in the loop. We illustrate the approach on our case study of a robotic arm for picking items in a real industrial setting.**

I. Introduction

The branch of planning called task planning [1] has been very extensively studied for decades. On the one hand, there is a tradition of planners, starting with STRIPS [2], and modeling languages such as Planning Domain Definition Language (PDDL) [3]. Planners produce a sequence of actions to be executed, the *plan*. However, in real-world scenarios, the environment may change during execution, perception may be inaccurate, and robot control may be imprecise. Consequently, whenever a discrepancy occurs, the plan cannot be used and *replanning* takes place, producing a new plan during runtime. On the other hand, the development in computer-aided verification has brought an alternative solution to planning via model checking [4]. This alternative became available also to continuous [5], temporal [6], or probabilistic [7] settings. There, the problems are typically modeled in some guarded command language, e.g. the PRISM language [8] in the probabilistic case. In contrast to planners, model checkers can in some settings produce a *strategy* (or *controller* or *policy*), a "universal" plan computed for all possible states of the system. Consequently, it can be followed and executed without any recomputation as everything is already precomputed. Although it avoids the replanning issue, it may still appear wasteful both in terms of size (storing all decisions for all possible states) and execution time (querying large lookup tables). This paper eliminates these drawbacks, provides a more efficient solution than planners and, in addition, utilizes the approach further, yielding a procedure to validate and improve the model.

We demonstrate our approach in a case study of a bin-picking robot with failure recovery in a real industrial setting. First, we model the problem in the PRISM language. This allows us to use the probabilistic model checker PRISM [8] to obtain the strategy. Second, we transform the strategy into a decision tree [9]. To this end, we use dtControl [10], a tool applied in formal verification for explaining controllers and counterexamples.

The produced strategy has several advantages over plans. Firstly, the decision-tree representation of the strategy is *small*. Secondly, due to its direct correspondence to if-else C-code, it is *orders of magnitude faster* to execute than plans with occasional replanning. Thirdly, the small size of the new representation also improves the explainability of the behavior [11], [12], [10].

On top of the advantages of our strategy representation, the model-checking approach offers additional, less expected benefits. Note that applying the model checker yields an answer to the planning problem for all states. For instance, in our probabilistic setting, besides the optimal actions to take, it yields the probability to finish a pick-cycle successfully for every state. A zero in any state then indicates the absence of failure recovery in that state. Hence we query the model checker to also return all states with zero probability, thereby obtaining a list of situations where failure recovery is not available yet. We show how to present this list also as an explainable summary, again in the form of a decision tree. Consequently, engaging the model checker in the loop helps us to *debug the model* and identify the fault states where recovery actions are missing.

Our contribution can thus be summarized as follows:

- novel benefits of the model checking approach to task planning, obtained by employing decision trees, in particular: easier modeling, low memory requirements, faster execution;
- a way to debug and improve the model;
- their demonstration on an industrial case study.

II. Related Work

Hierarchical planning is a well-studied problem in robotics [13]. Here, *high-level tasks are planned* by one layer and the low-level motion primitives are planned by another layer [14].

For task planning problems in robotics, researchers have used hierarchical task networks [15], PDDL planners [16], [17], Answer Set Programming [18], Boolean satisfiability [19], [20], [21], and SMT satisfiability [22].

Jonis Kiesbye, Kush Grover and Jan Křetínský are with the Technical University of Munich, Germany.
Pranav Ashok was with the Technical University of Munich, Germany. He is now with the Fraunhofer IKS, Germany.

In [23], [24], [25] temporal logics are used to perform task planning that is feasible also according to the dynamics. See [26] for a comprehensive survey of such approaches.

The use of *model checkers in planning* has also been well studied. Traditionally, model-checking-based planners operated in the non-deterministic domain [27], [28], [29], [30], but richer modeling formalisms such as timed automata [31] and probabilistic systems such as Markov Decision Processes (MDP) have also been explored [7]. In the context of robotics, model checkers have often been used to verify the high-level plans for applications such as robot swarms [32], [33], unmanned aerial vehicles [34] or autonomous surface vehicles [35] among others [36], [37]. [38] surveys work where formal methods have been used in the context of robotics.

For fast execution, *strategy representation* can be based on binary decision diagrams (BDD) [39] or algebraic decision diagrams (ADD) [40] as they are much more succinct than explicit state-action tables. However, they also come with many disadvantages as discussed in [10], [11]. To tackle this problem, the formal methods community has made use of *decision trees* to represent controllers and counterexamples arising out of model checking MDPs, stochastic games and Linear Temporal Logic (LTL) synthesis [11], [41], [12], [42]. Decision trees have also been used to represent hybrid system controllers explainably and succinctly [10], and learned policies from reinforcement learning [43].

For MDPs, [44], [45] use out-of-the-box strategies given by the model checkers, which can be huge lookup tables and, therefore, very slow to execute. In our paper, we convert them to decision trees, making them very compact and efficient while avoiding problems associated with BDDs and ADDs. We validate our approach on a real industrial case study.

Property-driven modeling in robotics has been explored in [46]. It repeatedly alternates between modeling and model checking in order to better validate the models. However, it treats only stochastic processes (specifically, Markov chains) and not controller synthesis.

Bin-picking is a common use case as it mirrors many of today's robotic challenges [47]. Progressing towards soft objects of unknown geometry requires better grasp pose estimation [48], incorporating haptic feedback [47], and providing corrective actions to deal with unsuccessful grasps [49].

III. Preliminaries

a) Markov Decision Processes (MDP) [50]: are a widely used model for systems with non-deterministic and probabilistic behavior. Intuitively, the process is always in one of finitely many states S, in which some of the actions A are available. Choosing one determines a probability distribution on the next state to which the process moves.

Definition 1. *An MDP is a tuple $\mathcal{M} = (S, Act, Av, E, s_0)$ where, S is a finite set of states, Act is a finite set of actions, $Av : S \to 2^{Act} \backslash \phi$ maps each state to the set of actions available there, $E : S \times Act \to Dist(S)$ is a probabilistic transition function that assigns each state action pair to a distribution over the successor states, and s_0 is the initial state.*

A *strategy* for an MDP is intuitively a way to resolve these choices. Formally, it is a function $\sigma : S \to A$ which picks an available action in every state.

An MDP combined with a strategy yields a Markov chain (MC), a fully stochastic process. Each state of the MC can be associated with a *value* equal to the probability of reaching a given target state from here. An *optimal strategy* for MDP $\mathcal{M}$ and a target set of states B is the one that maximizes the probability of reaching B from each state. This maximum probability of reaching the target set from a state s is also called the *optimal value* of s.

b) Model checking [51]: is the process of verifying whether a system satisfies a given specification. Various model checkers exist for non-deterministic, timed, probabilistic, and hybrid models (e.g., [8], [52], [53]). As a by-product of the verification task, model checkers often give additional useful outputs. If a specification is not satisfiable, a counterexample may be produced. If the specification is satisfiable, a strategy may be returned. For MDPs, the most common publicly available tools are PRISM [8] and STORM [54], which can compute the optimal strategy and values for any given MDP.

c) Decision trees (DT) [9]: are a popular data structure commonly used in machine learning in the context of classification and regression, known for its interpretability. The tree is commonly visualized as a top-down binary tree with each node containing a Boolean predicate over a set of input variables and every tree edge corresponding to either *true* or *false*. The leaf nodes of the binary tree contain the decisions. Intuitively, given a set of input variable valuations, a decision can be found by evaluating the predicate in each node and following the respective edge down the tree.

A natural generalization of binary predicates are multi-comparison predicates [55]. In this setting, the decision node just contains one of the input variables which can take values in a finite domain and has children corresponding to every value in the finite domain.

IV. Problem Description

We develop our approach on a concrete case study of a robotic arm. While the approach is independent of the case study, the concrete setting allows for an easier explanation and demonstration of its advantages.

In the context of the Supervised Autonomous Interaction in uNknown Territories (SAINT) project, a Franka Emika robot arm, shown in Fig. 1, shall pick clothes from a box and place them on a sorter tray in a warehouse. If all picks have been completed, the robot shall request a new box, detect it, receive orders for that box from the facility, and start picking again. The robot can be instructed using high-level actions, which make use of motion primitives provided by a motion planner module [56], e.g., moving the robot, grasping items, placing them on a tray, and measuring the

Fig. 1. Franka Emika robot in the logistics facility

TABLE I
SOME STATE VARIABLES AND THEIR VALUES FOR THE BIN-PICKING PROCESS

variable	type	values and (index)
item	object	unknown(0), detected(1), grasped(2), probably_grasped(3), lost(4), placed(5), stuck_in_gripper(6)
box	object	unknown(0), in_arrival(1), arrived(2), detected(3), info_received(4), decommissioned(5), none_present(6)
obstacle	object	unknown(0), obstacle_present(1)
gripper	device	unknown(0), free(1), occupied(2)
robot	device	unknown(0), find_box_pose(1), inspect_box_pose(2), post_grasp_pose(3), pre_place_pose(4), place_pose(5), check_empty_pose(6), home_pose(7), find_obstacle_pose(8)
graspability	property	unknown(0), unlikely(1), very_unlikely(2), unfeasible(3)

weight in the gripper. The high-level controller just knows the values given in Table I and forwards them to the motion planner, similar to [21]. A vision module provides services for detecting the box and the items therein [57]. A facility module forwards information to and from the warehouse control system. The robot system uses the Robot Operating System (ROS) for inter-process communication. Since the aforementioned software modules are not relevant for the task planner itself, we will not look at them in detail.

The challenge here is to implement a controller that can use the high-level actions to satisfy certain requirements, e.g., ensure that a pick-cycle is completed irrespective of failures in executing certain motion primitives. A human operator can provide assistance via semi-autonomous actions where the robot requests, e.g., marking the outline of the object if the autonomous object detection should repeatedly fail. The controller needs to solve faults autonomously whenever possible and find the next action without considerably increasing the cycle time for a pick.

In the following, we will focus on how the task planning is realized through the model checking paradigm. We will improve the model of the system, show how to synthesize a controller and evaluate it in the real warehouse. The problem formulation and our solution can be generalized in a straightforward way to task planning for finite-state abstractions of systems with logical (e.g. linear temporal logic) or reward-based specifications.

V. OUR APPROACH AND ITS ADVANTAGES

We present a paradigm for automatic controller synthesis that systematically helps the engineer to (i) model the robot as an MDP, (ii) analyze fault scenarios and coverage of recovery actions using a model checker in the loop, and (iii) automatically synthesize a controller in the form of a decision tree that can then be executed on the robot.

A. *Case Study*

The state of the bin-picking robot and its environment is described by a set of 14 finite variables (some of them shown in Table I) with over a million reachable states, disallowing for manual design and analysis. The actions have preconditions that ensure that the action can take place safely and successfully. The goal of the robot is to place an item on the moving sorter successfully. Based on its initial state, it will first need to register a new box, detect the item, check its bar code, and grasp the item. There are certain considerations to be taken into account while modeling in order to let the model checker handle as many fault scenarios as possible. In the real world, faults can occur that take the system to states that the modeler did not expect, and are hence not reachable following the transitions in the model. Accounting for this, we err on the side of caution and mark every physically feasible state in the model as an initial state. Hence, the strategy synthesized by the model checker can handle even those states that are not reachable from the real initial state of the robot, which makes the strategy fault-tolerant.

B. *Planning with PDDL*

Before we describe our model-checking approach, we explain how this problem is usually solved. This is shown in Fig. 2 in green: a PDDL domain is hand-crafted and during runtime, the planner finds a plan from the current state of the process. However, writing a PDDL domain is time-consuming and failure-prone compared to writing a PRISM model (see section V-C). Therefore, if one wants to use PRISM language just for writing the model and use PDDL planners for the real-time control of the process, there is the option for exporting the model to PDDL. ePMC (formerly known as IscasMC) [58] can convert PRISM models to JANI format and the converter published in [7] transforms JANI to Probabilistic PDDL. The blue pipeline in Fig. 2 shows this modified approach.

Listing 1. Our PRISM model for the bin-picking problem
```
mdp
module task_planner
         ⋮
    [move_to_find_box_pose] obstacle=0 & gripper=1 & robot=7
          & movability=0 -> (robot'=1);
         ⋮
endmodule
   ⋮
init item>=0 & barcode>=0
   ⋮
endinit
```

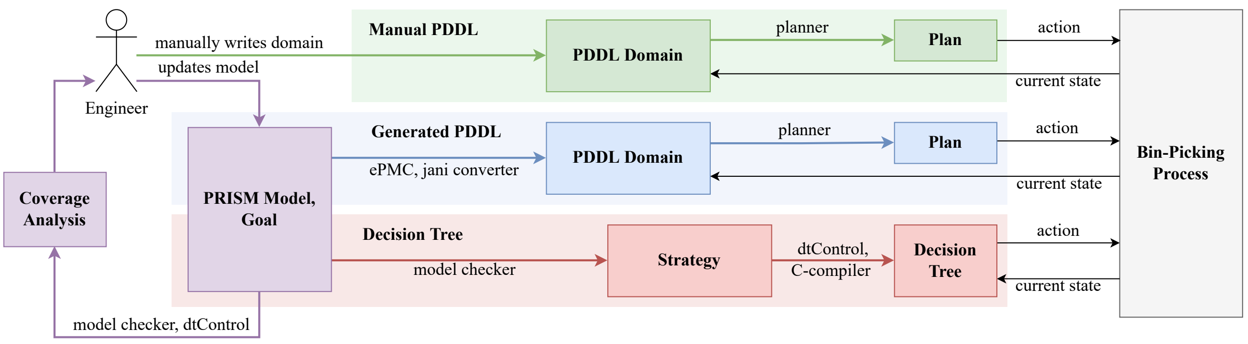


Fig. 2. Architecture of the Modeling and Control Framework

Listing 2. PDDL action generated by converting the PRISM model to PDDL format, equivalent to Line 22 of Listing 1

```
    (:action move_to_find_box_pose
        :parameters ()
        :precondition (and
                (value movability n0) (value robot n7)
                (value obstacle n0) (value gripper n1))
        :effect (and
                (not (value robot n7)) (value robot n1))
    )
```

C. Modeling and Model Checking with PRISM

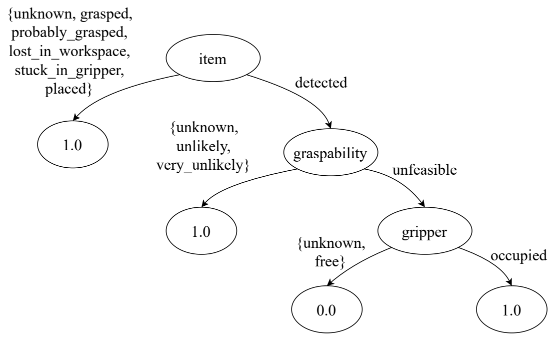
We use PRISM for creating and analyzing an MDP representing the robotic bin-picking process. The PRISM language is based on guarded commands, similar to PDDL, with preconditions and effects on the left and right of the "$\rightarrow$" symbol, respectively, for each action. An excerpt of the model is shown in Listing 1. Line 22 of Listing 1 illustrates how actions are written in the PRISM language.

While PDDL would also allow us to write an equivalent description as shown for the action `move_to_find_box_pose`, which moves the robotic arm to a pose from which the whole box is visible, in Listing 2, PRISM's more convenient syntax and its simulation capabilities made it more suitable for our application. Finally, Lines 102-108 describe the set of initial states.

The goal of the robot is to arrive in a state where `item=placed` and `gripper=free`, indicating a successful place. PRISM will compute a strategy that maximizes the probability of reaching the goal. The goal here is a reachability query but PRISM also has the capability to find strategies for arbitrary LTL specifications [8]. The strategy is exported as a lookup table that gives the appropriate action for every reachable state.

D. Improving the Model

After modeling the nominal actions in PRISM, it is obvious that they will not suffice to resolve the faulty states contained in the set of initial states. In a trivial approach towards fault tolerance, one could add an action "`technician_fixes_the_problem`" with no precondition that directly takes the model from any state to the goal state. But since there are quite a few autonomous options for fault recovery, we should exploit those first. In this section, we



{unknown, grasped, probably_grasped, lost_in_workspace, stuck_in_gripper, placed}
item
detected
1.0
{unknown, unlikely, very_unlikely}
graspability
unfeasible
1.0
gripper
{unknown, free}
occupied
0.0
1.0

Fig. 3. Decision trees show the variable combinations that decide whether the initial states are *solvable* or not.

design a procedure to find problematic states and refine the model based on those. This procedure stems from a novel combination of a model checker and a decision-tree learner.

We call states that have a path to a target state under the current optimal strategy *solvable*, implying that there are recovery actions available if it is a fault state. Adding recovery actions does not guarantee that all fault scenarios become *solvable* and without model checking, the designer can never be sure whether all faults will be handled. We use model checking to find the *solvable* and *unsolvable* states and display them in a tree view as shown in Fig. 3. From the tree, the designer can easily find problematic combinations of variables and implement recovery actions to handle those. The purple loop in Fig. 2 shows our proposed iterative workflow of the model checker analyzing the model and the engineer updating it.

We illustrate the process with an example. Consider the decision tree in Fig. 3, encoding the probabilities to reach the target as returned by the model checker. This concrete tree captures that almost all initial states are *solvable*, indicated by `1.0`; only the combination of `item=detected`, `graspability=unfeasible` and either `gripper=unknown` or `gripper=free` is *unsolvable*, indicated by `0.0`. By conceiving an action to recover from this situation, e.g. asking the human operator to mark a grasp point, the model can achieve perfect fault tolerance and the tree will simplify to `1.0`.

The approach can also be used to increase the "recoverability" of certain fault situations. If the decision tree has

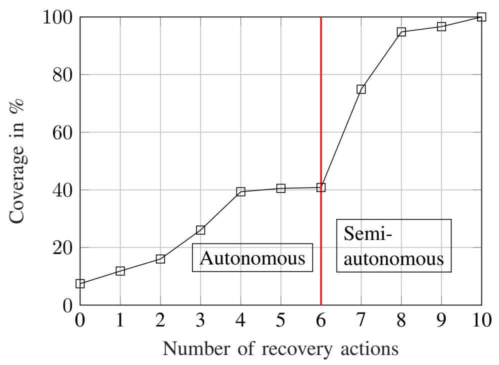
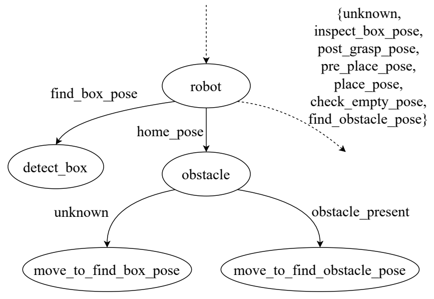


Fig. 4. Percentage of *solvable* initial states as new recovery actions are added through our *Coverage analysis* cycle. Actions 1–6 are autonomous, meaning the robot can perform them on its own, and actions 7–10 are semi-autonomous, requiring human intervention during execution.



robot
find_box_pose
home_pose
{unknown, inspect_box_pose, post_grasp_pose, pre_place_pose, place_pose, check_empty_pose, find_obstacle_pose}
detect_box
obstacle
unknown
obstacle_present
move_to_find_box_pose
move_to_find_obstacle_pose

Fig. 5. An excerpt of the decision tree representing the PRISM strategy.

nodes with non-zero numbers and the designer wants the success probability of all states to be greater than some threshold, they can look at the nodes below that threshold and add some actions which increase the probability of success. This also provides a way for the designer to prioritize states where to add recovery actions next, e.g., states with the lowest probabilities to succeed.

In most situations, the designer needs to add many recovery actions to increase the coverage of the model, i.e. the percentage of initial states that have a path to the target under the current strategy. Left of the red line in Fig. 4 shows the coverage improving from 7.4 % to 40.8 % when adding 6 autonomous recovery actions to the model of our bin-picking robot.

To bring the coverage to 100 %, we design semi-autonomous actions that ask a human operator for crucial data like the bar code of the item or the position of the box. These semi-autonomous actions are implemented so that the robot gathers the required camera pictures automatically and stays in a safe configuration until the operator answers the request. The operator can work from a remote location and tend to multiple robots. The area to the right of the red line in Fig. 4 shows the coverage when subsequently adding 4 semi-autonomous actions.

E. Representing the Controller as Decision Trees

Once we get the automatically synthesized policy from PRISM (or alternatively, other model checkers such as STORM), we use dtControl [10] to transform the strategy into a decision tree as shown by the red path in Fig 2. An excerpt of such a decision tree is given in Figure 5. It is easier to interpret and more compact than the lookup table.

dtControl exports the policy as a JSON file, as C-code, and as a DOT-file, which the user can visualize. The C-code generated by dtControl can be compiled into a shared library and called by the computer controlling the process. This way, we obtain controllers with small size, fast execution, and explainability at once.

With some changes to the source code, PRISM can be made to export the BDD which can also be translated into highly performant C-code. However, in BDDs, the state variables are bit-blasted, i.e. split into multiple binary variables. This makes it extremely difficult for the layperson to decipher the behavior. The DT representation on the other hand, as can be seen in Fig. 5, is very easy to interpret.

VI. Experimental Results

A. Coverage Analysis

Both the model checker PRISM and dtControl were run on an Intel Xeon E5-2630 CPU with 192 GB of RAM, utilizing a single core. With the bin-picking model having 1,357,376 reachable states, deriving the share of *solvable* initial states took 8.7 seconds at maximum. Converting PRISM's results into a decision tree using dtControl for further analysis took 8 minutes and 50 seconds. The maximum memory utilization was 1.5 GB. Performing the coverage analysis with a smaller and a larger model revealed that the time for the analysis scales proportionally with the number of states as can be seen in Table II.

B. Controller Synthesis

As shown in Fig. 2, one can control the process/system by running a planner on a manually written PDDL domain, or one generated from the PRISM model, or by synthesizing a decision tree from the PRISM generated strategy.

When following the decision-tree approach for the bin-picking model with 1,357,376 reachable states, PRISM took 6 seconds for model checking and exporting the strategy as shown in Table II. Deriving the decision tree with dtControl using the attribute-value grouping parameter (`--use-preset avg`) took 2 minutes and 35 seconds and yielded a tree with 213 inner nodes. dtControl outputs the tree as C-code as well. Compiling the C-code into a shared library that we can call from a ROS node takes 0.098 seconds. The library is only 24.4 kB in size and takes little resources to run.

Generating an equivalent PDDL domain with our PRISM model as input takes 10.0 seconds. The PDDL domain is used by the planner Metric-FF during the operation of the robot.

C. Real-World Evaluation

A robot-control PC, equipped with a six-core Intel i7-8700K CPU and 32 GB RAM runs the motion planner, the vision module, and the library executing the decision tree. Every action is implemented as a Python function in a SMACH state description [59] that verifies the guard and

TABLE II

Computation times for performing a coverage analysis and for generating a controller depending on the size of the model

no. of reachable states	coverage analysis		controller synthesis			PDDL conversion
	model checking	decision tree	model checking	decision tree	compilation	
30,726	1.1 s	16.3 s	1.1 s	7.3 s	0.094 s	8.3 s
1,357,376	8.7 s	530.5 s	6.1 s	155.0 s	0.098 s	10.0 s
13,421,120	129.3 s	7356.2 s	34.8 s	2669.4 s	0.182 s	10.3 s

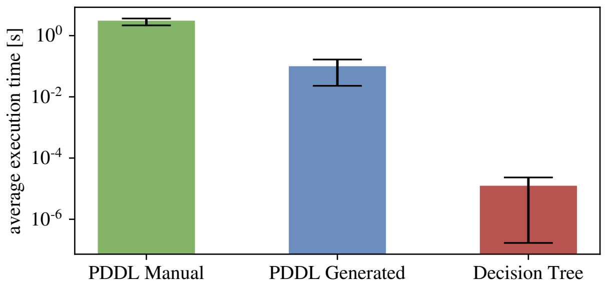


Fig. 6. Average time taken (with 3σ confidence intervals) for computing the next action to perform by different approaches shown in Fig. 2. Note the logarithmic scale.

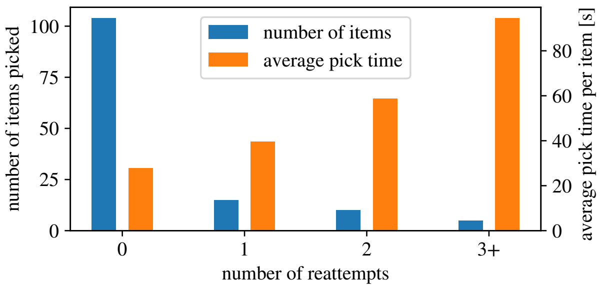


Fig. 7. Distribution of 134 picks over the number of reattempts needed, along with the average pick times.

calls the motion planner and vision module. A state observer updates the state variables.

With the robot taking about 30 seconds per cycle, it is desirable to keep the time spent for planning the action to be executed next down to about 1 % of the cycle time, i.e. 0.3 seconds. Fig. 6 shows the mean time that the decision tree and the PDDL planner took for returning the next action or a new plan. Executing the Metric-FF planner with a hand-written domain file from [60] took 2.9 seconds on average, thus significantly increasing the cycle time of the robot. Using the generated PDDL domain with the same planner yields a considerable improvement with an average execution time of 0.094 seconds, thereby not significantly impeding the cycle time anymore.

In contrast, finding the next action takes only about 11 *micro*seconds on average when using the decision tree. Assuming that the plans are 9 actions in length on average, the performance improvement of the decision tree compared to PDDL is in the range of three orders of magnitude.

To evaluate the robustness of the controller in real-world conditions, a series of trial runs were executed over two days in the warehouse. We observed the pick and place maneuvers of 134 items leading to 58 faults (e.g. the camera could not find the item in the box, the robot grasped two items instead of one, etc.). The controller found an autonomous solution for almost all faults, usually consisting of an appropriate recovery action and a reattempt of the failed action. After failing five times on one item, the controller switches from the autonomous to the semi-autonomous strategy and will then ask a human operator for additional input.

Fig. 7 shows how many items were picked and how many reattempts they needed. One can see that the cycle time is less than 30 seconds for items that are successfully placed on the first try and the average time rises with the recovery actions and reattempts executed in the reattempts.

VII. Conclusion and Future Work

We have improved task planning via model checking by employing decision trees as a compact and interpretable representation for the outputs of model checkers. As a consequence, strategies become easier to obtain and to run than plans, decreasing the execution time and therefore computational resources required by orders of magnitude, making decision trees a very viable option for controlling extremely resource-constrained systems. We validated our approach with an industrial case study.

Since the tool dtControl is applicable also to strategies for hybrid systems [10], one can lift this approach to robotic problems that require modeling over continuous domains. Additionally, since model checkers, including PRISM, often support rich specifications such as temporal logic formulae or multi-dimensional rewards and costs, our workflow would work the same even for such extensions. Further case studies could thus confirm the practical advantages also in this wider range of settings.

Acknowledgement

The authors would like to thank the other project partners in the SAINT project, namely the Chair of Applied Mechanics at TUM, Franka Emika GmbH, and FIEGE Logistics for their contributions to the project's success. The SAINT project was funded by the Bayerische Forschungsstiftung (AZ-1318-17). The work was partially supported also by DFG projects 383882557 *Statistical Unbounded Verification (SUV)*, 427755713 *Group-By Objectives in Probabilistic Verification (GOPro)* and research training group GRK 2428 *Continuous Verification of CYber-Physical Systems (ConVeY)*, and by the Bavarian Ministry for Economic Affairs, Regional Development and Energy as part of a project to support the thematic development of the Institute for Cognitive Systems.

References

[1] M. Ghallab, D. S. Nau, and P. Traverso, *Automated planning - theory and practice*. Elsevier, 2004.

[2] R. Fikes and N. J. Nilsson, "STRIPS: A new approach to the application of theorem proving to problem solving," *Artif. Intell.*, vol. 2, no. 3/4, pp. 189–208, 1971.

[3] D. V. McDermott, "The 1998 AI planning systems competition," *AI Mag.*, vol. 21, no. 2, pp. 35–55, 2000.

[4] A. Cimatti, F. Giunchiglia, E. Giunchiglia, and P. Traverso, "Planning via model checking: A decision procedure for *AR*," in *ECP*, ser. Lecture Notes in Computer Science, vol. 1348. Springer, 1997, pp. 130–142.

[5] S. Bogomolov, D. Magazzeni, A. Podelski, and M. Wehrle, "Planning as model checking in hybrid domains," in *AAAI*. AAAI Press, 2014, pp. 2228–2234.

[6] A. Heinz, M. Wehrle, S. Bogomolov, D. Magazzeni, M. Greitschus, and A. Podelski, "Temporal planning as refinement-based model checking," in *ICAPS*. AAAI Press, 2019, pp. 195–199.

[7] M. Klauck, M. Steinmetz, J. Hoffmann, and H. Hermanns, "Bridging the gap between probabilistic model checking and probabilistic planning: Survey, compilations, and empirical comparison," *J. Artif. Intell. Res.*, vol. 68, pp. 247–310, 2020.

[8] M. Z. Kwiatkowska, G. Norman, and D. Parker, "PRISM 4.0: Verification of probabilistic real-time systems," in *CAV*, ser. Lecture Notes in Computer Science, vol. 6806. Springer, 2011, pp. 585–591.

[9] T. M. Mitchell, *Machine learning, International Edition*, ser. McGraw-Hill Series in Computer Science. McGraw-Hill, 1997.

[10] P. Ashok, M. Jackermeier, P. Jagtap, J. Kretínský, M. Weininger, and M. Zamani, "dtcontrol: decision tree learning algorithms for controller representation," in *HSCC*. ACM, 2020, pp. 17:1–17:7.

[11] T. Brázdil, K. Chatterjee, M. Chmelik, A. Fellner, and J. Kretínský, "Counterexample explanation by learning small strategies in markov decision processes," in *CAV (1)*, ser. Lecture Notes in Computer Science, vol. 9206. Springer, 2015, pp. 158–177.

[12] P. Ashok, J. Kretínský, K. G. Larsen, A. L. Coënt, J. H. Taankvist, and M. Weininger, "SOS: safe, optimal and small strategies for hybrid markov decision processes," in *QEST*, ser. Lecture Notes in Computer Science, vol. 11785. Springer, 2019, pp. 147–164.

[13] E. Frazzoli, M. A. Dahleh, and E. Feron, *A Hybrid Control Architecture for Aggressive Maneuvering of Autonomous Aerial Vehicles*. Boston, MA: Springer US, 2000, pp. 325–343.

[14] E. Frazzoli, M. A. Dahleh, and E. Feron, "Maneuver-based motion planning for nonlinear systems with symmetries," *IEEE Transactions on Robotics*, vol. 21, no. 6, pp. 1077–1091, 2005.

[15] L. P. Kaelbling and T. Lozano-Pérez, "Hierarchical task and motion planning in the now," in *2011 IEEE International Conference on Robotics and Automation*, 2011, pp. 1470–1477.

[16] S. Srivastava, E. Fang, L. Riano, R. Chitnis, S. Russell, and P. Abbeel, "Combined task and motion planning through an extensible planner-independent interface layer," in *2014 IEEE International Conference on Robotics and Automation (ICRA)*, 2014, pp. 639–646.

[17] Y.-q. Jiang, S.-q. Zhang, P. Khandelwal, and P. Stone, "Task planning in robotics: an empirical comparison of pddl- and asp-based systems," *Frontiers of Information Technology Electronic Engineering*, vol. 20, pp. 363–373, 03 2019.

[18] E. Erdem, V. Patoglu, and P. Schüller, "A systematic analysis of levels of integration between high-level task planning and low-level feasibility checks," *AI Communications*, vol. 29, no. 2, pp. 319–349, 2016.

[19] H. Kautz and B. Selman, "Planning as satisfiability." 01 1992, pp. 359–363.

[20] J. Rintanen, "Engineering efficient planners with sat," *Frontiers in Artificial Intelligence and Applications*, vol. 242, pp. 684–689, 01 2012.

[21] G. Havur, K. Haspalamutgil, C. Palaz, E. Erdem, and V. Patoglu, "A case study on the tower of hanoi challenge: Representation, reasoning and execution," in *2013 IEEE International Conference on Robotics and Automation*, 2013, pp. 4552–4559.

[22] S. Nedunuri, S. Prabhu, M. Moll, S. Chaudhuri, and L. E. Kavraki, "Smt-based synthesis of integrated task and motion plans from plan outlines," in *2014 IEEE International Conference on Robotics and Automation (ICRA)*, 2014, pp. 655–662.

[23] G. E. Fainekos, H. Kress-Gazit, and G. J. Pappas, "Temporal logic motion planning for mobile robots," in *ICRA*. IEEE, 2005, pp. 2020–2025.

[24] C. Belta, A. Bicchi, M. Egerstedt, E. Frazzoli, E. Klavins, and G. J. Pappas, "Symbolic planning and control of robot motion [grand challenges of robotics]," *IEEE Robotics Autom. Mag.*, vol. 14, no. 1, pp. 61–70, 2007.

[25] H. Kress-Gazit, G. E. Fainekos, and G. J. Pappas, "Temporal-logic-based reactive mission and motion planning," *IEEE Trans. Robotics*, vol. 25, no. 6, pp. 1370–1381, 2009.

[26] E. Plaku and S. Karaman, "Motion planning with temporal-logic specifications: Progress and challenges," *AI Commun.*, vol. 29, no. 1, pp. 151–162, 2016.

[27] F. Kabanza, M. Barbeau, and R. St.-Denis, "Planning control rules for reactive agents," *Artif. Intell.*, vol. 95, no. 1, pp. 67–11, 1997.

[28] R. M. Jensen and M. M. Veloso, "Obdd-based universal planning for synchronized agents in non-deterministic domains," *J. Artif. Intell. Res.*, vol. 13, pp. 189–226, 2000.

[29] A. Cimatti, M. Pistore, M. Roveri, and P. Traverso, "Weak, strong, and strong cyclic planning via symbolic model checking," *Artif. Intell.*, vol. 147, no. 1-2, pp. 35–84, 2003.

[30] Y. Li, J. S. Dong, J. Sun, Y. Liu, and J. Sun, "Model checking approach to automated planning," *Formal Methods Syst. Des.*, vol. 44, no. 2, pp. 176–202, 2014.

[31] M. M. Quottrup, T. Bak, and R. Izadi-Zamanabadi, "Multi-robot planning: a timed automata approach," in *ICRA*. IEEE, 2004, pp. 4417–4422.

[32] S. Konur, C. Dixon, and M. Fisher, "Analysing robot swarm behaviour via probabilistic model checking," *Robotics Auton. Syst.*, vol. 60, no. 2, pp. 199–213, 2012.

[33] M. Massink, M. Brambilla, D. Latella, M. Dorigo, and M. Birattari, "On the use of bio-pepa for modelling and analysing collective behaviours in swarm robotics," *Swarm Intell.*, vol. 7, no. 2-3, pp. 201–228, 2013.

[34] R. Hoffmann, M. L. Ireland, A. Miller, G. Norman, and S. M. Veres, "Autonomous agent behaviour modelled in PRISM - A case study," in *SPIN*, ser. Lecture Notes in Computer Science, vol. 9641. Springer, 2016, pp. 104–110.

[35] P. Izzo, H. Qu, and S. M. Veres, "A stochastically verifiable autonomous control architecture with reasoning," in *CDC*. IEEE, 2016, pp. 4985–4991.

[36] B. Johnson and H. Kress-Gazit, "Analyzing and revising high-level robot behaviors under actuator error," in *IROS*. IEEE, 2013, pp. 741–748.

[37] A. Desai, T. Dreossi, and S. A. Seshia, "Combining model checking and runtime verification for safe robotics," in *RV*, ser. Lecture Notes in Computer Science, vol. 10548. Springer, 2017, pp. 172–189.

[38] M. Luckcuck, M. Farrell, L. A. Dennis, C. Dixon, and M. Fisher, "Formal specification and verification of autonomous robotic systems: A survey," *ACM Comput. Surv.*, vol. 52, no. 5, pp. 100:1–100:41, 2019.

[39] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Transactions on Computers*, vol. 100, no. 8, pp. 677–691, 1986.

[40] R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi, "Algebraic decision diagrams and their applications," *Formal Methods in System Design*, vol. 10, no. 2/3, pp. 171–206, 1997.

[41] P. Ashok, T. Brázdil, K. Chatterjee, J. Křetínský, C. H. Lampert, and V. Toman, "Strategy representation by decision trees with linear classifiers," in *QEST (1)*. Springer, 2019, pp. 109–128.

[42] T. Brázdil, K. Chatterjee, J. Kretínský, and V. Toman, "Strategy representation by decision trees in reactive synthesis," in *TACAS (1)*, ser. Lecture Notes in Computer Science, vol. 10805. Springer, 2018, pp. 385–407.

[43] L. D. Pyeatt and A. E. Howe, "Decision tree function approximation in reinforcement learning," Computer Science Department, Colorado State University, Tech. Rep., 1998.

[44] B. Lacerda, D. Parker, and N. Hawes, "Optimal and dynamic planning for markov decision processes with co-safe LTL specifications," in *IROS*. IEEE, 2014, pp. 1511–1516.

[45] ——, "Multi-objective policy generation for mobile robots under probabilistic time-bounded guarantees," in *ICAPS*. AAAI Press, 2017, pp. 504–512.

[46] M. Brambilla, A. Brutschy, M. Dorigo, and M. Birattari, "Property-driven design for robot swarms: A design method based on prescriptive modeling and model checking," *ACM Trans. Auton. Adapt. Syst.*, vol. 9, no. 4, pp. 17:1–17:28, 2015.

[47] S. Fuchs, S. Haddadin, M. Keller, S. Parusel, A. Kolb, and M. Suppa, “Cooperative bin-picking with Time-of-Flight camera and impedance controlled DLR lightweight robot III,” in *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Oct. 2010, pp. 4862–4867, iSSN: 2153-0866, 2153-0858, 2153-0858.

[48] K. Kleeberger, R. Bormann, W. Kraus, and M. F. Huber, “A survey on learning-based robotic grasping,” *Current Robotics Reports*, vol. 1, pp. 239–249, Dec. 2020. [Online]. Available: https://doi.org/10.1007/s43154-020-00021-6

[49] R. Matsumura, Y. Domae, W. Wan, and K. Harada, “Learning based robotic bin-picking for potentially tangled objects,” in *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2019, pp. 7990–7997.

[50] M. L. Puterman, *Markov Decision Processes: Discrete Stochastic Dynamic Programming*, 1st ed. USA: John Wiley & Sons, Inc., 1994.

[51] E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, Eds., *Handbook of Model Checking*. Springer, 2018.

[52] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri, “NUSMV: A new symbolic model checker,” *Int. J. Softw. Tools Technol. Transf.*, vol. 2, no. 4, pp. 410–425, 2000.

[53] G. Behrmann, A. David, K. G. Larsen, J. Håkansson, P. Pettersson, W. Yi, and M. Hendriks, “UPPAAL 4.0,” in *QEST*. IEEE Computer Society, 2006, pp. 125–126.

[54] C. Dehnert, S. Junges, J. Katoen, and M. Volk, “A storm is coming: A modern probabilistic model checker,” in *CAV (2)*, ser. Lecture Notes in Computer Science, vol. 10427. Springer, 2017, pp. 592–600.

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

[56] J. Wittmann, J. Jankowski, D. Wahrmann, and D. J. Rixen, “Hierarchical motion planning framework for manipulators in human-centered dynamic environments,” in *2020 29th IEEE International Conference on Robot and Human Interactive Communication (RO-MAN)*. IEEE, 2020.

[57] J. Wittmann, J. Kiesbye, D. J. Rixen, and U. Walter, “Supervised autonomous interaction in unknown territories - a concept for industrial applications in the near future,” in *52nd International Symposium on Robotics*, 2020.

[58] E. M. Hahn, Y. Li, S. Schewe, A. Turrini, and L. Zhang, “Iscasmc: A web-based probabilistic model checker,” in *FM 2014: Formal Methods: 19th International Symposium, Singapore, May 12-16, 2014. Proceedings*. Springer International Publishing, 2014, pp. 312–317.

[59] J. Bohren and S. Cousins, “The SMACH High-Level Executive,” *IEEE Robotics Automation Magazine*, vol. 17, no. 4, pp. 18–20, Dec. 2010.

[60] M. M. Khawaja, “Design and implementation of the autonomous task planner for a bin-picking robot,” Master’s thesis, Technical University of Munich, 2020.

D Model Checking for Proving and Improving Fault Tolerance of Satellites

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

Jonis Kiesbye, Kush Grover, and Jan Křetínský. “Model Checking for Proving and Improving Fault Tolerance of Satellites”. In: *2023 IEEE Aerospace Conference*. 2023, pp. 1–9. DOI: 10.1109/AERO55745.2023.10115801

Summary. Fault tolerance is a critical requirement for various systems, particularly in the context of space systems like satellites. Handling this is achieved through Fault Detection, Isolation and Recovery (FDIR) concepts. Fault isolation is a crucial component of FDIR, and in this paper, we reduced it to the reachability problem for MDPs which can then be solved by a probabilistic model checker to find an optimal strategy. However, the state space can grow exponentially with the number of variables, leading to a state-space explosion issue. To address this, we employ a method based on Monte Carlo Tree Search (MCTS) to trim the state space, retaining only significant decisions and states. This results in a much smaller MDP, for which we can quickly find an optimal policy. Ultimately, this policy is converted into a decision tree using dtControl as before.

In addition to fault isolation, we generate analysis reports that provide information about components that are not entirely isolable. This information can be used to improve the architecture during the development phase of a satellite. We built a comprehensive tool for these tasks, complete with a user-friendly GUI.

Contributions of thesis author. The author played a significant role in the development of the results presented in the paper and made valuable contributions to the manuscript. Noteworthy individual contributions are giving an algorithm by adapting Monte Carlo Tree Search to prune the model. Additionally, the author implemented the fault isolation part of the whole framework.

Model Checking for Proving and Improving Fault Tolerance of Satellites

Jonis Kiesbye
Chair of Astronautics
Technical University of Munich
85748 Garching, Germany
j.kiesbye@tum.de

Kush Grover
Chair for Theoretical Computer Science
Technical University of Munich
85748 Garching, Germany
kush.grover@tum.de

Jan Křetínský
Chair for Theoretical Computer Science
Technical University of Munich
85748 Garching, Germany
jan.kretinsky@tum.de

Abstract—**Developing the Fault Detection, Isolation & Recovery (FDIR) policy often happens late in the design phase of a spacecraft and might reveal significant gaps in the redundancy concept. We propose a process for continuously analyzing and improving the architecture of a spacecraft throughout the design phase to ensure successful fault isolation and recovery. The systems engineer provides a graph of the system's architecture containing the functional modes, the hardware components, and their dependency on each other as an input and gets back a weakness report listing the gaps in the redundancy concept. Overlaying the sub-graphs for every fault scenario allows us to reason about the feasibility of fault isolation and recovery. The graph is automatically converted to a Markov Decision Process for use with a model checker to generate a control policy for the FDIR process. The model is optimized by pruning inefficient branches with Monte Carlo Tree Search. We export this policy as a decision tree that ensures explainability, fast execution, and low memory requirements during runtime. We also generate C-code for fault isolation and reconfiguration that can be integrated in the FDIR software. The tool was used on system architectures created in the Modular ADCS project which is part of ESA's GSTP program. In this context, it helped to yield an effective redundancy concept with minimum overhead and dramatically reduce the programming effort for FDIR routines. Since we use model checking for the analysis, the designer gains formal verification of the robustness towards faults.**

TABLE OF CONTENTS

1. INTRODUCTION

In order to ensure reliability and performance of space systems, fault tolerance is not only desirable but necessary. Consequently, the methodology of *Fault Detection, Isolation and Recovery (FDIR)* has been integrated into the development process. Its purpose is twofold. Firstly, already during design time, it can give feedback on where more redundancy is needed to ensure better robustness against faults or where monitors are required to ensure faults can be diagnosed when they occur. Second, after the system is deployed, FDIR is running in the background, detecting possible faults, identifying faulty components, and deciding which redundancy to use to recover from the fault.

Due to the complexity of the systems, designing FDIR is a challenging task. As a result, several (semi-)automated approaches have been designed to analyse a model of the system and identify a policy (a.k.a. strategy) to follow, when a fault occurs. This involves choosing which actions should the system perform and to which modes to switch so that monitoring which behaviours are correct and which are faulty leads to identification of the faulty component. Notice that executing all possible functionalities would be wasteful if a subset is sufficient to isolate the fault. However, this paper goes even further: Also notice that choosing a minimal such subset (one we cannot reduce further) can still be extremely wasteful. Indeed, firstly, there may be more such subsets and the cost to execute them can be different (different actions/-modes consume different energy, time etc.). Secondly, the process of trying out is sequential and one can learn from the results so far which of the faults are still logically possible or even which are more probable. Depending on the current estimates, one can choose how to diagnose onwards so that the expected cost of the diagnosis is minimized.

Altogether, this paper tackles the problem of computing a *policy* (intuitively, a reactive recipe rather than a static plan, which gives a single sequence of actions) *minimizing the expected cost of diagnosis*. Our main idea is the following. Our approach reduces the problem to finding a policy in a *Markov Decision Process (MDP)* [1], which models the process of diagnosis. Since the MDP captures all the possible sequences of actions for any underlying fault, it is so huge that standard analysis techniques, such as those used in verification and reliable analysis of MDP [1], do not scale enough to solve the problem. Consequently, we apply the idea of *Monte Carlo Tree Search (MCTS)* ([2], [3], [4]), which has proved successful for very large systems, yielding near-optimal solutions. The essence is then partial exploration of the MDP (*tree-search*) guided by estimates of the expected costs from different situations; the latter is obtained by simulations (*roll-outs*).

2023 IEEE Aerospace Conference | 978-1-6654-9032-0/23/$31.00 ©2023 IEEE | DOI: 10.1109/AERO55745.2023.10115801

978-1-6654-9032-0/23/$31.00 ©2023 IEEE

Our contribution can be summarized as follows:

- We consider the problem of designing the cheapest diagnosis policy and reduce the problem to one in the domain of analysis of Markov decision processes.
- We design an algorithm based on Monte Carlo Tree Search to provide the policy as well as a procedure to check whether all faults are diagnosable and estimating the cost to do so.
- We implement a tool with graphical user interface allowing to model dependencies of components, the cost of their execution and the a-priori probabilities of faults, needed for calculating the policy with the cheapest expected cost. The policy is then exported as a decision tree, which is small and fast to execute, and can be integrated in the FDIR software.
- We demonstrate the efficacy of our approach on a case study of two satellite Attitude Determination and Control Systems (ADCS) that were studied in the Modular ADCS project in early 2022.

2. Related Work

The high reliability requirements of spacecraft necessitate a systematic approach towards reducing the probability of experiencing significant errors. Since in-orbit maintenance is prohibitively cost-intensive, the FDIR methods of the spacecraft shall keep it available in case of correctable faults. Common systematic approaches in the aerospace industry are Failure Modes and Effects Analysis (FMEA) [5] and Fault Tree Analysis (FTA).

FMEA utilizes knowledge of the mission engineers which conceive all possible fault modes and analyze their consequences in a bottom-up manner. As part of the 1996 Mars Global Surveyor mission, an alternative to FMEA, called Redundancy Verification Analysis [6], used a block diagram based approach to save analysis time in systems that have a redundant counterpart anyway. On the other hand, FTA is a top-down approach. Fault trees have been used in [7] and [8] to describe the architecture of a system in a similar but more extensive way than we do. They synthesize a recovery strategy that tries to optimize spare management, however, they assume knowledge of the basic events which include components where a fault has occurred. The goal of our approach is to figure out where exactly the fault occurred based on observables. [9] describes a model checking based technique to diagnose where a fault did occur based on monitoring a partially observable system at runtime. Their concept of coupled reachability is the same that we use for assessing fault isolability. The investigated models however use highly accurate continuous domain models in the background that are discretized by the Livingstone framework while we focus on the architecture of spacecraft in the design phase where no accurate simulation or prototype is available yet.

Approaches for analyzing the spacecraft and validating their FDIR methods use different kind of models. The FAME process [10] improves on FMEA by using Timed Fault Propagation Graphs. Here when a fault occurs, it propagates in the system, triggering several monitored nodes that can determine the source of the fault. The TASTE [11] middleware enables model-based software engineering for aerospace systems and can verify the behavior with the IF model checker. The engineers can specify the architecture of their system in the Architecture Analysis & Design Language. The more recent framework ERGO [12] targeting space robots builds on top of TASTE and brings an FDIR design and verification tool that utilizes the BIP tool set. Instead of manually defining the models that the individual frameworks need, it would be beneficial to utilize the system models from the systems engineering domain. [13] lays the foundation of translating the system architecture and behavior that is modeled in SysML for use in model checkers. This translator is used for code generation [14] and for a Model Checker as a Service [15] approach where the properties are verified by either the UPPAAL [16] or the Theta model checker.

Our approach focuses on early stages of spacecraft development where the architecture of the system has not been fixed yet and defining a detailed model would constrain the development too much. Several frameworks support the software engineering process by generating glue code or the whole software altogether. While state of the art approaches use model checking for verifying the properties of the systems, we use the model checker also for generating executable policies covering fault isolation and reconfiguration.

3. Preliminaries

For a set S, we use S^c to denote its complement and $\mathcal{D}(\mathcal{S})$ denotes the set of probability distributions over S.

Markov decision process—[1] A Markov Decision Process (MDP) is a widely used formalism to describe systems with probabilistic and non-deterministic behavior. Formally, it is defined as a 5-tuple (S, S_0, A, Δ, c) where S is a finite set of *states*, $S_0 \subseteq S$ is a set of initial states, A is a finite set of *actions*, $\Delta : S \times A \to D(S)$ is the *transition* function which describes how the system behaves upon choosing an action in a state, and $c : S \times A \times S \to \mathbb{R}$ is the *cost* function assigning a cost to each transition. An MDP can also be annotated with a subset of *target* states, which describes a set of desired states the system should eventually reach. A *policy* is a way of resolving non-determinism in the system by fixing an action to play from each state. An *optimal* policy is the one that optimizes some objective. In this paper, we will focus on optimal policies minimizing the expected total cost to reach a target set of states. This can model, for instance, the energy used until the diagnosis is finished.

Model checking—Model checking is a technique to verify whether a system satisfies a given specification. There are model checkers to verify non-deterministic, stochastic, timed or hybrid systems [17] [18] [16]. For MDPs, there are probabilistic model checkers like PRISM [17] and STORM [19] which can find an optimal policy for a given MDP and a reachability objective.

Decision tree—Decision trees are commonly used data structures for classification and regression in machine learning. They are well known for their interpretability because of their top-down binary tree like structure where each node contains a boolean predicate over input variables and every edge corresponds to either *true* or *false*. The leaf nodes contain the decisions and given a valuations of the input variables, a decision can be found by evaluating the predicate and taking corresponding edges down the tree.

The policies that model checkers output are stored in huge lookup tables whereas decision trees can store the same policies in a compact manner. dtControl [20] is a tool which can convert the lookup tables generated by model checkers into decision trees which are easier to understand and faster to execute.

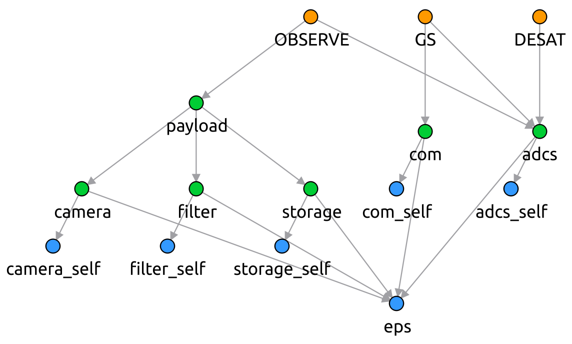


Figure 1. A satellite's architecture expressed as a graph

4. Methods

We assume that our system has n_C components, their set denoted by C, and n_M modes, their set denoted by M. Because of spare components, there can be several configurations for each mode, we denote them by $B_{m_1}, B_{m_2}, \cdots \subseteq C$ for a mode m. A cost function $C : M \to \mathbb{N}$ maps each mode to a number defining the cost to perform a check on it[2]. A fault-probability function $P : C \to [0, 1]$ maps each component to an a-priori probability of it failing. The whole system is then described by the tuple (G, P, C), where G is the architecture graph and P and C the functions above.

Spacecraft architecture representation

Similar to [21] and [7] but neglecting the temporal aspect, we represent the components, assemblies, and functional modes of the spacecraft as a directed graph. The modes describe high-level behavior of the vehicle, e.g. observing a star, desaturating the reaction wheels, keeping the satellite pointed at the ground station, etc. Modes form the roots of the graph. Edges run from the modes to the required components, which are the leaves of the graph. In between, the user can place intermediate nodes that we call assemblies.

We assume that it is easy to observe if the modes perform successfully or not. But every observation comes at a cost because the satellite will need time and energy to initialize the mode and evaluate whether it performs according to the expected behavior. We also assume that only the components (leaf nodes) contain the faults and thus cause the modes that depend on them to fail.

Note that this is not really a restriction because if an assembly contains a fault (apart from its sub-components), we can always add an auxiliary child node to represent that the assembly can be faulty by itself. Every component has a configurable fault probability. The exemplary graph in Figure 1 shows three modes in orange on the top level, six assemblies in green and six components in blue. All of camera, filter, storage, com, and adcs depend on the electrical power system (eps), which makes them an assembly by the definition laid out previously.

Every node depends on all its children. Looking at satellites, they usually have redundant assemblies. We include these two types:

[2]The cost can define how costly it is to perform a test on a mode by encoding the time it takes to run or by the amount of energy used.

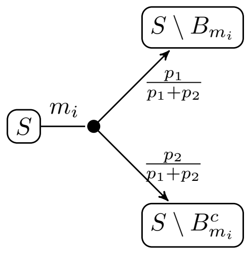


Figure 2. Transitions of the MDP

- OR: one of n children needs to work nominally.
- $\geq$: a specified amount of identical children needs to work nominally

Figure 6 shows the architecture of an ADCS (Attitude Determination & Control System). The star tracker and gyroscope are OR assemblies, the thrusters form a $\geq$ assembly.

Isolating faults

Whenever a fault in one of the components occurs, it will cause all modes depending on it to fail. Building on this premise, we can cycle through multiple modes and thus derive which component is faulty. Given the case that a component still responds to commands but produces skewed data, our method can still isolate the faulty component.

When a mode fails, we label all components that the mode depends on as suspects. Next, we will execute a different mode that depends on a subset of the suspects. If that mode succeeds, we can remove the components the mode depended on from the set of suspects. If the mode fails, we can remove the components that were not required for the mode. By repeating this process, we reach the minimal set of suspects which cannot be reduced anymore. If the minimal set has only one component, the fault has been isolated completely.

Knowing the architecture of a system, we can derive whether all faults can be isolated. For every component to be isolable, there must exist a superposition of modes that depends on all other components except this one. Formally,

Definition 1. *For a given set of suspects S, a component $c \in S$ is isolable if there exists mode configurations $x_1, x_2 \dots$ which contain c and $y_1, y_2 \dots$ which does not contain c such that $(S \bigcap_j B_{y_j}) \setminus \bigcup_i B_{x_i} = c$.*

We transform the architecture graph G into an MDP $\mathcal{M}$ where we keep track of the suspicious components. States of $\mathcal{M}$ are sets of suspects and the initial states are sets of components corresponding to different configurations of each mode. Therefore for every mode x, we mark all configurations B_{m_i} as initial states. The set of actions are also all mode configurations and executing an action means to test whether that configuration works or not. The transitions encode changes in the set of suspects according to whether the mode works accurately or not as shown in Figure 2. Here $p_1 = 1 - \prod_{c \in S \setminus B_{m_i}} (1 - P(c))$ which is the probability that a component in $S \setminus B_{m_i}$ fails and similarly $p_2 = 1 - \prod_{c \in S \setminus B_{m_i}^c} (1 - P(c))$.

For example, for the architecture graph shown in Figure 1, the states of the MDP would be vectors of length 6 since there are 6 components (we are using vectors to represent sets here).

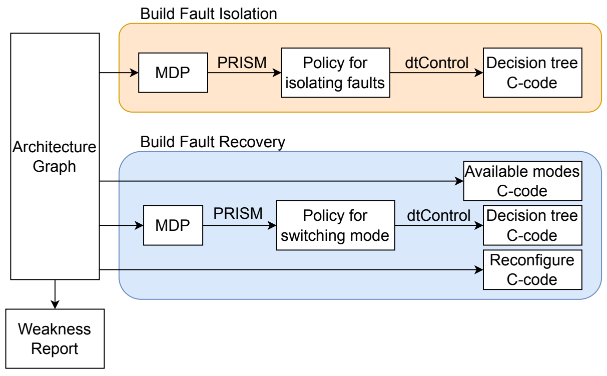


Figure 3. Artifacts generated for fault isolation and recovery

An initial state corresponding to a fault that occured while executing the ground station tracking mode (GS), would be $S = (0, 0, 0, 1, 1, 1)$ expressing that the components 4 (com_self), 5 (adcs_self), and 6 (eps) are suspicious. Since actions are synonymous with modes, an action corresponding to DESAT would require the components $B_{DESAT} = (0, 0, 0, 0, 1, 1)$. As shown in Figure 2, on executing DESAT from S, you either go to $S \setminus B_{DESAT} = (0, 0, 0, 1, 0, 0)$ if DESAT is working or you go to $S \setminus B^c_{DESAT} = (0, 0, 0, 0, 1, 1)$ otherwise.

Target states are the sets of suspects which cannot be reduced further by testing any mode configuration.

Since we need to feed this MDP to a model checker, the size of the input is dependent on number of transitions. As the size increases, parsing the input can become a problem for the model checkers. To tackle this problem, we suggest a simulation-based pruning of the MDP, inspired by the MCTS algorithms.

Pruning based on Monte Carlo Tree Search

We build a smaller MDP iteratively by pruning the useless parts on the fly. Pseudocode for this is shown in Algorithm 1 which can be broken down into four simple steps. It starts with only initial states as the set of states and line 5 picks a new state to expand (step 1), for this we can randomly select a state that has never been picked before. In line 6, it expands the selected state by adding all of its successors to the MDP (step 2). Line 7 corresponds to doing simulations starting from all of these newly added states to get an estimate of expected cost required to isolate the fault from that state (step 3). In line 8, all actions except the best ones are pruned (step 4). The best actions are based on the expected cost we get from simulations. Number of actions to keep (a) for each state is a parameter of the procedure and is fixed beforehand. This procedure terminates when all the states have been expanded and it does so in finite time because the number of states are bounded by a finite number.

We can now perform our analysis on this smaller MDP and figure out the best policy to isolate the fault. One way to get a policy is to pick the best action based on the expected cost that we got from the simulations. It is also possible to find the optimal policy for the smaller MDP. As described in section 3, PRISM is one such tool that can be used for this and dtControl can convert the lookup table into a decision tree to generate an efficient executable.

Algorithm 1 Monte-Carlo Pruning

```
1:  Input: $G, S_0, a$
2:  $\mathcal{M} \leftarrow S_0$
3:  $V \leftarrow \emptyset$
4:  while $V \neq S$ do
5:      $s \leftarrow \texttt{Select}(S \setminus V)$
6:      $\mathcal{M} \leftarrow \texttt{Expand}(s)$
7:      $\texttt{Simulate}(G, s)$
8:      $\texttt{Prune}(\mathcal{M}, s, a)$
9:      $V \leftarrow V \cup s$
10: end while
11: Output: $\mathcal{M}$
```

Recovering from faults

If the redundant assemblies in the system allow for a mode to not depend on the faulty component, we can reconfigure to keep the mode available. If all modes remain available for every single component fault, the system is fault-tolerant.

Definition 2. *An architecture graph is called fault-tolerant if for all mode configurations B_{m_i}, for every $c \in B_{m_i}$, there exists a mode configuration B_{m_j} of the same mode such that $c \notin B_{m_j}$.*

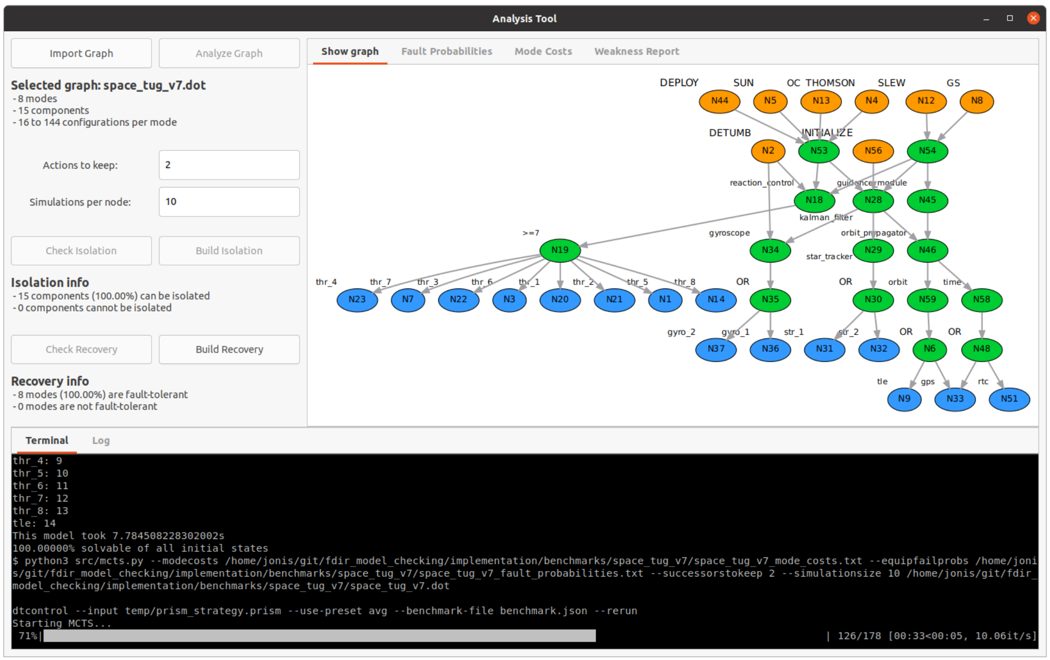
Generating code—The recovery executable reads the desired mode and the equipment state, a vector of Booleans describing whether each component is available or faulty. Based on the architecture described in the graph, the equipment state is translated to a list of available modes. The available modes, the desired mode, and optionally the dynamical state of the satellite, e.g. the angular rate or the battery level, are interpreted by a policy that decides which mode to execute. A PRISM model that is automatically generated and adaptable by the user defines this policy. The tool dtControl converts the policy to a compact decision tree. Based on the selected mode and the equipment state, we eventually decide what equipment is used and how to configure the redundant assemblies.

As shown in Figure 3, determining the available modes and reconfiguring the system is handled by C-code functions that are directly generated by the graph analysis tool. The mode switching is implemented in a C-code version of the decision tree generated by dtControl.

Iterative improvement

The architecture graph can be easily derived from the architecture model used for a Model-Based Systems Engineering (MBSE) process. The engineer can feed the current state of the satellite model to the algorithms described before and will get an analysis of the fault isolability and tolerance pointing out possible weaknesses. Based on this feedback, the engineer can improve the satellite's architecture and feed an updated graph to the tool for validating it.

Instead of solely relying on the expertise of the systems engineers for designing a fault-tolerant satellite and implementing the FDIR algorithms late in the development cycle, our tool enables a systematic approach that supports the engineer from phase B on. Figure 5 shows the iterative design cycle where the engineer derives a graph from the system architecture and improves the system architecture using the hints and metrics from the weakness report.



Selected graph: space_tug_v7.dot
- 8 modes
- 15 components
- 16 to 144 configurations per mode

Actions to keep: 2

Simulations per node: 10

Isolation info
- 15 components (100.00%) can be isolated
- 0 components cannot be isolated

Recovery info
- 8 modes (100.00%) are fault-tolerant
- 0 modes are not fault-tolerant

```
thr_4: 9
thr_5: 10
thr_6: 11
thr_7: 12
thr_8: 13
tle: 14
This model took 7.784508228302002s
100.00000% solvable of all initial states
$ python3 src/mcts.py --modecosts /home/jonis/git/fdir_model_checking/implementation/benchmarks/space_tug_v7/space_tug_v7_mode_costs.txt --equipfailprobs /home/joni
s/git/fdir_model_checking/implementation/benchmarks/space_tug_v7/space_tug_v7_fault_probabilities.txt --successorstokeep 2 --simulationsize 10 /home/jonis/git/fdir_
model_checking/implementation/benchmarks/space_tug_v7/space_tug_v7.dot

dtcontrol --input temp/prism_strategy.prism --use-preset avg --benchmark-file benchmark.json --rerun
Starting MCTS...
 71%|                                                                          | 126/178 [00:33<00:05, 10.06it/s]
```

Figure 4. The graphical interface integrates all algorithms of our tool

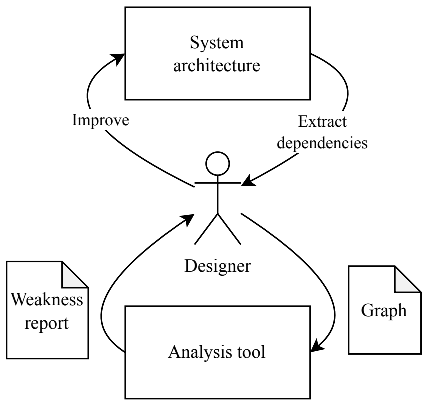


Figure 5. The engineer supplies the architecture graph to our tool and updates it aided by the weakness report

Graphical user interface

The algorithms for graph analysis and code generation can be called from a graphical user interface (GUI)[3]. The application is built with Python 3 and GTK and interfaces with the PRISM model checker and dtControl so it can run on any Linux PC. Notable third-party libraries include xdot and NetworkX. The engineer can execute the steps mentioned earlier in the GUI and gets visual and textual feedback on the model's performance. Figure 4 shows the GUI after loading and analyzing a graph. It visualizes the loaded graph, allows for editing the mode costs and component fault probabilities, displays the weakness report, and shows the log messages of the underlying algorithms.

[3]The source code of the GUI and the underlaying algorithms is available online at https://github.com/JonisK/build_and_improve_fdir

5. APPLICATION

In this section, we will use the GUI for analyzing a simplified space tug and generating code for it. In addition to the original and improved space tug, we also modeled the ADCS of an Earth observation satellite relying on magnetorquers and reaction wheels for actuation. This additional system was included for characterizing our tool's performance on a larger graph. Both spacecraft architectures stem from the Modular ADCS project carried out as part of the ESA General Support Technology Programme (GSTP) program.

Modeled spacecraft

An exemplary space tug shall deploy payloads to different orbits after separating from the launch vehicle. We focus on the thruster-based ADCS of the space tug.

Figure 6 shows the modes and components that can be derived from e.g. the block definition diagram that is available in early phases of the development cycle already.

The system was designed with single-fault tolerance in mind so it uses a set of eight thrusters of which at least 7 need to operate for 3-axis reaction control [22]. The gyroscope and star tracker are added in redundant pairs. Only one of each needs to stay available for nominal attitude determination performance. An orbit propagator feeds the reference models of the Kalman filter with the spacecraft's position and velocity. The orbit propagator needs to know the current time and the orbital parameters. A GPS sensor can provide both time and, through an estimation algorithm, the orbit parameters. If the GPS becomes unavailable, the time can be read from an on-board real-time clock.

The satellite has a detumbling mode (DETUMB) for reducing the angular rate. A THOMSON spin mode points one face of the satellite towards Earth, the SUN tracking mode

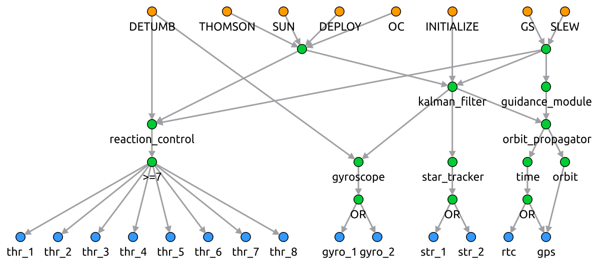


Figure 6. Architecture of the space tug's ADCS

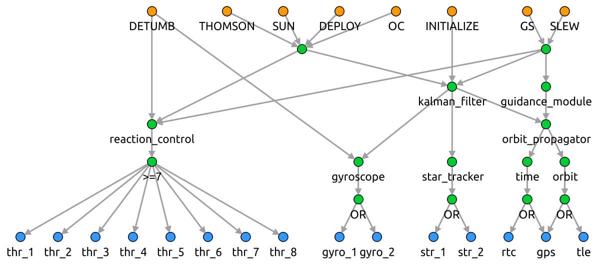


Figure 7. Improved architecture including a redundant assembly for determining orbit parameters

maximizes the power input on the solar panels, the DEPLOY mode maneuvers to the desired attitude for separation from a payload, and orbit control (OC) keeps the attitude stable while the apogee motor fires. The modes THOMSON, SUN, DEPLOY, and OC all depend on the same components so they are equivalent with regards to the isolation process which is based on overlaying different modes to confirm whether a component has a detrimental effect on the ADCS performance. Since there are multiple redundant assemblies that do not need all children to be available, every mode can be split up into a multitude of configurations that cover all feasible permutations of redundant configurations.

Analyzing this architecture with the algorithms laid out in section 4, reveals that all components except for one are isolable and only one mode stays available in case of a single fault. The weakness report shown in Figure 8 shows the gaps and gives the fault probability of every mode.

From the information regarding the isolability, the engineer can easily deduce that the GPS is a single point of error. By adding a second GPS and forming a redundant assembly with the original GPS receiver, one can achieve perfect isolability and single-fault tolerance. Alternatively, the engineer might look for a cheaper backup to the redundant GPS and use orbital parameters determined by ground radar and uploaded as two-line element (TLE) for feeding the orbit propagator.

Figure 7 shows the added TLE node. When analyzing the architecture including the TLE, the weakness report in Figure 9 does not show any gaps anymore. The compound fault probabilities of every mode have also improved relative to the original architecture. The only exception is the DETUMB mode which is not dependent on the orbit propagator.

After successfully resolving the gap, the engineer has the option extend the analytical effort by including underlying components, e.g. the EPS fuses or data buses, in the model and verify that the ADCS remains fault tolerant. Once the

The graph space_tug_v6.dot shows the following weaknesses:
- Component gps is not isolable
- Mode INITIALIZE is not single-fault tolerant
- Mode THOMSON is not single-fault tolerant
- Mode DEPLOY is not single-fault tolerant
- Mode OC is not single-fault tolerant
- Mode SLEW is not single-fault tolerant
- Mode GS is not single-fault tolerant
- Mode SUN is not single-fault tolerant

Assuming the component fault probabilities defined in space_tug_v6_fault_probabilities.txt, the modes have these fault probabilities:
- The fault probability for mode SLEW is 2.72 %
- The fault probability for mode GS is 2.72 %
- The fault probability for mode THOMSON is 2.23 %
- The fault probability for mode DEPLOY is 2.23 %
- The fault probability for mode OC is 2.23 %
- The fault probability for mode SUN is 2.23 %
- The fault probability for mode DETUMB is 1.74 %
- The fault probability for mode INITIALIZE is 0.500 %

Figure 8. Weakness report for the space tug graph

The graph space_tug_v7.dot shows no weaknesses.
Assuming the component fault probabilities defined in 'space_tug_v7_fault_probabilities.txt', the modes have these fault probabilities:
- The fault probability for mode SLEW is 1.74 %
- The fault probability for mode GS is 1.74 %
- The fault probability for mode OC is 1.74 %
- The fault probability for mode THOMSON is 1.74 %
- The fault probability for mode DEPLOY is 1.74 %
- The fault probability for mode SUN is 1.74 %
- The fault probability for mode DETUMB is 1.74 %
- The fault probability for mode INITIALIZE is 0.000750 %

Figure 9. Weakness report for the improved space tug graph

architecture has evolved, the verification and improvement cycle can be repeated.

Code generation

For fault isolation, the MCTS policy is compressed by dt-Control into a decision tree. Reading the branches of the decision tree from top to bottom, each node asks whether a specific component is still suspicious or has been verified as available already. The leaf nodes contain a command on what mode to execute next. The number after the mode name gives an identifier for the configuration of the redundant assemblies. After determining whether the commanded mode was successful, one can update the component state vector. As soon as all components are marked available except for one, the faulty component has been found and the recovery can start.

The fault recovery code evaluates the availability of modes, decides which mode to execute and which components to activate. The decision on the appropriate mode is made by evaluating a policy generated with the PRISM model checker. The PRISM model for choosing the mode is auto-generated by our tool with a basic policy in mind that whenever the desired mode is available, it will get selected. If a mode is unavailable, a fallback mode, e.g. safe mode, detumbling, or off, is activated and the system waits for ground commands on how to proceed. Environmental states can be considered in the mode selection, e.g. if the angular rate passes a critical threshold, DETUMB mode is selected no matter what the desired mode is. To give an example, the decision tree for the mode switcher of the Modular ADCS project is shown in Figure 10.

The PRISM language allows to define more state variables and can move via intermediate steps to the goal condition, similar to a high-level task planner. We could for example require the model to first select SLEW mode before initiating the GS tracking mode. Going further, we could describe a protective cover for an observation instrument with the states open and closed, prompting the spacecraft to open the cover before starting an observation and closing it afterwards. The use of PRISM-generated policies for task planning is laid out in [23]. Once the user has modified the auto-generated PRISM model to suit their mission, the policy is generated and translated to a decision tree with dtControl. PRISM evaluates the decision-making logic for all conceivable initial states and reports whether the model covers all of them.

6. Results

We show our results on two case studies, space tug and earth observation. We used two versions of the space tug benchmark, the original version (Figure 6) and the improved version (Figure 7). First part of Table 3 shows the size of these benchmarks.

While analyzing an architecture regarding fault isolation and recovery and generating the weakness report from that data takes less than 10 seconds as shown in Table 1, the process of generating code is slows down with increasing model complexity. Having more nodes in the graph and keeping more actions while pruning leads to high execution times for model checking and decision tree generation.

The size of the MDPs generated for fault isolation is shown in Table 2. The number of transitions in the original MDPs being rather high shows the need of pruning. We show size of pruned MDPs for two values of a (actions to keep).

Table 3 shows the expected costs to isolate a fault using policies generated by three approaches[4]. First approach is a naive one, that randomly picks a mode which can reduce the set of suspects. The second one picks the best action according to the simulations done during pruning of the model, this corresponds to pruning all the action from each state except the best one. The third policy is the optimal one for the pruned MDP that we get from PRISM. Compared to policies obtained in standard ways where the costs and probabilities are not reflected, we can save more than a half of the expected cost already by using our MCTS policy. This can be further improved by using the PRISM policy at the expense of higher computational time to compute (offline) the policy. Observe the trade-off between the expected cost and the time as the value of a is increased. Note that the table does not list the expected cost for the original whole MDP using PRISM because the MDP was too big to parse by PRISM, which returned an out-of-memory error.

Figure 10 shows the control policy of the mode switcher for the improved space tug. One can observe how the environmental variables influence the selected mode. Whenever the selected mode is unavailable, the fall-back option `off` activates which will wait for input from ground. The set of initial states is 248,832, of which 100 % are covered by the mode switcher, guaranteeing that the fault recovery executable will never get stuck during satellite operations.

7. Conclusions

Our tool detects deficiencies in a spacecraft's architecture from the early design stages on and supports the engineer in improving its isolability and fault tolerance, thereby reducing the risk of late design changes. Its model-based approach that relies on the engineer for translating the architecture makes it less powerful than model-driven software engineering frameworks like TASTE. On the other hand this very characteristic allows our tool to be easily integrated in dynamic and iterative design approaches which are often found in the CubeSat and SmallSat sector. We show how the utilization of a model checker allows us not only to verify the specifications of a system but also synthesize controllers and generate C-code that has low computational requirements for execution.

In the future, we intend to automatically translate architectures from SysML to enable integrating our tool with MBSE frameworks and thereby reduce the time an engineer needs to receive feedback on the fault handling abilities of their spacecraft. Further on, we will extend both the analysis algorithms and the isolation executable to handle the combination of two or three faults in the system. While small satellites rarely require multiple fault tolerance, determining fault combinations will typically hint at unmodeled common causes, e.g. a failing data bus, and thereby indicate whether the current architecture model is sufficient for isolating the faulty component or not.

	generate weakness report	**generate fault isolation**		**generate fault recovery**
		$a = 2$	$a = 10$	
space tug	3.46 s	24.33 s	33.08 s	15.14 s
space tug improved	5.54 s	65.70 s	104.20 s	17.38 s
earth observation	4.97 s	164.78 s	295.12 s	113.48 s

Table 1. Execution time of the analysis and code generation steps

		space tug	**space tug improved**	**earth observation**
original	st.	9,215	18,431	7,775
	tr.	454,834	1,325,232	687,974
$a = 2$	st.	453	635	874
	tr.	1,410	2020	2,784
$a = 10$	st.	962	1,150	1,954
	tr.	11,452	13,026	14,644

Table 2. Number of states and transitions in the original vs pruned MDPs

Acknowledgments

This work has been supported by the DFG research training group GRK 2428 Continuous Verification of Cyber-Physical Systems (ConVeY).

References

[1] Baier, C. & Katoen, J. Principles of Model Checking. (2008,1)

[4]The expected costs are calculated statistically, so there can be slight differences in the values while reproducing these results.

Benchmark	Size		Isolating Faults						
	n_c	mode configs	Naive	MCTS		PRISM $a = 2$		PRISM $a = 10$	
			cost	cost	time	cost	time	cost	time
space tug	14	98	783.8	245.7	136.0 s	244.9	123.8 s	243.6	163.1 s
space tug improved	15	178	865.2	246.2	532.0 s	246.5	389.2 s	245.0	433.6 s
earth observation	22	410	2308.3	1136.7	1475.0 s	1126.1	1528.6 s	1114.5	1780.7 s

Table 3. Size of inputs, expected cost of isolating a fault and time taken to find fault isolation policy

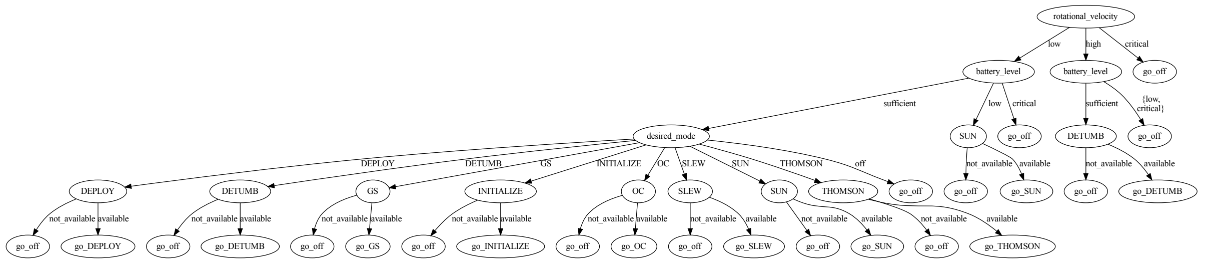


Figure 10. Decision tree of the mode switching logic

[2] Abramson, B. & Korf, R. A Model of Two-Player Evaluation Functions. (1987,1)

[3] Kocsis, L. & Szepesvári, C. Bandit Based Monte-Carlo Planning. *Machine Learning: ECML 2006*. pp. 282-293 (2006)

[4] Coulom, R. Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. *Computers And Games*. pp. 72-83 (2007)

[5] Dhillon, B. Failure modes and effects analysis — Bibliography. *Microelectronics Reliability*. 32 (1992), https://doi.org/10.1016/0026-2714(92)90630-4

[6] Sincell, J., Perez, R., Noone, P. & Oberhettinger, D. Redundancy verification analysis-an alternative to FMEA for low cost missions. *Annual Reliability And Maintainability Symposium. 1998 Proceedings. International Symposium On Product Quality And Integrity*. pp. 54-60 (1998)

[7] Müller, S., Gerndt, A. & Noll, T. Synthesizing FDIR Recovery Strategies from Non-Deterministic Dynamic Fault Trees. *AIAA SPACE And Astronautics Forum And Exposition*., https://arc.aiaa.org/doi/abs/10.2514/6.2017-5163

[8] Müller, S., Mikaelyan, L., Gerndt, A. & Noll, T. Synthesizing and optimizing FDIR recovery strategies from fault trees. *Science Of Computer Programming*. 196 pp. 102478 (2020), https://doi.org/10.1016/j.scico.2020.102478

[9] Cimatti, A., Pecheur, C. & Cavada, R. Formal Verification of Diagnosability via Symbolic Model Checking. *Proceedings Of The 18th International Joint Conference On Artificial Intelligence*. pp. 363-369 (2003)

[10] Bittner, B., Bozzano, M., Cimatti, A., De Ferluc, R., Gario, M., Guiotto, A. & Yushtein, Y. An Integrated Process for FDIR Design in Aerospace. *Model-Based Safety And Assessment*. pp. 82-95 (2014)

[11] Perrotin, A., Conquet, E., Dissaux, P., Tsiodras, T. & Hugues, J. The TASTE Toolset: turning human designed heterogeneous systems into computer built homogeneous software. *ERTS2 2010, Embedded Real Time Software & Systems*. (2010)

[12] Ocón, J., Colmenero, F., Estremera, J., Buckley, K., Alonso, M., Heredia, E., Garcia, J., Coles, A., Coles, A., Munoz, M. & Others The ERGO framework and its use in planetary/orbital scenarios. *Proceedings Of The 69th International Astronautical Congress (IAC)*. (2018)

[13] Wagner, D., Bennett, M., Karban, R., Rouquette, N., Jenkins, S. & Ingham, M. An ontology for State Analysis: Formalizing the mapping to SysML. *2012 IEEE Aerospace Conference*. pp. 1-16 (2012)

[14] Godart, P., Gross, J., Mukherjee, R. & Ubellacker, W. Generating real-time robotics control software from sysml. *2017 IEEE Aerospace Conference*. pp. 1-11 (2017)

[15] Horváth, B., Graics, B., Hajdu, Á., Micskei, Z., Molnár, V., Ráth, I., Andolfato, L., Gomes, I. & Karban, R. Model checking as a service: towards pragmatic hidden formal methods. *Proceedings Of The 23rd ACM/IEEE International Conference On Model Driven Engineering Languages And Systems: Companion Proceedings*. pp. 1-5 (2020)

[16] Behrmann, G., David, A., Larsen, K., Håkansson, J., Pettersson, P., Yi, W. & Hendriks, M. Uppaal 4.0. *Third International Conference On The Quantitative Evaluation Of Systems, QEST 2006*. pp. 125-126 (2006,1)

[17] Kwiatkowska, M., Norman, G. & Parker, D. PRISM 4.0: Verification of Probabilistic Real-time Systems. *Proc. 23rd International Conference On Computer Aided Verification (CAV'11)*. 6806 pp. 585-591 (2011)

[18] Cimatti, A., Clarke, E., Giunchiglia, F. & Roveri, M. NuSMV: A New Symbolic Model Verifier. *Computer Aided Verification*. pp. 495-499 (1999)

[19] Hensel, C., Junges, S., Katoen, J., Quatmann, T. & Volk, M. The probabilistic model checker Storm. *International Journal On Software Tools For Technology Transfer*. 24 pp. 1-22 (2022,8)

[20] Ashok, P., Jackermeier, M., Kretinsky, J., Weinhuber, C., Weininger, M. & Yadav, M. dtControl 2.0: Explainable Strategy Representation via Decision Tree Learning Steered by Experts. *Tools And Algorithms For The Construction And Analysis Of Systems - 27th International Conference, TACAS 2021, Held As Part Of The European Joint Conferences On Theory And Practice Of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part II*. 12652 pp. 326-345 (2021), https://doi.org/10.1007/978-3-030-72013-1_17

[21] Abdelwahed, S., Karsai, G. & Biswas, G. System diagnosis using hybrid failure propagation graphs. *The 15th International Workshop On Principles Of Diagnosis*. (2004)

[22] Pasand, M., Hassani, A. & Ghorbani, M. A study of spacecraft reaction thruster configurations for attitude control system. *IEEE Aerospace And Electronic Systems Magazine*. 32, 22-39 (2017)

[23] Kiesbye, J., Grover, K., Ashok, P. & Křetínský, J. Planning via model checking with decision-tree controllers. *2022 International Conference On Robotics And Automation (ICRA)*. pp. 4347-4354 (2022)

BIOGRAPHY

Jonis Kiesbye *received his Master's degree at Technical University of Munich (TUM) and participated on the student-driven CubeSat mission MOVE-II that launched on Dec 3rd 2018. Working as a research assistant at TUM's Chair of Astronautics in the domains task planning, robotics, and simulative verification since 2018.*

Kush Grover *completed his Masters in Computer Science at Chennai Mathematical Institute before joining as a PhD student at Technical University of Munich under supervision of Prof. Jan Křetínský. His area of research includes controller synthesis and verification of stochastic system.*

Jan Křetínský *is a professor for Formal Methods for Software Reliability at the Technical University of Munich. He works in the area of verification, controller synthesis, automata theory, and their intersections with machine learning.*