BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

DECISION TREES FOR MULTI-ENVIRONMENT MARKOV DECISION PROCESSES

ROZHODOVACÍ STROMY PRO RODINY MARKOVSKÝCH ROZHODOVACÍCH PROCESŮ

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR AUTOR PRÁCE	**Bc. LADISLAV DOKOUPIL**
SUPERVISOR VEDOUCÍ PRÁCE	**doc. RNDr. MILAN ČEŠKA, Ph.D.**

BRNO 2025

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Master's Thesis Assignment

162927

Institut:	Department of Intelligent Systems (DITS)
Student:	**Dokoupil Ladislav, Bc.**
Programme:	Information Technology and Artificial Intelligence
Specialization:	Mathematical Methods
Title:	**Decision trees for multi-environment Markov Decision Processes**
Category:	Formal Verification
Academic year:	2024/25

Assignment:

1. Study the state-of-the-art approaches for solving multi-environment Markov Decision Processes (MDPs), and for construction of decision trees representing MDP polices.
2. Investigate effective representations of polices for multi-environment MDPs with the focus on decision trees.
3. Design and implement a method synthetizing effective representations of winning polices in multi-environment MDPs.
4. Perform a detailed experimental evaluation of the proposed method on practically relevant planning problems.

Literature:

- Kochenderfer, M.J., Wheeler, T.A., and Wray K.H, Algorithms for Decision Making, MIT Press 2022.
- Andriushchenko, R., Češka, M., Junges, S., Katoen, J.P. and Stupinský, Š. PAYNT: A Tool for Inductive Synthesis of Probabilistic Programs. In *CAV 2021*.
- Junges, S., N. Jansen, R. Wimmer, T. Quatmann, L. Winterer, J. P. Katoen, and B. Becker. Finite-state controllers of POMDPs using parameter synthesis. In *UAI 2018.*
- Andriushchenko, Roman, Milan Češka, Sebastian Junges, and Filip Macák. ‘Policies Grow on Trees: Model Checking Families of MDPs’. In Automated Technology for Verification and Analysis, 2024.
- Ashok, P., Jackermeier, M., Křetínský, J., Weinhuber, C., Weininger, M., Yadav, M.: DtControl 2.0: Explainable strategy representation via decision tree learning steered by experts. In: TACAS 2021.

Requirements for the semestral defence:
Items 1, 2 and partially 3.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Češka Milan, doc. RNDr., Ph.D.**
Consultant:	Andriushchenko Roman, Ing.
Head of Department:	Kočí Radek, Ing., Ph.D.
Beginning of work:	1.11.2024
Submission deadline:	21.5.2025
Approval date:	31.10.2024

Abstract

Markov Decision Processes (MDPs) provide a fundamental framework for sequential decision making under uncertainty. However, practical systems often involve structural variations, best modeled by families of MDPs (f-MDPs). The challenge lies in synthesizing compact and interpretable policies covering the entire family, as existing approaches can be overly conservative, yielding large, difficult-to-interpret policies. This thesis presents novel techniques to address these issues. We propose two key orthogonal contributions: (1) a heuristic-based pruning approach to generate minimal controllers from the initial policies, and (2) a novel transformation that reduces family to an MDP problem, by leveraging a game abstraction to model worst-case system variations. Such transformation allows us to synthesize alternative policies using existing synthesis tools, while maintaining correctness. We further introduce a unified decision tree representation, comprising a candidate mapping tree and a unified policy tree. This structure provides a globally compact and interpretable solution for the entire f-MDP, minimizing redundancy compared to managing separate, often tabular, controllers. Implemented as an extension to the PAYNT tool, our methods are experimentally shown to substantially reduce controller size and improve interpretability, often with an acceptable trade-off in synthesis time.

Abstrakt

Markovské rozhodovací procesy (MDP) poskytují základní rámec pro sekvenční rozhodování v podmínkách nejistoty. Praktické systémy však často zahrnují strukturální variace, které nejlépe modelují rodiny MDP (f-MDP). Výzvou je syntéza kompaktních a interpretovatelných kontrolerů pokrývajících celou rodinu, jelikož existující přístupy mohou být příliš konzervativní a vést k rozsáhlým, obtížně interpretovatelným kontrolerům. Tato práce představuje nové techniky k řešení těchto problémů. Navrhujeme dva klíčové, ortogonální přínosy: (1) heuristický přístup pro generování minimálních kontrolerů z původních kontrolerů a (2) novou transformaci, která redukuje problém rodiny MDP na problém klasického MDP pomocí herní abstrakce, jež modeluje nejhorší možné systémové variace. Tato transformace umožňuje syntetizovat alternativní kontrolery s využitím stávajících nástrojů pro syntézu, a to při zachování korektnosti. Dále zavádíme sjednocenou reprezentaci pomocí rozhodovacích stromů, zahrnující strom mapování kandidátů a sjednocený strom kontrolerů. Tato struktura poskytuje globálně kompaktní a interpretovatelné řešení pro celou f-MDP, přičemž minimalizuje redundanci oproti správě samostatných, často tabulárních kontrolerů. Naše metody, implementované jako rozšíření nástroje PAYNT, experimentálně prokazují výrazné zmenšení velikosti kontrolerů a zlepšení jejich interpretovatelnosti, často s přijatelným kompromisem v čase syntézy.

Keywords

Markov Decision Processes, Families of MDPs, Decision Trees, Compact Controllers, Interpretable Controllers, Policy Optimization, Stochastic Games

Klíčová slova

Markovovy rozhodovací procesy, Rodiny MDP, Rozhodovací stromy, Kompaktní kontrolery, Interpretovatelné kontrolery, Optimalizace kontrolerů, Stochastické hry

Reference

DOKOUPIL, Ladislav. *Decision trees for multi-environment Markov Decision Processes.* Brno, 2025. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor doc. RNDr. Milan Češka, Ph.D.

Rozšířený abstrakt

Rodiny Markovských rozhodovacích procesů (f-MDP) představují klíčový formalismus pro modelování sekvenčního rozhodování v systémech s parametrickou nejistotou nebo variabilními provozními podmínkami, jaké se bežně vyskytují v praktických aplikacích, jako je robotika či autonomní systémy. Syntéza kontrolerů pro tyto systémy vyžaduje nejen zaručení robustnosti vůči specifikovaným cílům, ale také dosažení kompaktnosti a interpretovatelnosti výsledných řešení. Právě tato kombinace představuje zásadní výzvu, kterou tato práce adresuje.

Současné přistupy pro syntézu kontrolerů pro f-MDP, zejména ty založené na herních abstrakcích, sice umožňují generovat robustní kontrolery, avšak často za cenu značných kompromisů. Takto vytvořené kontrolery bývají typicky konzervativní, neboť jsou navrhovány proti nejhorší možné variaci prostředí, což může vést k zahrnutí mnoha nadbytečných stavů či akcí. Navíc jsou často reprezentovány v tabulární formě, která není pamětově efektivní a je obtížně interpretovatelná pro člověka. Dále může správa samostatných kontrolerů pro jednotlivé podrodiny vést k redundanci, pokud se podobná logika opakuje napříč více instancemi.

K překonání těchto omezení tato práce navrhuje sadu nových technik a reprezentačních strategií. Prvním klíčovým přínosem je vývoj heuristických algoritmů pro systematické prořezávání nadbytečných stavů a akcí z počátečních robustních kontrolerů. Tento přístup přímo redukuje jejich konzervatismus a složitost, při plném zachování korektnosti a robustnosti vůči celé rodině f-MDP.

Druhým významným přínosem je inovativní transformace problému f-MDP na odvozené MDP. Tato metoda transformuje vyřešenou herní abstrakci, kde je strategie prostředí fixována na ekvivalentní MDP. To následně umožňuje aplikaci pokročilých nástrojů pro syntézu MDP kontrolerů, jako je dtNESt, za účelem generování alternativních kontrolerů, které jsou kompaktnější a snáze reprezentovatelné, přičemž si zachovávají robustnost pro celou původní rodinu f-MDP.

Třetím klíčovým přínosem je zavedení nové, sjednocené reprezentace kontroleru ve formě *sjednoceného rozhodovacího stromu (UDT)*. Tato struktura, skládající se ze *stromu mapování kandidátů (CMT)* a *sjednoceného stromu kontrolerů (UPT)*, poskytuje globálně optimalizované a interpretovatelné řešení pro celou rodinu f-MDP. CMT efektivně mapuje podrodiny na příslušné kandidátské kontrolery (reprezentované identifikátory), zatímco UPT, syntetizovaný pomocí nástrojů jako dtControl, integruje rozhodovací logiku všech těchto kandidátů do jednoho kompaktního stromu. Tento přístup minimalizuje redundanci a zlepšuje celkovou srozumitelnost a kompaktnost ve srovnání se správou sady samostatných, často tabulárních, kontrolerů. Navržené techniky byly implementovány jako rozšíření nástroje PAYNT. Experimentální vyhodnocení na sadě zavedených benchmarků prokazuje, že naše metody významně redukují velikost kontrolerů a zvyšují jejich interpretovatelnost a to často za přijatelnou cenu v čase syntézy. Tato práce tak představuje významný krok k praktičtějšímu nasazení formálních metod pro syntézu srozumitelných a efektivních kontrolerů pro komplexní parametrizované systémy.

Decision trees for multi-environment Markov Decision Processes

Declaration

I hereby declare that this Master's thesis was prepared as an original work by the author under the supervision of Mr. doc. RNDr. Milan Češka, Ph.D. The supplementary information was provided by Mr. Andriushchenko Roman, Ing. I have listed all the literary sources, publications, and other sources that were used during the preparation of this thesis.

. . . .
Ladislav Dokoupil
May 19, 2025

Acknowledgements

I would like to express my sincere gratitude to my supervisor, doc. RNDr. Milan Češka, Ph.D., for his invaluable guidance, support, and insightful feedback throughout the development of this thesis. I am also thankful to Mr. Andriushchenko Roman, Ing., for his advice regarding implementation. A special thank you goes to my family and friends for their continuous encouragement, support, and motivation throughout this journey.

Contents

Chapter 1

Introduction

Markov Decision Processes (MDPs) represent a fundamental mathematical framework for modeling sequential decision-making under uncertainty. Their widespread adoption spans diverse domains including planning problems, robotics, reinforcement learning, and autonomous systems. A classical synthesis task in MDPs is to compute policies that achieve given specifications, such as reachability objectives. Several established tools exist for synthesizing such controllers for individual MDPs, including PRISM [15] and Storm [12]. These tools often tackle large state spaces, where techniques like abstraction refinement are crucial for managing complexity in probabilistic software [14]. As technological systems grow increasingly complex, there exists a critical need for controllers that not only achieve efficiency but also exhibit explainability and interpretability to human operators and end-users.

Traditionally, policies for individual MDPs are represented as mappings from states to actions, commonly implemented in tabular form. While ensuring determinism (i.e., each state is assigned exactly one action), this tabular representation presents significant challenges. It lacks intuitive interpretability and often requires substantial storage capacity, especially for large-scale MDPs with thousands of states. To address these limitations for single MDP policies, alternative representations like binary decision diagrams (BDDs) [10] and decision trees (DTs) [6] have been proposed. BDDs, while efficient for some tasks, can obscure the MDP's natural structure and diminish interpretability, particularly due to encoding complexities and the NP-hard problem of optimal variable ordering [21]. Decision trees, constructed by algorithms like CART [17], C4.5 [22], or Logistic Regression [16], offer a more interpretable structure by naturally identifying effective variable splits and representing decision logic as a hierarchical set of human-readable rules. This approach has been successfully implemented in the dtControl [7, 8], which synthesizes decision tree representations of MDP policies and other decision-making problems.

While standard MDPs offer a robust foundation, many real-world systems exhibit complexities, such as uncertain parameters [19], environmental variations [11], or varied operational configurations like alternative hardware choices [13], that are challenging to capture with a single, fixed MDP model. To address these scenarios, *families* of Markov Decision Processes (f-MDPs) provide a more expressive framework. Following the formalization in [5], f-MDPs generalize the standard MDP model by representing sets of related MDPs that typically share common state and action spaces but differ in their transition probabilities. This thesis addresses the significant challenge of synthesizing explainable and compact controllers for such f-MDPs. The objective for an f-MDP is to synthesize a solution, often a collection of policies, that together satisfy a given specification across the entire family. This work extends the PAYNT tool [4], which facilitates the synthesis of probabilistic programs from high-level specifications, by introducing advanced techniques for controller representation and synthesis specifically for f-MDPs.

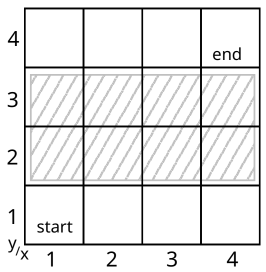
Synthesizing solutions for f-MDPs is more complex than for single MDPs. Naive enumeration of individual MDPs is computationally infeasible for large families. A key strategy involves finding either a single *robust policy* winning for all family members or, since such a robust policy for the entire family may not always exist, a *policy map* is more commonly employed. This map, often structured as a *policy tree* [5], partitions the family and assigns a specific winning policy (or indicates unsatisfiability) to each identified subfamily. The central challenge, which this thesis tackles, lies in efficiently constructing this policy map and representing both its structure and the constituent policies compactly and interpretably. A promising foundation for this, and the basis for our enhancements, is the approach in [5] which encodes the f-MDP problem as a two-player stochastic game [23]. Solving this game allows for the efficient synthesis of robust winning policies for subfamilies, which then form the leaves of the policy tree.

However, policies generated directly from such game-based abstractions often suffer from significant limitations. Primarily, these policies are designed against a worst-case environment, leading to conservative strategies that may include many irrelevant states or actions not strictly necessary for satisfying the specification in any family member. This results in unnecessarily large and complex policies. Furthermore, these policies are typically produced in a tabular format, which, as discussed, lacks interpretability and compactness. Finally, representing policies independently for each subfamily can lead to redundancy if similar logic is repeated across multiple policies.

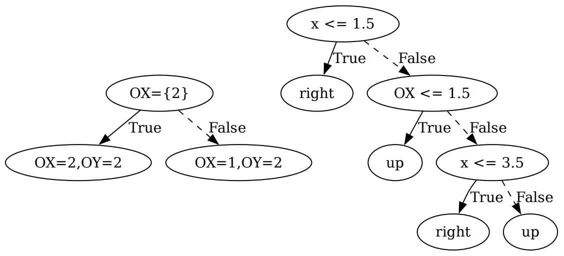
To overcome these shortcomings and to significantly improve the compactness and interpretability of controllers for f-MDPs, this thesis introduces a suite of novel techniques and representation strategies. First, we develop heuristic algorithms for **Policy State Pruning** to simplify the initial robust policies. By systematically eliminating irrelevant states and actions, these heuristics directly tackle the inherent conservatism, reducing policy complexity while preserving correctness guarantees. Second, we propose a novel **MDP Transformation for Alternative Policy Synthesis**. This method derives an MDP from the solved game abstraction of an f-MDP (sub)family. By fixing the environment's choices according to its optimal worst-case strategy (obtained from the game), we construct an MDP that models the family operating under these adversarial conditions. This transformation enables the use of existing MDP synthesis tools, such as dtNESt [3], to generate alternative policies, which can often be represented by smaller decision trees.

Third, to create a globally optimized controller for the entire family, we introduce a novel **Unified Decision Tree (UDT) Representation**. This two-tree structure, comprising a Candidate Mapping Tree (CMT) and a single Unified Policy Tree (UPT), integrates family mapping logic with policy execution logic. The UPT is synthesized using decision tree learning tools like dtControl from the set of (potentially pruned or dtNESt-generated) candidate policies, effectively minimizing redundancy and improving interpretability compared to collections of separate decision trees.

The effectiveness of these contributions is demonstrated through extensive experimental evaluation. Our findings show that the proposed UDT representation, particularly when combined with policy state pruning and the MDP transformation technique for leveraging tools like dtNESt, achieves significant reductions in controller size, in some cases resulting in an order of magnitude smaller representation, and enhances interpretability compared to existing methods. These improvements are often realized with acceptable trade-offs in



(a) Robot's objective: Exit the maze safely. Hatched areas indicate potential obstacle locations.



(b) Our unified decision tree concept. Left: Obstacle position (OX, OY) selects a candidate. Right: This candidate and the robot's position (x, y) determine an action.

Figure 1.1: Illustrative example for a family MDP and resulting unified decision tree.

synthesis time, underscoring the practical benefits of our approach for generating compact and understandable controllers for complex f-MDPs.

Illustrative example. We consider a robot navigating a grid maze with obstacles whose locations (OX, OY) are parameterized. Upon entering, the robot scans the environment to determine the obstacle configuration and aims to reach the exit while avoiding them (Figure 1.1a). We need a compact firmware solution for all configurations due to limited memory. A naive approach of enumerating all maze configurations is prohibitive. Our approach, as conceptualized in Figure 1.1b, produces a compact Unified Decision Tree (UDT). This UDT determines the robot's action through a distinct two-step process: First, the environmental parameters, specifically the obstacle locations (OX, OY), are processed by the Candidate Mapping Tree (CMT), which uses them to select an appropriate policy candidate. Then, in the second step, this chosen policy candidate, along with the robot's current state (x, y), becomes the input for the shared Unified Policy Tree (UPT). The UPT leverages both the general strategy (represented by the candidate) and the specific state to decide the final action for the robot. This structure avoids the duplication of state-dependent decision logic common in conventional approaches that might use separate controllers for each obstacle configuration. This unified and optimized structure results in a controller that is not only smaller and more efficient but also easier to interpret and verify.

The remainder of this thesis is structured as follows: Chapter 2 lays the theoretical groundwork, starting with fundamental concepts such as Markov Decision Processes (MDPs), their properties, and their generalization to families of MDPs (f-MDPs). It then introduces decision trees as a representation for policies, discusses compact input representations for f-MDPs using sketch files, and culminates in a formal problem statement that defines the core challenge of synthesizing compact and explainable controllers for these families. Chapter 3 reviews existing approaches to policy synthesis. It covers techniques for generating decision trees for single MDPs, including heuristic, formal, and hybrid methods, and then examines methods for handling families of MDPs, with a focus on game-based abstractions and their inherent limitations, thereby contextualizing our contributions. Chapter 4 details our novel contributions. We present our proposed pipeline, including heuristic-based policy state pruning algorithms, a refined policy merging strategy, a method for transforming the f-MDP problem into a derived MDP to leverage advanced synthesis tools like dtNESt, and our unified two-tree (CMT and UPT) representation for compact global controllers. Chapter 5 provides a rigorous empirical evaluation of our proposed methods. We compare their performance against baseline techniques using a suite of established benchmarks, analyzing aspects such as controller size, synthesis time, and the impact of different optimization stages. Finally, Chapter 6 summarizes the key findings and contributions of this thesis, discusses their implications for the field, and suggests promising avenues for future research in the synthesis of efficient and interpretable controllers for complex decision-making systems.

Chapter 2

Preliminaries and Problem Statement

This chapter provides a comprehensive theoretical foundation for the thesis, introducing key concepts, formal models, and tools that underpin the synthesis of explainable controllers for families of Markov Decision Processes (f-MDPs). We begin by defining MDPs and their properties, then generalize to families of MDPs, which capture parametric or uncertain environments. We further introduce a notion of decision trees and policy trees, which will later serve as compact representations of policies for f-MDPs. For most definitions, we refer to the standard literature on MDPs [20].

In the second part of this chapter, we introduce the concept of sketch files, which provide a compact and flexible representation for defining families of Markov Decision Processes (f-MDPs). This representation enables efficient modeling of parametric or uncertain environments. Finally, Section 2.3 formalizes the central problem addressed in this thesis: the synthesis of explainable and compact controllers for families of MDPs.

2.1 Markov Decision Processes

Markov Decision Processes (MDPs) are a fundamental mathematical framework for modeling sequential decision-making under uncertainty. They have become the standard formalism for solving optimization problems in stochastic environments where outcomes are partly random and partly under the control of a decision maker. The term "Markov" refers to the Markov property, which asserts that the future state of the system depends solely on the current state and the action taken, and not the history of states or actions that preceded it.

Definition 1 (Markov Decision Process)**.** *An MDP is defined as a tuple $M = (S, s_0, Act, P)$, where:*

- *S is a countable set of states,*
- *$s_0 \in S$ is the initial state,*
- *Act is a finite set of actions,*
- *$P : S \times Act \times S \to [0, 1]$. It is a partial transition function defining the probabilities of transitioning from one state to another when taking a specific action.*

For each state $s \in S$ and action $\alpha \in Act$, the function $P(s, \alpha, \cdot)$ defines a probability distribution over the set of states S whenever the action α is available in that state s. We write $P(s, \alpha, s') = P(s, \alpha)(s')$ to denote the probability of transitioning to state s' when taking action α in state s. *Available actions* can be defined as those actions that have a non-zero probability of leading to another state. We denote the set of available actions in state s as $Act_M(s) = \{\alpha \in Act \mid P(s, \alpha) \neq \bot\}$. The transition function P is defined as a partial function, meaning that for some states and actions, the transition may not be defined (denoted by $\bot$).

We omit the subscript M whenever the context makes it clear which MDP we are referring to. Throughout this thesis, we assume $Act(s) \neq \emptyset$ for each $s \in S$, meaning there are no deadlock states where no action can be taken.

Definition 2 (Markov Chain). *A Markov chain is a special case of an MDP where $|Act(s)| = 1$ for each $s \in S$, that is, there is precisely one action available in each state, eliminating the decision-making component. A Markov chain (MC) is denoted as a tuple (S, s_0, P).*

A finite path of an MDP is a sequence $\pi = s_0\alpha_0 s_1\alpha_1 \dots s_n$ where $P(s_i, \alpha_i, s_{i+1}) > 0$ for $0 \leq i < n$. This represents a possible trajectory through the MDP, starting from the initial state s_0 and taking actions α_i at each step, leading to the subsequent states s_{i+1}. The length of the path is denoted as $|\pi| = n$.

A *policy* (also called a strategy or controller) is a decision rule that specifies which action to take in each state of the MDP. A deterministic, memoryless policy is a function $\sigma : S \to Act$ where $\sigma(s) \in Act(s)$ for all $s \in S$. Such policies select exactly one action for each state, and the selection depends only on the current state, not on the history of previously visited states.

We denote by Σ_M the set of all deterministic, memoryless policies for MDP M. Each policy $\sigma \in \Sigma_M$ induces a Markov chain $M^\sigma = (S, s_0, P^\sigma)$ where $P^\sigma(s, s') = P(s, \sigma(s), s')$ for all $s, s' \in S$. Intuitively, M^σ represents the behavior of the system when actions are selected according to policy σ. Set of all paths in M^σ is denoted as π^σ. The probability of a finite path $\pi = s_0\alpha_0 s_1\alpha_1 \dots s_n$ in M^σ is defined as:

$$P(\pi) = \prod_{i=0}^{n-1} P(s_i, \alpha_i, s_{i+1}) \text{ for } n \geq 0$$

Specification and Winning Policies

MDPs are often analyzed with respect to specific properties, typically expressed as formal specifications using temporal logic formulas. These specifications enable reasoning about the system's behavior over time, including properties such as safety, liveness, and reachability.

In this work, we focus primarily on indefinite-horizon reachability properties [1], which are used to ensure safety and liveness guarantees in MDPs. These properties evaluate the probabilities of reaching or avoiding certain states within the MDP, providing a foundation for verifying and synthesizing reliable controllers.

Definition 3 (Reachability Property). *Let $M = (S, s_0, P)$ be a Markov chain and let $T \subseteq S$ be a set of target states. We denote by $P[M \models \Diamond T]$ the probability of reaching any state in T from s_0 in M.*

Reachability probability can be calculated as an integral over the probabilities of all finite paths starting in s_0 and ending in any state of T. Now assume MDP $M = (S, s_0, Act, P)$.

The maximum reachability probability can be extended as the maximum over all policies $\sigma \in \Sigma_M$: $P_{\max}[M \models \Diamond T] := \sup_{\sigma \in \Sigma_M} P[M^\sigma \models \Diamond T]$.

For the remainder of this thesis, we consider reachability specifications of the form $\varphi = P_{\geq \lambda}[M \models \Diamond T]$ for arbitrary thresholds $\lambda \in [0,1]$. A *winning policy* for MDP M with respect to the specification φ is a policy σ such that $P[M^\sigma \models \Diamond T] \geq \lambda$. We also denote this by $M, \sigma \models \varphi$. We say that MDP M is satisfiable with respect to a specification φ if there exists a winning policy for M. More detailed formal definitions can be found in standard references such as [20].

Decision Trees

Decision trees are a widely used representation for policies in MDPs and other decision-making problems. Traditional methods of representing policies in MDPs, such as tabular representations, often result in large structures that are difficult to interpret. By leveraging tree structures, we can create more compact and understandable representations for both individual policies and policy maps.

To formally define the decision trees used in this work, we begin with the basic structure of a *binary tree*, which provides the foundational scaffold for their decision logic. A binary tree can be described as a tuple $T = (V, l, r)$, where V is a set of nodes, and $l, r : V \to V$ are functions defining the left and right children, respectively.

To obtain a degree of abstraction for specific trees described later, we first define a generalized version. An *abstract decision tree* is a tree structure $\mathcal{T} = (T, \gamma, \delta)$ used to represent various decision-making processes. Here, T is a binary tree as defined above, γ assigns a decision predicate to each inner node, and δ assigns a label (e.g., an action or outcome) to each leaf node. Based on the satisfiability of the predicates assigned by γ, this tree is traversed from the root to a leaf node, where the label assigned by δ is obtained as the decision or action.

Definition 4 (Decision Tree)**.** *Given an MDP (S, s_0, Act, P) defined over the set V of variables with domains Dom, A decision tree (DT) is an Abstract Decision tree $\mathcal{T} = (T, \gamma, \delta)$ where:*

- *γ assigns to each inner node a predicate in the form $v_i \leq b_i$, where $v_i \in V$ and $b_i \in Dom(v_i)$*
- *δ assigns to each leaf node an action $a \in Act$*

A decision tree represents a policy $\sigma : S \to Act$ by partitioning the state space based on the values of specific variables and assigning actions to each partition. This hierarchical representation improves the interpretability of the policy and often reduces its size compared to a tabular format.

The policy encoded by a decision tree $\mathcal{T}$, referred to as the *policy induced by the decision tree*, is denoted as $\sigma_{\mathcal{T}}$. Formally, it is defined as $\sigma_{\mathcal{T}}(s) = \mathcal{T}(s)$ for all $s \in S$, where $\mathcal{T}(s)$ corresponds to the leaf node reached by traversing the tree based on the evaluation of predicates in γ.

For clarity, we omit the subscript $\mathcal{T}$ whenever the context makes it clear which decision tree we are referring to.

2.2 Families of MDPs

Although traditional approaches focus on solving individual Markov Decision Processes (MDPs), this thesis extends the scope to *families of MDPs* (f-MDPs). Families of MDPs generalize the standard MDP model by representing collections of MDPs that share the same state and action space but differ in their transition functions. This extension is particularly relevant in scenarios involving parametric uncertainty, environmental variability, or systems operating in multiple modes. The definitions and concepts used in this work build upon the foundational framework introduced in [5].

Definition 5 (Family of MDPs)**.** *A family of MDPs over the set S_M of states and set Act_M of actions is an indexed set $\mathcal{M} = \{(S_M, s_0, Act_M, P_i)\}_{i \in I}$ of MDPs, where I is a finite index set of identifiers.*

We assume that for each state, the sets of available actions coincide in all MDPs in the family:

$$\forall s \in S_M : \forall i, j \in I : P_i(s, \alpha) \neq \bot \Rightarrow P_j(s, \alpha) \neq \bot$$

As a consequence, all MDPs in the family have the same set of available policies, which we denote as Σ_M.

As already mentioned, the key distinction between family members is the transition function P_i. This may lead to various reachable state spaces and different winning policies for the same specification.

To efficiently reason about families of MDPs, we define an equivalence relation $\sim_{s,\alpha}$ on the index set I by $i \sim_{s,\alpha} j$ if and only if $P_i(s, \alpha) = P_j(s, \alpha)$, i.e., MDPs M_i and M_j have identical transition probabilities for action α in state s. We denote by $I/\sim_{s,\alpha}$ the corresponding equivalence partitioning of I with respect to $\sim_{s,\alpha}$. This partitioning will be helpful later when we introduce efficient encodings of MDP families.

Policy Map

Given the introduction of families of MDPs (f-MDPs), new challenges arise in synthesizing policies that satisfy specification requirements across the entire family. One such challenge is organizing the solution space effectively, which leads to the concept of a *policy map* [5].

Formally, a satisficing policy map is a function $P : \mathcal{M} \to \Sigma_M \cup \{\emptyset\}$ that associates each MDP in family $\mathcal{M}$ with either a winning policy from the set Σ_M or explicitly indicates unsatisfiability with the empty set. For any MDP $M_i \in \mathcal{M}$ and specification φ:

- $P(M_i) = \sigma$ if and only if $M_i, \sigma \models \varphi$ (policy satisfies the specification)
- $P(M_i) = \emptyset$ if and only if $M_i \not\models \varphi$ (no winning policy exists)

A straightforward approach to computing a policy map is to invoke a model checker for each member of the family. However, this naive approach becomes computationally infeasible for large families.

This shortcoming motivates the need for a more general solution: a robust policy. Such a policy satisfies the specification for all MDPs in the family, providing a unified solution that eliminates the need to compute multiple policies.

Robust Policy

Definition 6 (Robust Policy)**.** *Given a family of MDPs $\mathcal{M}$ and a specification φ, a policy σ is said to be robust for φ if it satisfies the specification for all MDPs in the family, i.e., $\forall M_i \in \mathcal{M}, M_i, \sigma \models \varphi$ [5].*

Robust policies are particularly valuable in scenarios where a controller must operate across diverse environments or under parameter uncertainty without requiring or limiting runtime parameter detection or policy switching.

However, it is essential to note that not all families of MDPs admit robust policies for a given specification. This limitation arises when the variations between MDPs in the family are too significant or when their optimal strategies fundamentally conflict, making it impossible to satisfy the specification universally with a single policy.

To better understand the structural properties of policies within a family of MDPs, we introduce the concept of *policy consistency*. A policy $\sigma \in \Sigma_Q$ is said to be consistent if and only if $\exists i \in I$ such that $\forall s \in S_{\mathcal{M}} : i \in \Gamma(\sigma(s))$. This means that the policy selects actions that are consistent with the transition dynamics of a specific MDP M_i in the family.

While consistency is not a requirement for robustness, understanding this property helps in analyzing the feasibility of robust policies and guides the development of algorithms for their synthesis.

Synthesizing robust policies often involves conservative optimization techniques, as the policy must perform well under worst-case scenarios across the entire family. This makes the problem inherently more challenging than solving for individual MDPs, as the policy must account for the full range of variability within the family while ensuring correctness for all instances.

Policy Tree

Similarly to representing policies, representing policy maps is infeasible in a simple tabular form. To address this, we introduce the concept of a *policy tree* that can represent a policy map for a family of MDPs.

Definition 7 (Policy Tree)**.** *Given a family of MDPs $\mathcal{M} = \{M_i\}_{i \in I}$, a policy tree is a abstract decision tree $\mathcal{T} = (T, \gamma, \delta)$ where:*

- *γ assigns to each inner node a predicate in the form $v_i \subseteq b_i$, where $v_i \subseteq I$ and $b_i \subseteq I$ are sets of MDP identifiers*
- *δ assigns to each leaf node a policy $\sigma \in \Sigma_M \cup \{\emptyset\}$ for the corresponding subfamily of MDPs*

The policy tree $\mathcal{T}$ maps the family of MDPs to policies by recursively partitioning the family at each inner node based on the predicate $\gamma(n)$. Each leaf node l represents a subfamily of MDPs that share the same winning policy $\delta(l)$. An example of a policy tree is shown in Figure 2.1.

Policy trees provide a compact and interpretable representation of policy maps, enabling efficient reasoning and decision-making. In practical implementations within this work, policy trees are often N-ary, and their decision nodes typically partition the family's parameter space. However, for theoretical analysis, these aspects can be simplified to binary trees with predicates on MDP identifiers without any loss of generality.

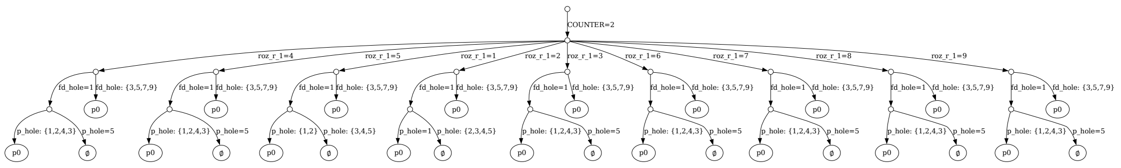


Figure 2.1: Detail of a policy tree segment for benchmark `uav_roz`.

Compact Input Problem Representation

A direct, tabular representation of a family of MDPs is impractical, as it results in a representation size that grows proportionally with the number of MDPs in the family. Such representation also fails to capture the structural similarities between MDPs in the family, which are essential for efficient synthesis.

To address these limitations, we introduce a compact and flexible representation of MDP families using sketch files [24]. A sketch is a template-based description of a program written in an enhanced version of the PRISM language [15]. In this context, a sketch defines a program with "holes", placeholders that can take discrete values from finite sets. By systematically instantiating these holes, the sketch generates a finite family of programs, where each instantiation corresponds to a specific MDP.

Each program in the family is characterized by shared states and actions, but with distinct transition probabilities. This approach allows for the efficient representation of parametric or uncertain environments. A standard PRISM file is structured as a set of modules, each containing variables, actions, and updates of the form:

$$[actionId]\ guard \rightarrow prob_1 : update_1 + \ldots + prob_n : update_n;$$

Here, the guard is a condition that must be satisfied for the action (right side of the guard) to be taken. Probabilities of action $prob_1, \ldots, prob_n$ are non-negative real numbers that sum to 1, between which a non-deterministic choice is made. Updates $update_1, \ldots, update_n$ are expressions that generally describe how the variables in the module are updated when the action is taken. $actionId$ is an identifier of the action that can be used to force the synchronous choice of actions in different modules if the name coincides.

The concept of holes extends the PRISM language by allowing the definition of variables whose values are not fixed but can be filled with any value from a specified set. For example:

```
hole int startX in {0..maxX};
hole int startY in {0..maxY};
```

In this example, the holes `startX` and `startY` are defined as integer variables that can take values from the ranges $[0, maxX]$ and $[0, maxY]$, respectively. Each unique combination of hole values generates a distinct MDP instance.

The primary objective is to synthesize a concrete policy or a minimal set of policies that satisfy a given reachability specification for all possible instantiations of the holes.

2.3 Compact Solution Representation

Building on the formal theoretical framework established thus far, we now articulate the central problem addressed in this thesis.

Given a family of MDPs $\mathcal{M} = \{M_i\}_{i \in I}$ defined by a parameterized sketch program and a reachability objective φ, the goal is to synthesize a minimal representation of a satisfiable solution for the entire family. Formally, we seek to minimize the representation size of a policy mapping function $P : \mathcal{M} \to \Sigma_M \cup \{\emptyset\}$ that assigns either a winning policy or explicitly indicates unsatisfiability for each MDP in the family. This optimization problem involves two key objectives:

1. Minimizing the representation of individual policies $\sigma \in \Sigma_M$ for each identifiable subfamily of MDPs that shares a common winning strategy.

2. Minimizing the structure that maps these policies to their corresponding families within the parameter space.

This minimization can be achieved in two alternative ways. One is to minimize the problem state space, which involves identifying and removing states that are redundant or unnecessary for solving the problem. This approach aims to reduce the size of individual policies by eliminating irrelevant choices. By focusing only on states that are essential for satisfying the specification, we can create more compact policy representations.

The second approach focuses on optimizing the output encoding by leveraging and enhancing state-of-the-art decision tree synthesis algorithms. These algorithms encode policies (and policy maps) as decision trees, which provide a hierarchical structure for representing the decision-making process. The details of this approach, along with a discussion of existing techniques and their limitations, will be presented in the next chapter.

Chapter 3

State of the Art

This chapter situates our research within the broader context of policy synthesis for Markov Decision Processes (MDPs) and families of MDPs (f-MDPs). We analyze existing approaches, identify their limitations, and establish the foundation for the contributions of this work.

Efficiently representing controllers for MDPs presents significant challenges, particularly when scaling to families of MDPs. Traditional tabular representations, while precise, become impractical for large state spaces due to their size and lack of interpretability. Existing compact representations, such as binary decision diagrams (BDDs) and decision trees, address some of these issues, but often fail to scale effectively for parameterized models or sacrifice interpretability.

We begin by reviewing decision tree synthesis techniques for individual MDPs, including heuristic-based methods (e.g., dtControl [7]), formal verification approaches (e.g., dtPaynt [6]), and hybrid methodologies (e.g., dtNESt [3]). These techniques have demonstrated success in generating compact and interpretable policies for single MDPs. We then explore how these techniques extend or fail to extend to families of MDPs, discussing the critical gaps that emerge when moving from single MDPs to parameterized families.

Next, we examine existing approaches for synthesizing policies for families of MDPs. These range from naive enumeration strategies, which solve each MDP independently, to game-based abstractions that reason about the entire family simultaneously [5]. Although game-based abstraction avoids the computational infeasibility of enumeration, it often produces overly complex controllers with redundant states and actions.

Throughout this work, we highlight how our work addresses these limitations through three complementary approaches: (1) systematic elimination of irrelevant states, (2) improved encoding of decision structures, and (3) unification of policy and decision trees into a single coherent representation.

By addressing these challenges, our work aims to advance the state of the art in policy synthesis for f-MDPs, enabling more interpretable solutions. The remainder of this chapter provides a detailed review of existing techniques, their limitations, and how our contributions build upon and extend these methods.

3.1 Decision Trees for MDP

Decision trees have emerged as a practical approach for representing controllers in MDPs due to their interpretability and compact form. We review key algorithms and approaches for constructing decision trees to represent policies.

The most basic representation of a policy is a tabular format that maps states to actions: $\sigma : S \to Act$. While this representation is precise, it becomes impractical for large MDPs, as the size of the table grows proportionally with the number of states.

Due to the limitations of large table representations, decision trees have gained prominence. Decision trees provide a hierarchical partitioning of the state space, capturing the underlying structure of the decision-making process while significantly reducing the representation size. Several approaches exist for constructing these tree representations, each with distinct advantages and limitations.

dtControl

dtControl [7] is a specialized tool for compressing and optimizing memoryless policies for MDPs. It encodes the policies as a reinforcement learning problem and applies a set of well-known machine learning algorithms to generate decision trees, such as C4.5, CART, ID3, or logistic regression.

These algorithms work by iteratively splitting the data into subsets based on the value of input features, leading to a tree-like structure where each leaf node represents a classification (in our case, an action). The splitting criteria typically include information gain, Gini impurity, and mean squared error.

The main advantage of dtControl is its computational efficiency, as the algorithms are designed to work with large datasets and can handle high-dimensional data. However, since these algorithms are greedy, they may converge at local optima and fail to find the globally optimal decision tree.

dtPaynt

dtPaynt [6] has been developed as an extension of the PAYNT program to find potentially more efficient decision trees than those generated by dtControl. This framework formulates the tree encoding problem as an SMT (Satisfiability Modulo Theories) encoding task.

The process begins by constructing an abstract decision tree of fixed depth, with free predicates and actions assigned to its nodes. Using SMT encoding, the algorithm verifies whether the policy can be represented by the given tree. The SMT solver either returns a valid assignment of predicates and actions (SAT) or identifies an unsatisfiable core, which highlights the constraints causing the failure. The UNSAT core is then used to refine the abstract decision tree predicates, and the process repeats until a valid decision tree is found or the maximum number of iterations is reached. If no solution is found, the tree depth is increased, and the process is restarted.

The primary advantage of dtPaynt is its ability to generate decision trees with minimal depth, offering a more predictable structure compared to the heuristic-based approach of dtControl. However, this method is computationally expensive, as SMT solving becomes impractical for trees with depths exceeding 8. Additionally, dtPaynt does not explicitly optimize the number of nodes in the tree, which may result in suboptimal tree sizes despite achieving minimal depth.

dtNESt

dtNESt [3] represents a hybrid approach that combines the strengths of heuristic-based methods (dtControl) and formal verification techniques (dtPaynt). It addresses fundamental limitations in existing approaches by generating more compact, interpretable decision trees for MDPs.

The core innovation of dtNESt lies in its novel abstraction refinement technique, which operates through a bidirectional optimization process:

- **Top-down heuristic tree construction**: Starts by constructing a decision tree from the starting optimal policy using heuristic methods.
- **Bottom-up formal verification**: Problematic or suboptimal subtrees are identified and refined using formal verification techniques.
- **Iterative refinement**: The optimized subtrees are reintegrated into the overall structure, iteratively repeating the process.

This hybrid methodology enables dtNESt to overcome the limitations of both approaches used in isolation. While dtNESt is less restrictive than dtPaynt in terms of depth limitations, its scalability remains a challenge for very large MDPs.

Limitations of Existing Approaches

Current research provides methods for synthesizing decision trees for individual MDPs. However, these approaches are not directly applicable to families of MDPs.

The most obvious alternative, often termed naive enumeration, involves exhaustively generating each MDP within the family and applying established synthesis algorithms individually. This brute-force technique suffers from two significant drawbacks: Firstly, its computational cost is typically prohibitive for families of non-trivial size. Secondly, treating each MDP in isolation disregards the potential to exploit structural similarities across the family. Consequently, there is a pressing need for more efficient methods that can synthesize robust decision trees for families of MDPs.

3.2 Policy Trees for Families of MDPs

Synthesizing controllers for families of MDPs presents unique challenges. As discussed previously, representing the solution often requires a policy map, potentially structured as a policy tree (defined in Section 2.2). However, constructing this map efficiently is non-trivial. Naive approaches are computationally infeasible for large families, such as those that require enumerating or solving each MDP $M_i \in \mathcal{M}$ individually. Moreover, we are interested in a small set of robust policies to reduce the overall solution size. Similarly to the dtPaynt approach, one could encode the entire family synthesis problem as a single monolithic SMT formula. However, this quickly becomes intractable.

A more sophisticated approach, proposed in [5], leverages a game-based abstraction to reason about the entire family simultaneously, avoiding costly enumeration. This method forms the basis upon which parts of our work are built and extended. The core idea involves modeling the f-MDP synthesis problem as a two-player stochastic game.

Quotient MDP

To efficiently represent the shared structure and variations within an f-MDP, the concept of a quotient MDP is introduced. This serves as a compact representation and the foundation for the game abstraction.

Definition 8 (Quotient MDP)**.** *Let $\mathcal{M} = \{M_i\}_{i \in I}$ be a family of MDPs. The* quotient MDP *is a pair $(Q_{\mathcal{M}}, \Gamma)$, where $Q_{\mathcal{M}} = (S_{\mathcal{M}}, s_0, \mathrm{Act}_Q, P_Q)$ is an MDP over the set of actions $\mathrm{Act}_Q = \mathrm{Act}_{\mathcal{M}} \times 2^I$, and:*

- *The action set is defined as:*

$$\mathrm{Act}_Q(s) = \{(\alpha, I) \mid \alpha \in \mathrm{Act}(s),\ I \in I^{\sim}_{s,\alpha}\}$$

  Action (α, I) is denoted as α^I. Action $\alpha^{\{i\}}$ is denoted as α^i for brevity.

- *The transition function is defined as:*

$$P_Q(s, \alpha^I, s') = P_i(s, \alpha, s') \quad \text{where } i \in I$$

- *The function $\Gamma : \alpha^I \mapsto I$ maps an action to its set of identifiers.*

Essentially, the quotient MDP $Q_{\mathcal{M}}$ allows executing action α in state s from an arbitrary $M_i \in \mathcal{M}$. The shape (α, I) of actions allows us to efficiently encode families of MDPs where action α coincides in many family members. Figure 3.1b illustrates this, by merging action γ shared by MDPs 1 and 2 into $\gamma^{\{1,2\}}$. We omit the subscript $\mathcal{M}$ when the context is clear.

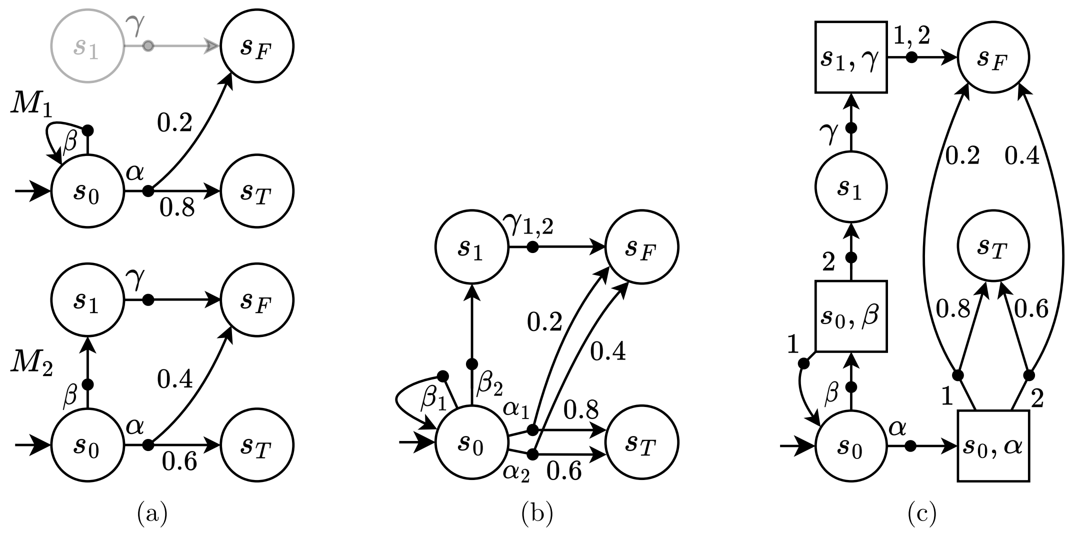


Figure 3.1: (a) Family $\mathcal{M}$ of two MDPs. (b) Quotient MDP $Q_{\mathcal{M}}$ for the family $\mathcal{M}$. (c) Game abstraction for family $\mathcal{M}$. Circles (squares) denote states of Player 1 (2) Transitions without explicit probability have a probability of 1. States s_T and s_F are absorbing, and their action with the self-loop is omitted. Taken from [5].

Stochastic Game Formulation

The synthesis problem is framed as a zero-sum, two-player stochastic game [23] (SG) played on a structure derived from the quotient MDP.

Definition 9 (Stochastic Game). *A stochastic game (SG) is a tuple $G = (\mathcal{M}', S_1, S_2)$ where $\mathcal{M}' = (S, s_0, \mathrm{Act}, P)$ is an underlying MDP (in our case, derived from $Q_{\mathcal{M}}$) and (S_1, S_2) is a partition of the state set S into Player 1 and Player 2 states.*

Player 1 (the agent) aims to maximize the probability of satisfying the specification (e.g., reaching T), while Player 2 (the environment or parameter selector) aims to minimize it. Policies σ_1 for Player 1 and σ_2 for Player 2 determine the actions taken in S_1 and S_2 respectively. A pair (σ_1, σ_2) induces a Markov chain G^{σ_1,σ_2}. Alternatively, we can denote this set as $(\sigma_{\text{agent}}, \sigma_{\text{env}})$. For such reachability games, optimal memoryless deterministic policies exist for both players. The value of the game for a reachability objective is:

$$\mathbb{P}_{\max}[G \models \lozenge T] := \sup_{\sigma_1 \in \Sigma^1_{\mathcal{M}}} \inf_{\sigma_2 \in \Sigma^2_{\mathcal{M}}} \mathbb{P}[G^{\sigma_1,\sigma_2} \models \lozenge T]$$

Game Abstraction

The specific game abstraction $G_{\mathcal{M}}$ for an f-MDP $\mathcal{M}$ is constructed to separate the agent's choices from the environmental uncertainty (i.e., the choice of M_i).

Definition 10 (Game Abstraction [5]). *Let $\mathcal{M} = \{M_i\}_{i \in I}$ be a family of MDPs. The game abstraction for $\mathcal{M}$ is a stochastic game $G_{\mathcal{M}} = (\mathcal{M}', S_1, S_2)$ with $S_1 = S_{\mathcal{M}}$, $S_2 = S_{\mathcal{M}} \times \mathrm{Act}_{\mathcal{M}}$, and an underlying MDP $\mathcal{M}' = (S_1 \cup S_2, s_0, \mathrm{Act}_G, P_G)$ where:*

- $\mathrm{Act}_G = \mathrm{Act}_{\mathcal{M}} \cup \{I' \mid I' \in I^{\sim}_{s,\alpha}$ *for some* $s, \alpha\}$
- *Player 1 states $s \in S_1$: Player 1 chooses an action $\alpha \in \mathrm{Act}_{\mathcal{M}}(s)$. The transition is deterministic: $P_G(s, \alpha, (s, \alpha)) = 1$.*
- *Player 2 states $(s, \alpha) \in S_2$: Player 2 chooses an equivalence class $I \in I^{\sim}_{s,\alpha}$ representing a set of MDPs with identical transitions for (s, α). The transition probabilities are determined by any M_i with $i \in I$: $P_G((s, \alpha), I, s') = P_i(s, \alpha, s')$.*

As illustrated figure 3.1c in this game, Player 1 chooses the intended action α, and then Player 2 chooses the specific transition dynamics (represented by I) that will occur, effectively selecting the worst-case MDP from the perspective of Player 1 for that state-action pair.

Policy Synthesis via Game Solving

Solving this game yields optimal policies σ_1^* and σ_2^* for Player 1 and Player 2, respectively. The key insight is the connection between the game solution and policies for the original f-MDP.

Definition 11 (Consistent Game Policy). *We say that Player 2 policy $\sigma_2 \in \Sigma^2_{\mathcal{M}}$ is* consistent in identifier i *if $\exists i \in I\ \forall (s, \alpha) \in S_2 : i \in \sigma_2(s, \alpha)$. That is, Player 2 consistently selects transitions corresponding to a single MDP M_i.*

Lemma 1. *If σ_2 is consistent in i, then for any Player 1 policy σ_1:*

$$\mathbb{P}[G^{\sigma_1,\sigma_2}_{\mathcal{M}} \models \lozenge T] = \mathbb{P}[M_i^{\sigma_1} \models \lozenge T]$$

This lemma connects the game outcome under a consistent Player 2 strategy to the outcome in a specific MDP M_i. The main theorem provides the basis for synthesizing robust policies:

Theorem 1 (Policy from Winning Game [5])**.** *Let $\mathcal{M} = \{M_i\}_{i \in I}$ and its game abstraction $G_{\mathcal{M}}$. Let σ_1^* be an optimal policy for Player 1. If the value of the game satisfies $\mathbb{P}_{\max}[G_{\mathcal{M}} \models \lozenge T] \geq \lambda$, then σ_1^* is a robust winning policy for the entire family $\mathcal{M}$:*

$$\forall M_i \in \mathcal{M} : \mathbb{P}[M_i^{\sigma_1^*} \models \lozenge T] \geq \lambda$$

If the game value is below the threshold λ, a robust policy for the entire family $\mathcal{M}$ might not exist or cannot be found by this conservative abstraction. In such cases, the native approach uses the optimal Player 2 strategy σ_2^* to identify how to partition the family $\mathcal{M}$ into smaller subfamilies. The process is then applied recursively to these subfamilies. This recursive partitioning naturally generates a policy tree structure where internal nodes represent splits in the family based on Player 2's choices (predicates on parameters/identifiers).

Properties and Limitations of the Game Abstraction

The game abstraction provides a sound method for synthesizing robust policies, as established by Theorem 1. If the game value meets the threshold λ, the resulting Player 1 policy σ_1^* is guaranteed to be winning for the *entire* family $\mathcal{M}$. This ability to reason about the whole family simultaneously, leveraging structural similarities, is a key advantage over naive enumeration.

However, the abstraction is not complete. A game value below the threshold ($\mathbb{P}_{\max}[G_{\mathcal{M}} \models \lozenge T] < \lambda$) does not necessarily imply that no robust policy exists for $\mathcal{M}$, nor that any specific M_i is unsatisfiable. This incompleteness stems from the structure of the game, which grants Player 2 significant power: Player 2 chooses the worst-case MDP dynamics (I) after observing Player 1's chosen action α. This information asymmetry allows Player 2 to adapt its strategy based on Player 1's move in a way that might not correspond to any single, fixed MDP M_i across all states. The resulting σ_1^* policy is therefore conservative, designed to win against this powerful adversary.

A specific condition exists where unsatisfiability can be deduced: if the optimal Player 2 strategy σ_2^* happens to be consistent in some identifier i (meaning Player 2 always chooses transitions corresponding to M_i) and the game value is below λ, then we can conclude that M_i itself is unsatisfiable ($\mathbb{P}_{\max}[M_i \models \lozenge T] < \lambda$), and thus no robust policy exists for $\mathcal{M}$.

The conservatism inherent in the game abstraction, while ensuring robustness, presents two main limitations that motivate the contributions of this thesis:

1. **Policy Complexity**: The resulting robust policy σ_1^* might be unnecessarily complex or suboptimal in terms of performance for many individual MDPs M_i within the family, as it's designed for the worst-case scenario dictated by the powerful Player 2. It may contain actions or cover states that are only relevant for specific, adversarial choices made by Player 2.

2. **Lack of Size Guarantees**: The game abstraction provides no guarantees on the size or interpretability of the resulting policy σ_1^*, which is typically generated in a tabular format.

Our work directly addresses these limitations. Firstly, we introduce pruning techniques (Section 4.2) to reduce the conservatism by identifying and eliminating states and actions from σ_1^* that are not essential for satisfying the property φ, leading to more compact policies. Secondly, we propose transforming the solved game back into an MDP by fixing Player 2's strategy (Section 4.4). This allows us to leverage state-of-the-art decision tree synthesis algorithms like dtNESt to generate compact and interpretable policy representations (Section 4), moving beyond the limitations of tabular formats.

Chapter 4

Synthesis of Small Policy Trees

The state-of-the-art approach for synthesizing controllers for families of MDPs (f-MDPs) [5], based on game abstraction, yields a policy tree. This tree maps subfamilies of $\mathcal{M}$ to robust policies σ_{agent} for Player 1 (the agent). As discussed in Section 3.2, while sound, this method has limitations:

- The resulting policies σ_{agent} are often conservative and overly complex due to the nature of the game abstraction.
- The policies are generated in a tabular format, which lacks interpretability.
- Robust policies are represented independently for each subfamily, leading to potential redundancy and inefficiency in the overall solution.

This chapter introduces novel techniques designed to overcome these limitations by transforming the output of the game abstraction into more efficient and interpretable tree-based representations. Our goal is to minimize the overall size of the solution representation, encompassing both the structure mapping subfamilies to policies and the representation of the policies themselves.

4.1 Overview

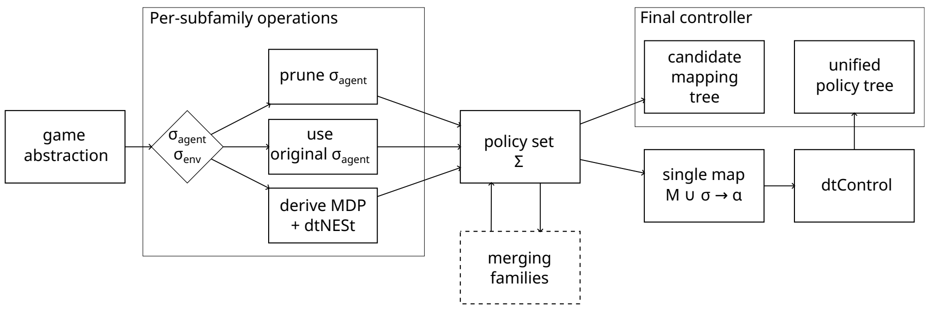


Figure 4.1: Simplified synthesis pipeline detailing our work. Dashed blocks indicate optional steps.

Our pipeline, illustrated in Figure 4.1, begins with the policy tree and its associated robust agent policies σ_{agent} produced by the game abstraction process [5]. While correct, these policies are often conservative and tabular, leading to potentially large and less interpretable solutions. This chapter introduces novel techniques to transform these outputs into more efficient and compact tree-based controllers. Our approach involves several enhancement stages: First, individual policies obtained from the game abstraction can be refined using *per-subfamily operations*. Second, the resulting set of candidate policies can be optimized by reducing the number of distinct policies. Finally, these policies are integrated into a unified representation. Our contributions, detailed below, provide specific methods for these stages:

1. **Policy State Pruning (Section 4.2)**: Heuristic algorithms simplify the robust agent policies by removing irrelevant states and actions, directly addressing the game abstraction's conservatism while preserving correctness. This stage corresponds to the `Prune` block in Figure 4.1.

2. **Deriving an MDP for Alternative Policy Synthesis (Section 4.4)**: We obtain an MDP from the solved game abstraction for a given family by fixing the environment's strategy σ_{env}. This enables leveraging tools like dtNESt [3] to synthesize alternative policies, which are often represented as compact decision trees, as shown by the `Derive MDP + dtNESt` block.

3. **Enhanced Policy Merging (Section 4.3)**: Our refined strategy optimizes the set of candidate policies by reducing the number of distinct policies required for the overall solution. This corresponds to the `merging families` block in Figure 4.1.

4. **Unified Decision Tree Representation (Section 4.5)**: We propose a method to combine the family mapping logic and individual policy behaviors into a single, globally optimized Unified Decision Tree (UDT), aiming to minimize redundancy and improve interpretability. This involves constructing a Candidate Mapping Tree (CMT) and then synthesizing a Unified Policy Tree (UPT) using tools like dtControl from the set of (refined and merged) candidate policies, corresponding to the final stages in Figure 4.1.

To illustrate the benefits of our integrated approach, consider the robot navigation example introduced in Figure 1.1 (Chapter 1). The initial game-based synthesis for such a problem might yield a policy tree with numerous leaves, each corresponding to a different obstacle configuration and potentially leading to a separate, tabular policy. Our techniques aim to transform this initial, potentially unwieldy solution into a single, more compact, and interpretable controller. Specifically, policy state pruning (Section 4.2) would simplify each of these policies. By deriving an MDP for alternative synthesis (Section 4.4), we could alternatively generate policies of lower complexity, often representable as more compact decision trees than the original policies. Subsequently, our enhanced policy merging (Section 4.3) would reduce the number of distinct policy candidates needed. Finally, the unified decision tree representation (Section 4.5), particularly the two-tree CMT and UPT structure shown in Figure 1.1b, would integrate the logic for mapping obstacle configurations to concrete policies and the execution of those policies into a globally optimized structure. This holistic approach seeks to minimize redundancy and significantly improve both the size and the interpretability of the final controller for the entire family of maze configurations.

4.2 Pruning Policy States

The initial policies targeted by our techniques are synthesized using the stochastic game abstraction approach from [5]. This method involves solving a two-player game $G_{\mathcal{M}}$ derived from the quotient MDP for a given (sub)family $\mathcal{M}$. In this game, Player 1 (the agent) maximizes the property φ against Player 2 (the environment), who minimizes it by choosing the worst-case MDP dynamics after observing Player 1's action. This grants Player 2 an informational advantage.

Solving this game (e.g., via value iteration) yields optimal policies for both players, $(\sigma_{\text{agent}}, \sigma_{\text{env}})$. The agent's policy, σ_{agent}, is guaranteed to be robust for the entire family $\mathcal{M}$. However, this robustness, stemming from Player 2's advantage and adversarial nature, means σ_{agent} is often conservative, stronger than strictly required by the property φ. It might specify actions in states or handle transitions that are only relevant under the most adversarial environmental choices dictated by σ_{env}.

This conservatism implies that the policy σ_{agent} might still satisfy the property $\varphi = P_{\geq\lambda}[\lozenge T]$ even if some of its state-action pairs $(s, \sigma_{\text{agent}}(s))$ were removed or replaced with less effective actions for states not essential to maintain the guarantee. This creates an opportunity to simplify the policy representation by identifying a core set of *relevant states* and actions, and defining a new, potentially smaller policy based on them, without compromising the winning guarantee. Our goal is to identify a minimal set of relevant states $S_{\text{relevant}} \subseteq S_1$ and define a policy σ_{relevant} based on σ_{agent} for these states (using arbitrary action elsewhere), such that σ_{relevant} is still robust and winning for $\mathcal{M}$.

Pruned Policy

Let $(\sigma_{\text{agent}}, \sigma_{\text{env}})$ be the optimal policy pair obtained from solving the game $G_{\mathcal{M}}$, such that the value of the game $V(s_0) = \mathbb{P}[G^{\sigma_{\text{agent}},\sigma_{\text{env}}} \models \lozenge T] \geq \lambda$. The core idea of pruning is to identify a subset of essential Player 1 states $S_{\text{relevant}} \subseteq S_1$ and define a policy σ_{relevant} as:

$$\sigma_{\text{relevant}}(s) = \begin{cases} \sigma_{\text{agent}}(s) & \text{if } s \in S_{\text{relevant}} \\ \alpha_{\text{noop}} & \text{if } s \in S_1 \setminus S_{\text{relevant}} \end{cases}$$

Where α_{noop} is a designated action assumed to be no better (and potentially worse) for Player 1 than any other available action in terms of satisfying φ.

Noop Action Justification: For maximizing reachability ($\lozenge T$), a common and safe choice for α_{noop} is a self-loop action, where $P(s, \alpha_{\text{noop}}, s) = 1$. Let $V_{\sigma_{\text{agent}},\sigma_{\text{env}}}(s)$ be the reachability probability from state s in the induced Markov chain $G^{\sigma_{\text{agent}},\sigma_{\text{env}}}$. Since σ_{agent} is optimal for Player 1, replacing $\sigma_1(s)$ with any action α cannot increase the resulting reachability probability from state s when played against the optimal σ_{env}. Specifically, for a self loop α_{noop}:

$$\sum_{s'} P(s, \alpha_{\text{noop}}, s') V_{\sigma_{\text{agent}},\sigma_{\text{env}}}(s') = V_{\sigma_{\text{agent}},\sigma_{\text{env}}}(s)$$

Replacing $\sigma_{\text{agent}}(s)$ with α_{noop} in states $s \in S_1 \setminus S_{\text{relevant}}$ can, therefore, only decrease or maintain the overall reachability probability $V(s_0)$.

Lemma 2 (Robustness with Relevant States). *Let $\sigma_{relevant}$ be defined as above. If the set $S_{relevant}$ is chosen such that the reachability probability in the game $G_{\mathcal{M}}$ under the modified policy pair $(\sigma_{relevant}, \sigma_{env})$ still meets the threshold, i.e., $\mathbb{P}[G^{\sigma_{relevant},\sigma_{env}} \models \lozenge T] \geq \lambda$, then $\sigma_{relevant}$ is a robust winning policy for the family $\mathcal{M}$.*

By Theorem 1, if $\mathbb{P}_{\max}[G_{\mathcal{M}} \models \Diamond T] \geq \lambda$, then σ^*_{agent} is robust. Lemma 2 states that if the policy σ_{relevant} still achieves a value $\geq \lambda$ against the optimal adversary σ_{env} in the game $G_{\mathcal{M}}$, it must also be robust for the original family $\mathcal{M}$, as the game value provides a lower bound on the performance in any individual M_i. The heuristics described below aim to find such a sufficient set S_{relevant}.

This approach of identifying relevant states is particularly beneficial when followed by decision tree synthesis tools like dtControl. These tools aim to find a compact tree representation that mimics the behavior of a given policy. After pruning, the states $S_1 \setminus S_{\text{relevant}}$ can be treated as irrelevant states with regard to property satisfaction, allowing flexibility in action choice by subsequent synthesis tools.

Theorem 2 (Correctness of DT Synthesis with Relevant States)**.** *Let $\sigma_{relevant}$ be a robust winning policy obtained by identifying a sufficient set $S_{relevant}$ (Lemma 2). Let σ_{DT} be a policy represented by a decision tree such that:*

- *For $s \in S_{relevant}$, $\sigma_{DT}(s) = \sigma_{relevant}(s) = \sigma_{agent}(s)$.*
- *For $s \in S_1 \setminus S_{relevant}$, $\sigma_{DT}(s)$ can be any action $\alpha \in Act(s)$.*

Then σ_{DT} is also a robust winning policy for $\mathcal{M}$.

We know $\mathbb{P}[G^{\sigma_{\text{relevant}},\sigma_{\text{env}}} \models \Diamond T] \geq \lambda$, where σ_{relevant} uses α_{noop} in $S_1 \setminus S_{\text{relevant}}$. Since σ_{agent} was optimal against σ_{env}, any action $\alpha \in Act(s)$ chosen by dtControl for $s \in S_1 \setminus S_{\text{relevant}}$ cannot yield a higher reachability probability from s than $\sigma_{\text{agent}}(s)$ did. Crucially, assuming α_{noop} represents the worst possible outcome for Player 1, any action α chosen by dtControl cannot be worse than α_{noop}. Therefore, the overall reachability probability under $(\sigma_{\text{DT}}, \sigma_{\text{env}})$ cannot be lower than under $(\sigma_{\text{relevant}}, \sigma_{\text{env}})$. Thus, $\mathbb{P}[G^{\sigma_{\text{DT}},\sigma_{\text{env}}} \models \Diamond T] \geq \mathbb{P}[G^{\sigma_{\text{relevant}},\sigma_{\text{env}}} \models \Diamond T] \geq \lambda$. By Theorem 1, σ_{DT} is robust winning for $\mathcal{M}$.

This theorem justifies allowing dtControl the freedom to choose any action for the irrelevant states $S_1 \setminus S_{\text{relevant}}$ in the interest of creating a smaller decision tree, without violating the correctness guarantee.

Heuristic Pruning Algorithms

The pruning process aims to identify a minimal set of states S_{relevant} such that the policy σ_{relevant} satisfies the property φ.

While the above provides theoretical justification, finding the absolute minimal set S_{relevant} that satisfies Lemma 2 is computationally hard. Instead, we employ heuristic algorithms to identify a reasonably small, sufficient subset of states. These algorithms explore the state space of the Markov Chain induced by $(\sigma_{\text{agent}}, \sigma_{\text{env}})$, starting from s_0, and incrementally build S_{relevant}.

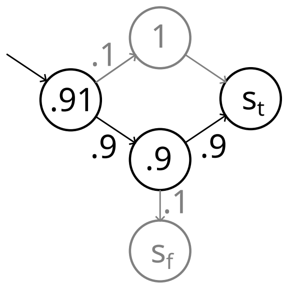
The core idea is to prioritize exploring states deemed more important for reaching the target set T. We use exploration-based algorithms guided by priority queues, effectively performing a guided search. The exploration stops once the partially built policy σ_{relevant} (defined only on the currently visited states $\mathcal{V} = S_{\text{relevant}}$ and using α_{noop} elsewhere) is verified to satisfy φ. To achieve this, we introduce two complementary heuristics to guide this exploration:

1. **Reachability-based Exploration**: Prioritizes exploring states s with higher reachability values $V_{\sigma_{\text{agent}},\sigma_{\text{env}}}(s)$. These values, representing the probability of reaching T

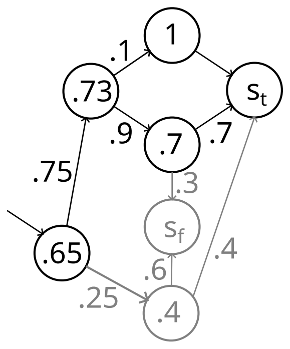
from s under $(\sigma_{\text{agent}}, \sigma_{\text{env}})$ in the induced Markov chain, are typically available from the game-solving phase. This heuristic focuses on paths most likely to lead to the target set T. (Details in Algorithm 1).

2. **Maximum Probability Transition Exploration**: Prioritizes exploring transitions $(s, \sigma_{\text{agent}}(s), s')$ with the highest probability $P(s, \sigma_{\text{agent}}(s), s')$. This heuristic focuses on the most likely execution paths within the induced MC. (Details in Algorithm 2).

Both heuristics implicitly rely on the existence of the α_{noop} action (e.g., a self-loop implemented as part of this work) to define the behavior in unvisited states during the verification step. As part of this work, we implemented the self-loop action as a noop action to allow the verification of the pruned policy σ_{relevant}. However, for non-reachability or minimizing properties, the noop action must be explicitly defined in the PRISM model, as identifying the worst-case action/transition for the agent may be nontrivial. Our experiments (Section 5) show that combining these heuristics, typically by running reachability-based exploration first, yields the best results in practice.



(a) Maximum probability transition method finds a smaller winning set S_{relevant}.



(b) Reachability method finds a smaller winning set S_{relevant}.

Figure 4.2: Comparison of S_{relevant} (solid black) found by the two heuristics for $\lambda = 0.5$. Using the non-optimal heuristic in each case results in $S_{\text{relevant}} \approx S$. Numbers in states represent reachability values $V_{\sigma_{\text{agent}},\sigma_{\text{env}}}(s)$. Self-loops and probabilities for deterministic transitions are omitted for clarity.

Figure 4.2 illustrates why neither heuristic universally dominates the other in finding the smallest sufficient set S_{relevant}. The figure depicts two scenarios where the chosen exploration method leads to different outcomes. In Figure 4.2a (left), prioritizing states reachable via high-probability transitions (Maximum Probability Transition Exploration) identifies a smaller winning sub-MC compared to prioritizing states with high reachability values. Conversely, in Figure 4.2b (right), prioritizing states with high reachability values (Reachability-based Exploration) finds the smaller winning sub-MC. This demonstrates that the optimal exploration strategy depends on the specific structure and probabilities of the underlying Markov chain, justifying the use of both heuristics.

Reachability-Based Exploration

The first heuristic prioritizes exploring states within the induced Markov chain $G^{\sigma_{\text{agent}},\sigma_{\text{env}}}$ that have higher reachability values $V_{\sigma_{\text{agent}},\sigma_{\text{env}}}(s)$. The intuition is that states in the induced Markov chain that are more likely to lead to the target should be included in S_{relevant} first.

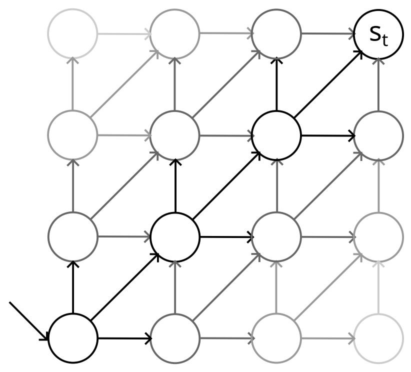
Algorithm 1 Reachability Gradient-based Policy Optimization

```
Require: Original policy $\sigma_{\text{orig}}$, reachability function $V : \mathcal{S} \to \mathbb{R}$, target states $T \subset \mathcal{S}$
Ensure: Policy $\sigma_{\text{relevant}}$
 1: Initialize empty policy $\sigma_{\text{relevant}} \leftarrow \emptyset$
 2: Initialize visited set $\mathcal{V} \leftarrow \emptyset$
 3: Get initial state $s_0 \in \mathcal{S}$
 4: Initialize priority queue $\mathcal{Q} \leftarrow \{s_0\}$        ▷ Prioritized by $V(s)$ in descending order
 5: while $\mathcal{Q} \neq \emptyset$ do
 6:     Extract state $s \leftarrow \arg\max_{s \in \mathcal{Q}} V(s)$
 7:     Remove $s$ from $\mathcal{Q}$
 8:     if $s \in \mathcal{V}$ then
 9:         continue                                            ▷ Skip already visited states
10:     Copy action $\sigma_{\text{relevant}}(s) \leftarrow \sigma_{\text{orig}}(s)$
11:     Mark state as visited $\mathcal{V} \leftarrow \mathcal{V} \cup \{s\}$
12:     for $s' \in \text{Succ}(s, \sigma_{\text{orig}}(s))$ do
13:         if $T \nsubseteq \mathcal{V}$ then
14:             Insert $s'$ into $\mathcal{Q}$ with priority $V(s')$
15:         else
16:             Append $s'$ into $\mathcal{Q}$
17:     if $T \subseteq \mathcal{V}$ and IsCheckpoint() then
18:         if VerifyPolicy($\sigma_{\text{relevant}}$) then
19:             return $\sigma_{\text{relevant}}$                   ▷ Early termination if criteria met
20: return $\sigma_{\text{relevant}}$
```

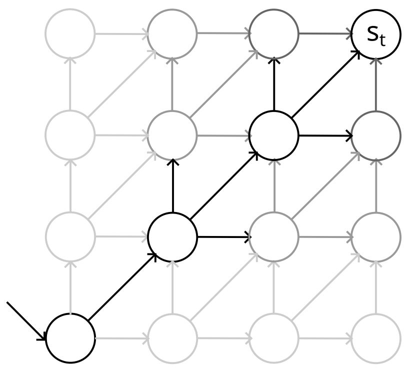
Algorithm 1 implements this heuristic using a priority queue ordered by reachability values in descending order. The exploration starts from s_0. When a state s is extracted, its action $\sigma_{\text{agent}}(s)$ is added to the partial policy σ_{relevant}, and its successors under $\sigma_{\text{agent}}(s)$ are added to the queue if not already visited.

To optimize performance, the algorithm verifies the current partial policy σ_{relevant} (using α_{noop} for unvisited states) against the property φ only periodically and only after at least one target state T has been visited ($\mathcal{V}$), see Line 17. If verification succeeds, the algorithm terminates early, returning the current sufficient $S_{\text{relevant}} = \mathcal{V}$. The frequency of verification balances computational cost against the potential for earlier termination.

A key feature of this algorithm is the shift in exploration strategy once the target set T has been reached (i.e., $T \subseteq \mathcal{V}$). Initially, before any path to T is secured in $\mathcal{V}$, the priority queue strictly follows reachability values $V(s)$, resembling a best-first search focused on efficiently finding paths to T. However, once T is included in $\mathcal{V}$, continuing to strictly prioritize states with the highest individual $V(s)$ values might lead to excessive exploration around already identified high-reachability paths leading to T. This could delay the inclusion of other states that, while having lower individual $V(s)$ values (i.e., being further from T), are crucial for accumulating sufficient overall probability mass from the initial state s_0 to satisfy the global property $\varphi = P_{\geq\lambda}[\Diamond T]$. Therefore, after T is reached, successors are added without strict priority ordering (Lines 13–16), transitioning towards a broader, breadth-first-like exploration of the remaining state space. This aims to ensure that a diverse set of states, collectively contributing to meeting the probability threshold λ



(a) Exploration with strategy switch. Demonstrates a broader search for global property satisfaction.



(b) Exploration without strategy switch. Demonstrates over-focusing on local optima.

Figure 4.3: Conceptual illustration of exploration strategies on an example MC, highlighting the benefit of a strategy switch after target discovery. States are colored by exploration order (darker is earlier), transition probabilities are assumed equal, and omitted for clarity.

from s_0, is identified efficiently, rather than over-focusing on states that are merely locally optimal for reaching an already-found target.

Figure 4.3 illustrates this strategic shift. Figure 4.3a depicts the exploration pattern with the strategy switch. After reaching T (darker states), the exploration broadens (medium grey states), ensuring that various contributing paths are considered. This can lead to a sufficient S_{relevant} more efficiently by not getting trapped exploring only the vicinity of the first-found target state. In contrast, Figure 4.3b shows exploration without the switch. Here, after finding initial paths to T, the algorithm might continue to explore states with high individual reachability to T (more dark/medium grey states concentrated near already included target state). This risks neglecting other states (lighter grey) that are needed to secure enough overall probability from s_0 to satisfy the property φ. These essential lower-value states might be significantly delayed if the search remains too narrowly focused on local reachability to target maxima after T is initially found.

Maximum Probability Transition Exploration

Our second heuristic prioritizes exploring transitions that have the highest probability, given by $P(s, \sigma_{\text{agent}}(s), s')$. The intuition here is to follow the most likely execution paths within the Markov chain induced by $(\sigma_{\text{agent}}, \sigma_{\text{env}})$.

Algorithm 2 implements this using a priority queue ordered by the transition probability leading into the state being added, Line 14. When a state s is extracted, its action $\sigma_{\text{agent}}(s)$ is added to σ_{relevant}, and its successors s' are added to the queue, prioritized by the probability $P(s, \sigma_{\text{agent}}(s), s')$. Similar to the reachability-based heuristic, verification for early termination occurs periodically after T is visited.

Unlike the reachability-based approach, this method does not explicitly switch exploration strategy. The focus remains on exploring paths that are high-probability throughout the process. This is because the maximum transition probability metric naturally maintains exploration focus even after reaching the target, as it approximates reachability from the start state rather than distance to the goal state.

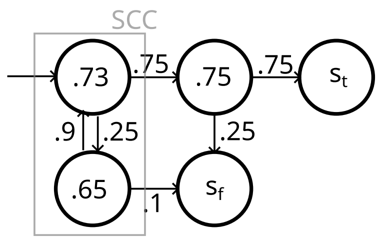
Algorithm 2 Probability-driven Policy Optimization

```
Require: Original policy $\sigma_{\text{orig}}$, transition probability matrix $P : \mathcal{S} \times \mathcal{S} \to [0, 1]$, target states $T \subset \mathcal{S}$
Ensure: Policy $\sigma_{\text{relevant}}$
 1: Initialize empty policy $\sigma_{\text{relevant}} \leftarrow \emptyset$
 2: Initialize visited set $\mathcal{V} \leftarrow \emptyset$
 3: $s_0 \leftarrow$ initial state
 4: Initialize priority queue $\mathcal{Q} \leftarrow \{s_0\}$          ▷ Prioritized by max transition probability
 5: while $\mathcal{Q} \neq \emptyset$ do
 6:     $s \leftarrow \arg\max_{s \in \mathcal{Q}, s' \in \mathcal{S}} P(s', s)$          ▷ Get highest probability transition
 7:     Remove $s$ from $\mathcal{Q}$
 8:     if $s \in \mathcal{V}$ then
 9:         continue
10:     $\sigma_{\text{relevant}}(s) \leftarrow \sigma_{\text{orig}}(s)$
11:     $\mathcal{V} \leftarrow \mathcal{V} \cup \{s\}$
12:     for $s' \in \text{Succ}(s, \sigma_{\text{orig}}(s))$ do
13:         if $s' \notin \mathcal{V}$ then
14:             Insert $s'$ into $\mathcal{Q}$ with priority $P(s, s')$
15:     if $T \subseteq \mathcal{V}$ and IsCheckpoint() then
16:         if VERIFYPOLICY($\sigma_{\text{relevant}}$) then
17:             return $\sigma_{\text{relevant}}$
18: return $\sigma_{\text{relevant}}$
```

Incremental Steady State Analysis

The heuristic pruning algorithms rely on periodically verifying if the currently constructed partial policy σ_{relevant} satisfies the property φ (Line 16). In our implementation, this verification is performed by invoking an external model checker (Storm) on the Markov chain induced by $(\sigma_{\text{relevant}}, \sigma_{\text{env}})$, where undefined actions in σ_{relevant} are treated as α_{noop}.

An alternative theoretical approach involves incrementally computing the reachability probabilities within the partially constructed Markov chain. Consider the example MC induced by some $(\sigma_{\text{agent}}, \sigma_{\text{env}})$ shown in Figure 4.4. The reachability probability $V(s)$ for each state s (probability to reach $T = S_t$) can be described by a system of linear equations based on the Bellman equation for reachability [9]:



$$
\begin{aligned}
S_0 &= 0.75 \cdot S_1 + 0.25 \cdot S_2\\
S_1 &= 0.75 \cdot S_t + 0.25 \cdot S_f\\
S_2 &= 0.9 \cdot S_2 + 0.1\\
S_t &= 1\\
S_f &= 0
\end{aligned}
$$

Figure 4.4: Example Markov chain featuring a Strongly Connected Component (left), and its corresponding system of linear equations for computing reachability probabilities (right).

Solving this system yields the reachability probabilities for all states. Now, consider the pruning process starting with $S_{\text{relevant}} = \{S_0, S_1, S_t\}$. We only evaluate transitions within this subset. The initial computation yields $S_0 = 0.75 \cdot 0.75 = 0.5625$, but for $\lambda = 0.6$, this partial policy is insufficient.

Exploring additional states, such as S_2, increases the complexity due to the formation of a Strongly Connected Component (SCC) between S_0 and S_2. In this case, the equation gives $S_2 = 0.9 \cdot 0.5625 = 0.50625$. This change propagates back to S_0, requiring its recalculation as $S_0 = 0.75 \cdot 0.75 + 0.25 \cdot 0.50625 = 0.689$.

The presence of SCCs complicates simple incremental updates. Changes within an SCC require solving a system of equations for all states in that component. In contrast, adding states that only extend acyclic paths allows for straightforward local updates. This can be demonstrated with $S_{\text{relevant}} = \{S_t\}$. If we subsequently explore S_1, we can compute $S_1 = 0.75 \cdot 1 = 0.75$. This value is stable and corresponds exactly to the solution in the complete Markov chain.

While theoretically feasible, implementing efficient incremental reachability verification, especially handling SCCs dynamically, adds significant complexity. Given that our empirical evaluation shows external model checking with Storm performs sufficiently well for the benchmarked scenarios, this incremental approach was not pursued in our final implementation. However, it remains a potentially valuable optimization for scenarios with very large state spaces dominated by acyclic structures, or where repeated model checker invocations become a bottleneck.

4.3 Merging Strategies

The original work [5] introduced a post-processing step to reduce policy tree size by merging leaves corresponding to compatible subfamilies. Two subfamilies $\mathcal{M}_i, \mathcal{M}_j$ with respective winning policies σ_i, σ_j were considered mergeable if a combined policy could be found that was winning for the union $\mathcal{M}_i \cup \mathcal{M}_j$. The algorithm checked two candidate policies:

- $\sigma_{i \oplus j}$: Prefers actions from σ_i and uses σ_j only for states undefined in σ_i.
- $\sigma_{j \oplus i}$: Prefers actions from σ_j and uses σ_i only for states undefined in σ_j.

If either candidate was verified as winning for $\mathcal{M}_i \cup \mathcal{M}_j$, the families were merged using that candidate policy.

However, our analysis revealed a potential correctness issue stemming from the original algorithm's greedy approach, which permitted multiple merges involving the same family within a single pass. This lack of sequential validation could lead to transitive compatibility violations. For example, consider families $\mathcal{M}_1, \mathcal{M}_2, \mathcal{M}_3$ with policies $\sigma_1, \sigma_2, \sigma_3$. If, in one pass, $\mathcal{M}_1$ is found compatible with $\mathcal{M}_2$ (using $\sigma_{1\oplus 2}$) and later compatible with $\mathcal{M}_3$ (using $\sigma_{(1\oplus 2)\oplus 3}$), the algorithm might implicitly create a merged policy for $\mathcal{M}_1 \cup \mathcal{M}_2 \cup \mathcal{M}_3$ based on these pairwise checks. However, the validity of the resulting combined policy $\sigma_{(1\oplus 2)\oplus 3}$ for all three families was not explicitly guaranteed for $\mathcal{M}_2$.

To address this inconsistency and ensure the soundness of the merged policy tree, we propose a refined merging strategy with two key modifications:

1. **Single-merge constraint**: In each pass of the merging process, only a single compatible pair of families is merged.
2. **Smallest policy selection**: Together with current candidate policies $\sigma_{i\oplus j}$ and $\sigma_{j\oplus i}$, we also consider the original policies σ_i and σ_j. If multiple candidates are valid, the

smallest (in terms of state-action pairs) policy is selected. This size-based selection was not an explicit criterion in the original work [5], but it is important for our approach, which aims to minimize the overall representation size.

This refined strategy guarantees the correctness of the merged policies by avoiding transitive compatibility violations. While potentially requiring more iterations than the original greedy approach, it ensures the soundness of the final policy tree and explicitly prefers smaller policy representations. This makes it particularly relevant when dealing with pruned policies of varying sizes, and accordingly, this enhanced merging strategy is adopted for all experiments involving policy trees.

4.4 Deriving a Markov Decision Process

We now present the second orthogonal approach for optimizing policy representations: Transforming the f-MDP synthesis problem back into a standard MDP problem to leverage state-of-the-art MDP synthesis tools. The core idea is to fix the environment's behavior based on its optimal strategy derived from the game abstraction, thereby creating an MDP for which alternative policy synthesis algorithms can be applied. We specifically utilize dtNESt [3] for this purpose.

Recall from Section 3.2 that the game abstraction $G_{\mathcal{M}}$ models the f-MDP problems as a two-player game. Player 1 (agent) chooses an action α, and Player 2 (environment) subsequently chooses the MDP dynamics $I \in I^{\sim}_{s,\alpha}$ to minimize the agent's objective. Solving this game yields optimal policies $(\sigma_{\text{agent}}, \sigma_{\text{env}})$ (also referred to as σ_1, σ_2). The resulting σ_{agent} is robust winning but potentially conservative and unoptimized for representation size.

Instead of directly using σ_{agent}, a derived MDP can be constructed. Let's call it $M^{\sigma_{\text{env}}}$, by resolving Player 2's choices in the game abstraction with its optimal strategy σ_{env}.

Definition 12 (Derived MDP $M^{\sigma_{\text{env}}}$)**.** *Let $\mathcal{M} = \{M_i\}_{i \in I}$ be the family and $G_{\mathcal{M}}$ its game abstraction with optimal policies $(\sigma_{agent}, \sigma_{env})$. The derived MDP is defined as $M^{\sigma_{env}} = (S_{\mathcal{M}}, s_0, \mathrm{Act}_{\mathcal{M}}, P^{\sigma_{env}})$, where:*

- *The state space $S_{\mathcal{M}}$ and initial state s_0 are the same as in the original family members.*
- *The action space $\mathrm{Act}_{\mathcal{M}}$ contains only the agent's actions (Player 1's actions in the game).*
- *The transition function $P^{\sigma_{env}} : S_{\mathcal{M}} \times \mathrm{Act}_{\mathcal{M}} \times S_{\mathcal{M}} \to [0,1]$ is defined by fixing the environment's choice according to σ_{env}. For a state $s \in S_{\mathcal{M}}$ and agent action $\alpha \in \mathrm{Act}_{\mathcal{M}}(s)$, the environment chooses the dynamics $I = \sigma_{env}(s, \alpha)$. The resulting transition probabilities are taken from any M_i where $i \in I$:*

  $$P^{\sigma_{env}}(s, \alpha, s') = P_i(s, \alpha, s') \quad \textit{for any } i \in \sigma_{env}(s, \alpha)$$

  Recall that all $i \in I$ lead to the same $P_i(s, \alpha, \cdot)$ distribution.

Essentially, $M^{\sigma_{\text{env}}}$ represents the dynamics when the agent plays against the fixed, optimal (worst-case) environment strategy σ_{env} derived from the game solution.

Correctness Guarantee

The crucial property of this transformation is that solving the derived MDP $M^{\sigma_{\text{env}}}$ still provides guarantees for the original family $\mathcal{M}$.

Theorem 3 (Robustness via Derived MDP)**.** *Let $\mathcal{M} = \{M_i\}_{i \in I}$ be an f-MDP, $\varphi = P_{\geq \lambda}[\Diamond T]$ a reachability specification, and $M^{\sigma_{env}}$ the MDP derived using the optimal environment strategy σ_{env} from the game abstraction $G_{\mathcal{M}}$. If a policy σ satisfies the specification in the derived MDP, i.e.,*

$$\mathbb{P}[(M^{\sigma_{env}})^{\sigma} \models \Diamond T] \geq \lambda$$

then σ is a robust winning policy for the original family $\mathcal{M}$, i.e.,

$$\forall M_i \in \mathcal{M} : \mathbb{P}[M_i^{\sigma} \models \Diamond T] \geq \lambda$$

Proof Sketch. The optimal environment strategy σ_{env} represents the worst-case choice of dynamics for the agent at each step, considering the agent's potential actions. The value achieved by any agent policy σ in the derived MDP $M^{\sigma_{\text{env}}}$ corresponds to the value achieved by σ against the fixed adversary σ_{env} in the game $G_{\mathcal{M}}$:

$$\mathbb{P}[(M^{\sigma_{\text{env}}})^{\sigma} \models \Diamond T] = \mathbb{P}[G_{\mathcal{M}}^{\sigma,\sigma_{\text{env}}} \models \Diamond T]$$

Since σ_{env} is the optimal (minimizing) strategy for Player 2, the value against σ_{env} provides a lower bound on the value against any other Player 2 strategy, including strategies consistent with a single M_i:

$$\mathbb{P}[G_{\mathcal{M}}^{\sigma,\sigma_{\text{env}}} \models \Diamond T] \leq \mathbb{P}[G_{\mathcal{M}}^{\sigma,\sigma_{\text{env},i}} \models \Diamond T] = \mathbb{P}[M_i^{\sigma} \models \Diamond T]$$

where $\sigma_{\text{env},i}$ is a Player 2 strategy consistent with M_i. Therefore, if $\mathbb{P}[(M^{\sigma_{\text{env}}})^{\sigma} \models \Diamond T] \geq \lambda$, it follows that $\mathbb{P}[M_i^{\sigma} \models \Diamond T] \geq \lambda$ for all $M_i \in \mathcal{M}$. $\square$

Theorem 3 provides the formal justification for using standard MDP solvers on $M^{\sigma_{\text{env}}}$ to find robust policies for $\mathcal{M}$.

Synthesizing Alternative Policies (σ_{alt})

While the original agent policy σ_{agent} (from the game solution) is optimal for maximizing the property value in $M^{\sigma_{\text{env}}}$ (and robustly for $\mathcal{M}$), this policy is typically derived from value iteration or policy iteration on the game structure. These methods do not prioritize the compactness of the policy representation, often resulting in tabular policies or policies derived directly from value functions that lead to large and complex decision trees when converted naively.

Decision tree synthesis tools like dtNESt, however, are specifically designed to find *compact decision tree representations* of policies, often by accepting a small, controllable trade-off in optimality (ϵ-optimality). For a given MDP, dtNESt aims to find a decision tree $\mathcal{T}_{\text{dtNESt}}$ such that the policy induced by it, $\sigma_{\mathcal{T}_{\text{dtNESt}}}$ (as per the definition in Section 2), has a value $V^{\sigma_{\mathcal{T}_{\text{dtNESt}}}}$ close to the optimal value V^* (i.e., $V^{\sigma_{\mathcal{T}_{\text{dtNESt}}}} \geq V^* - \epsilon$) while minimizing the size of the decision tree $\mathcal{T}_{\text{dtNESt}}$.

Therefore, we apply dtNESt to the derived MDP $M^{\sigma_{\text{env}}}$ to synthesize such a compact decision tree, which we denote $\mathcal{T}_{\text{alt}}$. From this tree, we induce an alternative policy, $\sigma_{\text{alt}} = \sigma_{\mathcal{T}_{\text{alt}}}$. This policy σ_{alt} (and its compact representation $\mathcal{T}_{\text{alt}}$) aims to:

1. Achieve a compact and interpretable decision tree representation (the primary goal of using dtNESt).

2. Maintain robust correctness for the family $\mathcal{M}$. This is guaranteed by Theorem 3, provided the value achieved by σ_{alt} in $M^{\sigma_{\text{env}}}$ remains $\geq \lambda$. To ensure this condition, the ϵ parameter of dtNESt must be chosen carefully. If V^* is the optimal value for $M^{\sigma_{\text{env}}}$, then ϵ must be set such that $V^{\sigma_{\text{alt}}} \geq V^* - \epsilon \geq \lambda$. This typically means choosing $\epsilon \leq V^* - \lambda$ (assuming $V^* \geq \lambda$). If $V^* < \lambda$ initially, then no policy, even the optimal one for $M^{\sigma_{\text{env}}}$, can satisfy the constraint, and thus no robust policy satisfying λ can be found for $\mathcal{M}$ via this derived MDP.

This transformation allows us to leverage the strengths of specialized MDP-to-DT synthesis tools like dtNESt to obtain compact, robust policies for the original f-MDP problem, addressing the size and interpretability limitations of directly using σ_{agent}. The resulting policy σ_{alt}, induced from its compact decision tree representation $\mathcal{T}_{\text{alt}}$, can then be used as a candidate policy within the unified tree structure described in Section 4.5.

4.5 Unified Decision Tree Representation

Having discussed methods to obtain potentially more compact policy representations (pruning in Section 4.2 or alternative policy synthesis via MDP transformation in Section 4.4), we now introduce a novel structure to represent the entire solution i.e. mapping subfamilies to policies within a single, optimized tree: the *unified decision tree.*

Definition 13 (Unified Decision Tree (UDT))**.** *A unified decision tree for a family $\mathcal{M} = \{M_i\}_{i \in I}$ over state variables V_{state} and family parameters/identifiers V_{family} is an abstract decision tree $\mathcal{T} = (T, \gamma, \delta)$ (as defined in Section 2) where:*

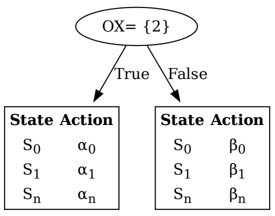
- *γ assigns to each inner node n a predicate $\gamma(n)$ of one of two types:*
    1. ***Family Predicate**: A condition on the family parameters/identifiers V_{family} (e.g., $param \leq c$; $id \in \{i_1, i_2\}$). These predicates partition the family $\mathcal{M}$.*
    2. ***State Predicate**: A condition on the state variables V_{state} in the form $v \leq c$. These predicates partition the state space $S_{\mathcal{M}}$.*
- *δ assigns to each leaf node an action $\alpha \in \text{Act}_{\mathcal{M}}$ or the symbol $\emptyset$ (indicating unsatisfiability for the corresponding subfamily and state partition).*

The UDT interleaves decisions based on family parameters/identifiers and decisions based on the current state, mapping a specific MDP M_i and a state s directly to an action $\alpha = \mathcal{T}(M_i, s)$ or $\emptyset$.

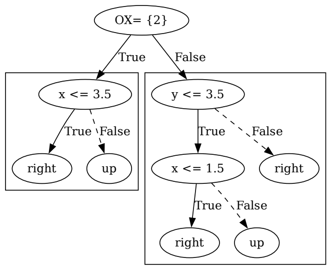
Construction Approaches

Realizing the benefits of the UDT hinges on its effective construction. We will now examine different strategies for building such a unified structure, beginning with a straightforward, albeit potentially redundant, approach and then detailing our proposed optimized method designed for enhanced compactness and efficiency.

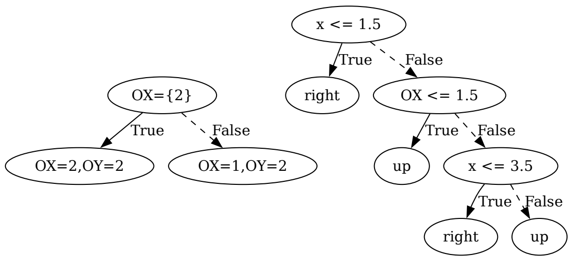
Trivial Construction (Policy Tree + DTs): A UDT can be trivially constructed by taking an existing policy tree (Section 2.2) and replacing each leaf node ℓ (representing



(a) Policy Tree (PT) with tabular leaf policies.



(b) Trivial UDT: PT with individual DT leaf policies.



(c) Proposed UDT: candidate mapping tree (left) + unified policy tree (right).

Figure 4.5: Comparison of controller representations for MDP families.

subfamily $\mathcal{M}_\ell$ with policy σ_ℓ) with the root of a decision tree $\mathcal{T}_\ell$ that implements σ_ℓ. This approach is illustrated conceptually in Figure 4.5b. While straightforward, this often leads to significant redundancy as identical subtrees may appear multiple times across different $\mathcal{T}_\ell$.

Optimized Construction using dtControl (Proposed): To overcome the limitations of the trivial construction and achieve a more compact UDT, we propose leveraging a decision tree learning tool like dtControl. The challenge is to formulate the problem such that dtControl can learn a single tree representing the complex mapping $(M_i, s) \mapsto \alpha$.

We treat this as a classification problem. The input features would ideally include both the state variables V_{state} and the family parameters/identifiers V_{family}. The target classes are the actions $\mathrm{Act}_{\mathcal{M}} \cup \{\emptyset\}$. However, a naive approach requiring training data for every state $s \in S_{\mathcal{M}}$ and every MDP $M_i \in \mathcal{M}$ is computationally infeasible due to the potentially vast number of (M_i, s) combinations.

Instead, we employ a *two-tree representation* (illustrated in Figure 4.5c) that decouples family mapping from state-based decisions, making the learning task tractable:

1. **Candidate Mapping Tree (CMT)**: This tree retains the structure of the original policy tree obtained from the existing approach (potentially refined by our merging strategy, Section 4.3). However, instead of mapping to full policies σ_ℓ, its leaves map to candidate identifiers c_ℓ. Each c_ℓ represents a distinct policy required by the overall solution. Let this mapping be $CMT(M_i) = c_\ell$.

2. **Unified Policy Tree (UPT)**: This is the main decision tree learned by a tool like dtControl. Its input features include the state variables V_{state} and the candidate identifier c_ℓ (treated as an additional categorical feature). The UPT is learned using

input-output mappings of $((s, c_\ell), \alpha)$, where $\alpha = \sigma_\ell(s)$ is the action prescribed by the policy σ_ℓ associated with candidate c_ℓ for state s.

The final action for a given MDP M_i and state s is determined by first finding the candidate identifier $c_\ell = CMT(M_i)$ and then evaluating the action $\alpha = UPT(s, c_\ell)$.

This two-tree approach significantly reduces the complexity for the decision tree learning tool (dtControl):

- The UPT only needs to distinguish between different policies (represented by c_ℓ) rather than all individual MDPs M_i. The number of distinct policies is typically much smaller than the number of MDPs ($|\{c_\ell\}| \ll |I|$).
- dtControl can effectively find redundancies and shared logic across the different policies σ_ℓ when learning the UPT, leading to a more compact representation.

Although this approach introduces a two-tree structure, the combined size (CMT nodes + UPT nodes) is often significantly smaller than the trivially constructed UDT or the sum of individual policy DTs, as demonstrated later in our experiments.

Chapter 5

Experimental Evaluation

In this chapter, we present a comprehensive empirical evaluation of our proposed methods for synthesizing and optimizing controllers for families of MDPs (f-MDPs). We systematically compare our approaches against the state-of-the-art game abstraction technique [5], focusing on the effectiveness of the enhancements introduced in Chapter 4: the refined policy merging strategy (Section 4.3), the policy state pruning heuristics (Section 4.2), the transformation to a derived MDP for alternative policy synthesis using dtNESt (Section 4.4), and the unified decision tree representation (Section 4.5). Our evaluation focuses on answering the following key questions:

- Q1: What are the effects of the refined merging strategy compared to the original approach?
- Q2: How does policy state pruning affect the size of the final unified decision tree (UDT) representation?
- Q3: What are the effects of using the MDP transformation approach with dtNESt on the size of the UDT representation?
- Q4: What controller size improvements does the proposed UDT offer compared to the baseline (individual DTs for policy tree leaves)?

Experimental Setting: Our methodologies are built as an extension of the PAYNT tool [4] and integrate with dtControl [7] for decision tree synthesis from policies, as well as the dtNESt component [3] for synthesizing alternative policies from derived MDPs. dtNESt, in turn, employs the Z3 SMT solver [18] as its internal constraint-solving engine. All experiments were conducted on a machine equipped with an Intel Core i7-1280P processor (20 cores) and 32GB of RAM. Each benchmark problem was allocated a single CPU core with a minimum timeout threshold of 4 hours.

Benchmarks: Our evaluation utilized publicly available benchmarks initially used in the original work on game-based abstraction for f-MDPs [5].[1] These benchmarks required a slight modification to ensure compatibility with our enhanced framework, particularly for the dtNESt component. Original problems utilize constraint-based properties, while dtNESt requires properties in a maximization format to find an ϵ-optimal policy. To address this, we created alternative property specifications with the suffix "_alt.prop" for each benchmark, transforming the constraint $\mathbb{P}_{\geq\lambda}[\dots]$ into a maximization objective $\mathbb{P}_{\max=?}[\dots]$.

[1] All benchmarks are available at `https://github.com/randriu/synthesis` in `models`

Verification of Correctness: Ensuring that a synthesized policy (or set of policies) satisfies the specification for every member of an f-MDP is crucial. Verifying this by testing the policy against each individual MDP instance is computationally intractable for large families. Therefore, for policies associated with subfamilies in a policy tree (generated by the game abstraction, potentially with merging or pruning), we rely on the verification mechanism inherent in the game-based synthesis framework [5]. A policy σ_{agent} for a subfamily $\mathcal{M}' \subseteq \mathcal{M}$ is verified as correct by fixing Player 1's strategy to σ_{agent} (thereby defining an MDP where Player 2, the environment, makes choices constrained by $\mathcal{M}'$), and then ensuring that Player 2, playing optimally to minimize the property's value, cannot force this value below the threshold λ. This process is effectively equivalent to checking the property in the Markov Chain induced by both Player 1's strategy (σ_{agent}) and Player 2's optimal counter-strategy (σ_{env}) for $\mathcal{M}'$.

For alternative policies σ_{alt} synthesized using dtNESt from a derived MDP $M^{\sigma_{\text{env}}}$ (Section 4.4), Theorem 3 guarantees robustness for the family $\mathcal{M}$ if σ_{alt} achieves a value $\geq \lambda$ in $M^{\sigma_{\text{env}}}$. In our dtNESt experiments, we use a standard $\epsilon = 0.05$ for ϵ-optimality. This is justified as the optimal value $V^*(M^{\sigma_{\text{env}}})$ typically exceeds the original threshold λ by more than ϵ. Consequently, the ϵ-optimal policy σ_{alt}, satisfying $V((M^{\sigma_{\text{env}}})^{\sigma_{\text{alt}}}) \geq V^*(M^{\sigma_{\text{env}}}) - \epsilon$, is still expected to meet the λ requirement. Occasional minor deviations, where $V^*(M^{\sigma_{\text{env}}}) - \epsilon$ falls slightly below λ, are considered negligible, potentially due to the floating-point numerical precision of the model checker. The property transformation to $\mathbb{P}_{\max=?}[\dots]$ for dtNESt is thus deemed practically equivalent for evaluation.

Throughout this chapter, the symbol † indicates experiments that exceeded either memory constraints or the four-hour timeout threshold, representing cases where the corresponding approach faced scalability challenges.

5.1 Policy Merging Strategy

The game-based synthesis approach [5] can result in a policy tree with numerous distinct policies. Reducing this number is key for a more compact and interpretable controller. The original work [5] included a merging post-processing step, but as discussed in Section 4.3, its greedy nature could lead to soundness issues.

For this evaluation, we employ our *refined merging strategy* to ensure soundness. This enhanced strategy is used in all subsequent experiments. The limitations of the original merging strategy become more pronounced when combined with further optimizations. Consequently, direct comparisons under such advanced settings would not provide a fair or meaningful assessment and are thus considered outside the scope of this work.

Table 5.1 provides a comparison of the original policy merging strategy and our enhanced strategy, applied before any other optimizations like pruning or UDT synthesis. The leftmost columns characterize each benchmark: $|\mathcal{M}|$ denotes the total number of family members, $|S_M|$ signifies the number of states, and $\sum$ Act represents the total number of actions in the quotient MDP $Q_{\mathcal{M}}$. *SAT%* indicates the percentage of MDPs in the family for which a winning policy exists with respect to the original property specification. Columns labeled *Original* correspond to the PAYNT framework's original algorithm [5], while *Enhanced merging* corresponds to our refined strategy. The table evaluates these strategies based on synthesis time, the number of distinct policies at the policy tree leaves (*#Pols*), and the total policy tree nodes (*#Nodes*).

Table 5.1: Comparison of the original policy merging strategy [5] and our enhanced strategy (Section 4.3).

Model	Model Information				Original			Enhanced merging		
	$\lvert S_M \rvert$	$\sum$ Act	$\lvert \mathcal{M} \rvert$	SAT%	Time (s)	#Pol	#Node	Time (s)	#Pol	#Node
av-8-2	2e4	9e4	4e3	95	208	27	460	340	**16**	**433**
av-8-2-e	2e4	9e4	4e3	100	8	3	4	8	**1**	**1**
dodge-2	2e5	7e5	3e4	100	575	29	44	1e3	**19**	**31**
dodge-3[2]	2e5	7e5	9e7	100	†	†	†	†	†	†
dpm-10	2e3	1e4	2e4	18	37	**97**	1e3	59	106	1e3
dpm-10-b	9e3	1e5	1e5	23	425	**6**	89	789	12	89
obs-8-6	5e2	9e2	5e4	90	13	**267**	1e3	27	372	1e3
obs-10-6	8e2	3e3	3e6	98	6	**90**	**301**	12	169	364
obs-10-9	1e3	2e3	4e8	100	606	**3e3**	7e3	1e3	4e3	7e3
rov-100	2e3	5e4	2e7	47	2e3	246	9e4	2e3	246	9e4
rov-1000	2e4	5e5	4e6	100	3e3	3e3	2e4	4e3	3e3	2e4
uav-roz	2e4	2e5	5e3	99	40	2	124	87	2	124
uav-work	9e3	1e5	2e6	100	263	7	3e3	512	7	3e3
virus	2e3	1e5	7e4	83	770	541	7e3	1e3	**537**	7e3
rocks-6-4	3e3	7e3	7e3	100	540	3e3	**6e3**	974	**2e3**	7e3

The results in Table 5.1 indicate that our enhanced merging strategy often achieves a comparable or reduced number of distinct policies (*#Pol*) and policy tree nodes (*#Node*) when compared to the original approach. For instance, benchmarks like *av-8-2-e* and *dodge-2* show a notable reduction in both policy count and tree size. However, the original greedy strategy sometimes produces a smaller number of policies or nodes, as seen in benchmarks like *obs-10-6* or *dpm-10-b*. While the synthesis time for the enhanced strategy is sometimes higher due to the more rigorous, sound verification process, the resulting policy sets provide a more reliable foundation for subsequent optimizations, which is the primary motivation for its adoption.

It is worth noting that in some benchmarks, such as *uav-roz*, the policy tree structure appears disproportionately large compared to the number of contained policies. This phenomenon occurs because the family structure may require complex decision-making pathways to differentiate between subfamilies that ultimately map to the same policy, as illustrated in Figure 2.1. This structural complexity represents an inherent characteristic of particular problem domains rather than a limitation of our merging strategy.

Q1: Effects of the new merging strategy on policy trees? The empirical results demonstrate that our enhanced merging strategy generally maintains or improves compactness compared to the original approach. While the sounder verification process can increase synthesis time, this trade-off is justified by the creation of a reduced set of distinct policies, which is crucial for subsequent optimizations. Therefore, this enhanced strategy is utilized for all the following experiments.

5.2 Policy Pruning

We now evaluate the effects of policy state pruning (Section 4.2) on the final controller representation. This analysis focuses on the size of the Unified Decision Tree (UDT), particularly its Unified Policy Tree (UPT) component, as depicted in Figure 4.5c. Our heuristic pruning techniques (Algorithms 1 and 2) aim to reduce policy complexity by removing irrelevant

[2] *dodge-3* is not included in any other results as it didn't terminate due to memory constraints (even for original approach)

Table 5.2: Policy pruning effects on UDT size (#Node) and total policy actions (Act). Comparison focus is Unpruned UPT vs 100%.

Model	Unpruned UPT			100%				90%			80%		
	SAT%	#Node	Act	SAT%	#Node	Act	$\Delta t(s)$	SAT%	#Node	Act	SAT%	#Node	Act
av-8-2	95	6e3	6e4	†	†	†	†	†	†	†	100	867	7e3
av-8-2-e	100	203	4e3	100	**39**	**879**	446	100	33	880	100	37	880
dodge-2	100	9e4	1e6	†	†	†	†	†	†	†	†	†	†
dpm-10	18	2e3	4e4	18	2e3	**7e3**	125	67	4e3	2e4	98	659	3e4
dpm-10-b	23	247	2e4	23	**149**	**1e3**	442	†	†	†	†	†	†
obs-8-6	90	3e3	2e4	90	**853**	**5e3**	129	90	727	4e3	98	611	3e3
obs-10-6	98	2e3	2e4	98	**401**	**3e3**	71	98	263	1e3	100	211	1e3
obs-10-9	100	2e4	4e5	100	**7e3**	**6e4**	1e3	100	7e3	6e4	100	325	1e3
rov-100	47	3e3	2e4	47	**585**	**2e3**	2e3	†	†	†	†	†	†
rov-1000	†	†	†	†	†	†	†	100	1	1e3	100	1	1e3
uav-roz	99	21	1e4	99	**11**	**1e3**	1e4	100	21	9e3	100	21	9e3
uav-work	100	41	2e4	†	†	†	†	100	25	3e3	100	25	2e3
virus	83	7e3	2e5	83	**3e3**	**4e4**	1e4	83	3e3	3e4	83	2e3	3e4
rocks-6-4	100	2e4	4e5	100	2e4	**9e4**	3e3	100	2e4	9e4	100	2e4	8e4

states while maintaining correctness guarantees. The Candidate Mapping Tree (CMT) structure is largely determined by the family partitioning, remaining relatively unaffected by state pruning within the policies themselves. Consequently, this analysis of pruning effects focuses on the UPT component, and any impact on the CMT size is considered negligible and not the primary subject of this specific comparison.

Table 5.2 presents comprehensive results of applying our pruning heuristics. We compare four configurations:

- **Unpruned UPT**: Represents the UPT synthesized without any pruning applied.
- **100%**: Pruning preserves the original property specification λ as per Section 4.2.
- **90%**: Allowing pruned policies to satisfy a relaxed property with threshold $0.9 \cdot \lambda$.
- **80%**: Allowing a threshold of $0.8 \cdot \lambda$.

Measuring relaxed properties allows us to evaluate the trade-off between policy size and the satisfaction threshold. When comparing the *100%* pruning configuration against the *Unpruned UPT*, the better values are highlighted in bold. The *SAT%* column indicates the percentage of MDPs in the family for which a winning policy exists. *#Node* denotes the number of nodes in the UPT component. The *Act* sums the number of state-action pairs defined across all distinct policies, giving a measure of raw policy complexity. Finally, $\Delta t(s)$ represents the additional synthesis time for the 100% pruning configuration compared to the unpruned UPT. The synthesis time for the Unpruned UPT, which serves as the baseline for $\Delta t(s)$, is given by the *Enhanced merging* times presented in Table 5.1.

Interestingly, pruning does not universally improve results across all benchmarks. We observe two principal situations where pruning may produce unintuitive outcomes:

First, property relaxation can substantially increase the percentage of satisfiable families within the total model space. This is evident in the *dpm-10* model, where relaxing from 100% to 90% increases the number of satisfiable families. This dramatic expansion of the solution space necessitates encoding significantly more families and information in the unified tree, increasing its complexity rather than reducing it. Furthermore, there are many cases where further relaxation rendered the problem unsolvable (†) due to the same increase in satisfiable families. On the contrary, some problems become solvable with further relaxation (e.g. *av-8-2*).

Second, the intrinsic limitations of our heuristic-based pruning approach become apparent in examples like the *av-8-2-e* model. Here, despite maintaining 100% satisfiability across all pruning levels, we observe variations in encoding complexity. This highlights a fundamental challenge. Our pruning strategy optimizes for minimal individual policies without explicitly optimizing for the structure of the unified tree. Consequently, even when individual policies become smaller through pruning, their collective representation in the unified tree may become more complex. Lastly, these pruned policies may not encode as many families as the original approach, potentially requiring additional policies to be retained in the unified tree.

Q2: effect of pruning on unified decision tree size? Despite the aforementioned complexities and limitations, our results demonstrate a generally positive effect of pruning (at 100% property preservation) on reducing both the raw policy action count and the final UPT size for many benchmarks. For most benchmarks, the pruning approach yields notably smaller trees while preserving or improving family satisfiability. As we increase property relaxation, we observe two notable trends. In many cases, such as *obs-8-6* and *obs-10-6*, further relaxation continues to reduce tree size. However, in other cases, such as *av-8-2-e*, relaxation beyond 100% yields minimal additional benefits or even slight degradation in tree structure. The computational overhead of pruning, which increased synthesis time on average 20-fold (with a median increase of 4-fold, influenced by outliers), is often justified by the resulting compactness, but for large or complex families, it may lead to timeouts †.

5.3 State of the Art Comparison

In this final evaluation, we compare our proposed methodologies against alternative approaches for synthesizing controllers for f-MDPs. This comparison provides insights into the relative strengths and limitations of each technique. We evaluate the following configurations:

- **Baseline**: The standard policy tree structure where each leaf policy is synthesized individually as a decision tree using dtControl. Corresponds conceptually to Figure 4.5b.
- **dtPaynt**: Similar to the baseline, but dtPaynt [6] is used instead of dtControl for synthesizing the decision trees.
- **Unified Tree (UT)**: Our proposed two-tree representation (CMT + UPT) as described in Section 4.5. The UPT is synthesized using dtControl based on the unpruned policies from the game abstraction. Corresponds to the *Unpruned UPT* configuration in Table 5.2 and conceptually to Figure 4.5c.
- **UT+Pruning**: Our proposed two-tree representation, where individual policies are pre-processed using our pruning heuristics (Section 4.2). Corresponds to the *100%* configuration in Table 5.2.
- **UT+dtNESt**: Our proposed two-tree representation, where individual policies are pre-processed using the MDP transformation approach (Section 4.4) with dtNESt.

- **UT+dtNESt+Pruning**: Our proposed two-tree representation, where individual policies are pre-processed using both the MDP transformation approach (Section 4.4) with dtNESt and our pruning heuristics (Section 4.2).

These different structural approaches are illustrated conceptually in Figure 4.5. We evaluate these approaches using computational time (t), representation size (*Size*), and policy count (*#Pol*). Representation size is the total number of nodes in the resulting tree structure(s). For the Unified Tree approaches (UT, Pruning, dtNESt), we report size as $x + y$, where x is the CMT size and y is the UPT size. Policy count includes all distinct policies required, including $\emptyset$ for unsatisfiable subfamilies.

A significant challenge arose when attempting to include *dtPaynt* in the quantitative comparison. Despite extended runtimes, this approach consistently failed to terminate for any benchmark within the four-hour limit. Qualitative analysis of partial results suggested its outputs were generally comparable to or larger than those from baseline for these benchmarks. This might be attributed to dtPaynt's focus on minimizing tree depth via SMT, which can be computationally expensive and may not lead to the smallest overall tree size. This contrasts with findings in [6] on different benchmarks, suggesting the relative performance depends heavily on the problem characteristics. Due to the lack of complete results, this SMT-based approach is omitted from the comparison.

Next, the combined approach of *UT+dtNESt+Pruning* was explored preliminarily. Applying pruning before dtNESt did not consistently yield further significant benefits over dtNESt alone. Suggesting the specific states removed by our pruning heuristics do not significantly simplify the task for dtNESt's synthesis algorithm. Therefore, it was also not included in the final comparison. The comprehensive comparison of the remaining approaches, based on the metrics defined above, is detailed in Table 5.3.

Table 5.3: Benchmark comparison of different methods for synthesizing controllers for families of MDPs.

Model[3]	Baseline			Unified tree (UT)			UT+Pruning			UT+dtNESt		
	Time (t)	Size	#Pol	t	Size	#Pol	t	Size	#Pol	t	Size	#Pol
av-8-2	405	**2e3**	16	**297**	433+6e3	16	†	†	†	†	†	†
av-8-2-e	17	204	1	**11**	1+203	1	457	**1+39**	1	3e4	1+117	1
dodge-2	1e3	2e5	19	1e3	**31+9e4**	19	†	†	†	†	†	†
dpm-10	568	7e3	106	**56**	1e3+2e3	106	181	1e3+2e3	**77**	377	**1e3+5**	106
dpm-10-b	**830**	502	12	943	89+247	12	1e3	89+149	**10**	3e4	**89+107**	12
obs-8-6	2e3	2e4	372	**17**	1e3+3e3	372	140	**1e3+853**	**182**	3e4	1e3+5e3	372
obs-10-6	715	1e4	169	**11**	364+2e3	169	88	**346+401**	**73**	2e4	364+2e3	169
obs-10-9	1e3	3e5	4e3	1e3	7e3+2e4	4e3	2e3	7e3+7e3	**2e3**	3e4	**7e3+3e3**	4e3
rov-100	4e3	1e5	246	**3e3**	9e4+3e3	246	5e3	**9e4+585**	**28**	†	†	†
uav-roz	**999**	145	2	2e3	124+21	2	1e4	**124+11**	2	†	†	†
uav-work	2e3	3e3	7	**838**	3e3+41	7	†	†	†	†	†	†
virus	2e3	6e4	527	**1e3**	7e3+7e3	537	1e4	**7e3+3e3**	**327**	†	†	†
rocks-6-4	6e3	2e5	2e3	**912**	**3e3+2e4**	2e3	4e3	5e3+2e4	2e3	†	†	†

Our empirical results reveal several significant patterns. The Unified Tree (UT) approach consistently demonstrates superior computational efficiency compared to the baseline across most benchmarks. This is likely due to invoking dtControl once on a combined dataset for the UPT, rather than multiple times for individual leaf DTs. Exceptions like *dpm-10-b* and *uav-roz* might occur if the overhead of encoding family information (c_ℓ) outweighs the benefits of a single dtControl invocation.

[3] *rov-1000* is not included due to timeout in all approaches

Applying additional techniques (Pruning, dtNESt) generally yields more compact tree representations (smaller *Size*, particularly the UPT component y) compared to the base UT, but at the cost of increased computational time. This often leads to more benchmarks exceeding the time limit (†).

The *UT+dtNESt* approach typically produces the most compact UPTs, showcasing dtNESt's strength in finding alternative policies with small DT representations for the individual policies derived via MDP transformation. However, this comes at a substantial computational cost, often making it the slowest approach. It is important to note that in our experiments, to manage this cost, dtNESt was applied to the minimized policy set obtained after family merging. This differs from the strict pipeline (Figure 4.1) where other enhancements are applied to the larger set of policies existing prior to merging. Consequently, the reported synthesis times for *UT+dtNESt* might be optimistic. If applied strictly according to the pipeline to a larger set of initial policies, the number of timeouts for dtNESt could be higher.

Such an increase in timeouts for dtNESt under a stricter pipeline application would potentially highlight the *UT+Pruning* approach as a more consistently scalable method for achieving significant UPT size reduction within practical time limits. The pruning approach, as evaluated, offers a balance, significantly reducing UPT size compared to the basic UT and is computationally less demanding than *UT+dtNESt*, although it remains slower than the basic UT.

Q3: Impact of MDP transformation with dtNESt on UDT size? Using the MDP transformation with dtNESt reduces the size of the unified policy tree component on average by 8-fold in most benchmarks where it terminates, often resulting in the most compact overall representation. However, this substantial size reduction comes with a significant increase in computational time, making it a trade-off between representation compactness and synthesis efficiency.

Q4: UDT size improvements compared to baseline? The Unified Tree (UT) representation, in most cases, outperforms the baseline in terms of computational time and often yields comparable or smaller total representation as seen in *Size.* The primary advantage comes from dtControl's ability to optimize across policies when learning the single UPT, eliminating redundancies that exist when synthesizing separate DTs for each leaf in the baseline approach.

Chapter 6

Conclusion

This thesis has addressed the significant challenge of efficiently representing and optimizing policies for families of Markov Decision Processes (f-MDPs). We have introduced a comprehensive framework that advances the state of the art by proposing novel policy merging strategies, a method for transforming the f-MDP problem into a derived MDP to leverage advanced MDP synthesis tools, a unified two-tree representation for compact controller synthesis, and effective heuristic-based techniques for policy state pruning. Our refined merging strategy ensures a sounder basis for policy tree construction, the MDP transformation enables the use of tools like dtNESt while preserving robustness, the unified decision tree structure offers a more compact and interpretable representation, and pruning techniques significantly reduce policy complexity.

Algorithmically, our contributions include two complementary heuristics for state pruning, one guided by reachability analysis and the other by maximum transition probabilities. We detailed the construction of a derived MDP from the game abstraction's solution, providing a formal basis for applying single-MDP solvers to the f-MDP problem. Furthermore, we proposed an efficient encoding scheme for the Unified Decision Tree (UDT) through a Candidate Mapping Tree (CMT) and a Unified Policy Tree (UPT), and introduced a robust method for managing self-loop action, which enables aggressive state pruning while maintaining policy correctness guarantees.

Our experimental evaluation rigorously validates the effectiveness of these contributions. The proposed unified decision tree representation (CMT+UPT) consistently yields smaller and more interpretable controllers compared to baseline methods. State pruning significantly reduces the raw complexity of policies. Crucially, our MDP transformation approach allows the successful application of dtNESt, which further shrinks the resulting decision trees, albeit with an increased computational cost. The two-tree UDT strategy, in particular, offers a compelling trade-off between controller compactness and synthesis efficiency, outperforming traditional approaches in most evaluated scenarios.

The results also underscore essential trade-offs inherent in policy optimization. For instance, relaxing property preservation requirements during pruning can lead to more compact trees, but may also increase the number of satisfiable families, potentially expanding the overall representation needed. Similarly, leveraging dtNESt via our MDP transformation, while highly effective at minimizing tree size, can face scalability challenges due to its computational demands. This highlights that employing powerful optimization techniques like dtNESt through our MDP transformation, while computationally intensive, is the preferred strategy when the ultimate compactness and interpretability of the controller are the primary objectives, and longer synthesis times can be accommodated.

Beyond the immediate scope of f-MDPs, the principles and methods developed in this thesis have broader relevance. They can inform approaches in domains such as reinforcement learning, robotics, automated planning, and formal verification, where the synthesis of compact, interpretable, and efficient policies is increasingly vital, especially in safety-critical or resource-constrained environments. The unified decision tree approach, in conjunction with the MDP transformation and intelligent policy simplification techniques, provides a solid foundation for future research and application in complex decision-making systems, contributing to the development of more trustworthy and efficient autonomous agents. The findings emphasize the critical interplay between interpretability, compactness, and efficiency in policy synthesis, paving the way for further advancements in this important research area.

Looking ahead, several avenues for future research emerge from this work. One direction involves exploring more granular policy differentiation: even for subfamilies where a single robust policy exists, further heuristic partitioning of these subfamilies might yield even simpler constituent policies, potentially leading to a more compact unified policy tree when synthesized. Another significant extension would be to adapt the concepts of unified tree representations and robust synthesis to more complex settings, such as Partially Observable Markov Decision Processes (POMDPs) or families thereof, where managing policy complexity and interpretability is even more challenging [2]. Additionally, exploring novel machine learning approaches, beyond the decision tree induction used for UPT synthesis, could offer new ways to optimize the overall UDT. This might include techniques for directly learning the candidate mapping tree structure or for a more integrated, joint synthesis of the CMT and candidate policies feeding into the UPT, potentially unlocking further gains in compactness and efficiency. Finally, while our work establishes formal correctness for the proposed techniques, developing new formal methods or specialized learning algorithms that could provide stronger guarantees on the global optimality or minimality of the resulting unified decision tree representation presents a challenging but rewarding long-term research goal.

Bibliography

[1] Abdulla, P. A.; Sistla, A. P. and Talupur, M. *Handbook of Model Checking.* Springer, 2018. ISBN 978-3-319-10574-1.

[2] ANDRIUSHCHENKO, R.; ALEXANDER, B.; Češka, M.; JUNGES, S.; KATOEN, J. et al. Search and Explore: Symbiotic Policy Synthesis in POMDPs. In: *Computer Aided Verification.* Cham: Springer, 2023, vol. 13966, p. 113–135. Lecture Notes in Computer Science. ISBN 978-3-031-37708-2.

[3] Andriushchenko, R.; Češka, M.; Chakraborty, D.; Junges, S.; Křetínský, J. et al. Symbiotic Local Search for Small Decision Tree Policies in MDPs. In: *Proceedings of the Conference on Uncertainty in Artificial Intelligence.* 2025. UAI '25.

[4] Andriushchenko, R.; Češka, M.; Junges, S.; Katoen, J.-P. and Stupinský Šimon. PAYNT: A Tool for Inductive Synthesis of Probabilistic Programs. In: *International Conference on Computer Aided Verification.* Springer, 2021, vol. 12759, p. 856–869. Lecture Notes in Computer Science. ISBN 978-3-030-81684-1.

[5] Andriushchenko, R.; Češka, M.; Junges, S. and Macák, F. Policies Grow on Trees: Model Checking Families of MDPs. In: *Proceeding of 22nd International Symposium on Automated Technology for Verification and Analysis.* Springer, 2024, p. 51–75. Lecture Notes in Computer Science. ISBN 978-3-031-78749-2.

[6] Andriushchenko, R.; Češka, M.; Junges, S. and Macák, F. Small Decision Trees for MDPs with Deductive Synthesis. In: *Proceedings of the International Conference on Computer Aided Verification (CAV'25).* 2025. Available at: https://arxiv.org/abs/2501.10126.

[7] Ashok, P.; Jackermeier, M.; Jagtap, P.; Křetínský, J.; Weininger, M. et al. DtControl: decision tree learning algorithms for controller representation. In: *Proceedings of the 23rd International Conference on Hybrid Systems: Computation and Control.* New York, NY, USA: Association for Computing Machinery, 2020. HSCC '20. ISBN 9781450370189.

[8] Ashok, P.; Jackermeier, M.; Křetínský, J.; Weinhuber, C.; Weininger, M. et al. DtControl 2.0: Explainable Strategy Representation via Decision Tree Learning Steered by Experts. In: *Tools and Algorithms for the Construction and Analysis of Systems.* Springer International Publishing, 2021, p. 326–345. ISBN 9783030720131.

[9] Bellman, R. *Dynamic Programming.* Dover Publications, 1957. ISBN 978-0-486-42809-3.

[10] Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 1986, C-35, no. 8, p. 677–691. Available at: https://doi.org/10.1109/TC.1986.1676819.

[11] Hallak, A.; Castro, D. D. and Mannor, S. Contextual Markov Decision Processes, 2015. Available at: https://arxiv.org/abs/1502.02259.

[12] Hensel, C.; Junges, S.; Katoen, J.-P.; Quatmann, T. and Volk, M. The probabilistic model checker Storm. *International Journal on Software Tools for Technology Transfer*, august 2022, vol. 24, p. 589–610. ISSN 1433-2779. Available at: https://doi.org/10.1007/s10009-021-00633-z.

[13] Jansen, N.; Humphrey, L.; Tumova, J. and Topcu, U. Structured Synthesis for Probabilistic Systems, 2018. Available at: https://arxiv.org/abs/1807.06106.

[14] Kattenbelt, M.; Kwiatkowska, M.; Norman, G. and Parker, D. Abstraction Refinement for Probabilistic Software. In: *Verification, Model Checking, and Abstract Interpretation*. Springer, January 2009, vol. 5403, p. 182–197. Lecture Notes in Computer Science. ISBN 978-3-540-93899-6.

[15] Kwiatkowska, M.; Norman, G. and Parker, D. PRISM 4.0: Verification of Probabilistic Real-time Systems. In: *Computer Aided Verification: 23rd International Conference, CAV 2011*. Springer, July 2011, vol. 6806, p. 585–591. ISBN 978-3-642-22109-5.

[16] Landwehr, N.; Hall, M. and Frank, E. Logistic Model Trees. In: *Machine Learning: ECML 2003*. Berlin, Heidelberg: Springer, 2003, p. 241–252. ISBN 978-3-540-20121-2.

[17] Loh, W.-Y. Classification and Regression Trees. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, january 2011, vol. 1, p. 14–23. Available at: https://doi.org/10.1002/widm.8.

[18] Moura, L. de and Bjørner, N. Z3: An Efficient SMT Solver. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer, 2008, vol. 4963, p. 337–340. Lecture Notes in Computer Science. ISBN 978-3-540-78799-0.

[19] Nilim, A. and Ghaoui, L. Robust Control of Markov Decision Processes with Uncertain Transition Matrices. *Operations Research*, october 2005, vol. 53, p. 780–798.

[20] Puterman, M. L. *Markov decision processes: discrete stochastic dynamic programming*. Hoboken, NJ: John Wiley & Sons, 2014. ISBN 978-1-118-62587-3.

[21] Rudell, R. Dynamic variable ordering for ordered binary decision diagrams. In: *Proceedings of 1993 International Conference on Computer Aided Design (ICCAD)*. 1993, p. 42–47. ISBN 0-8186-4490-7.

[22] Salzberg, S. and Segre, A. Review of C4.5: Programs for Machine Learning by J. Ross Quinlan. *Machine Learning - ML*, september 1994, vol. 16, p. 235–240. Available at: https://doi.org/10.1007/BF00993309.

[23] Shapley, L. S. Stochastic Games. In: *Stochastic Games And Related Topics: In Honor of Professor L. S. Shapley*. Dordrecht: Springer, 1991, p. 201–206. ISBN 978-94-011-3760-7.

[24] Solar Lezama, A. Program Sketching. *International Journal on Software Tools for Technology Transfer*, 2013, vol. 15, no. 5, p. 475–495. ISSN 1433-2787. Available at: https://doi.org/10.1007/s10009-012-0249-7.

Appendix A

Contents of the External Attachment

The submitted media contains the source files of PAYNT with the extensions developed in this thesis, as well as the source files of this text. The primary components are organized as follows:

```
/
  docs/ ........................ LaTeX source files of this text
  install.sh ................... Installation script
  models/ ...................... Experiment models from section 5
  paynt/ ....................... Source files of PAYNT
    cli.py ..................... Source files for CLI
    parser/ .................... Source files for parsing input
    quotient/ .................. Source file for model representation
    synthesizer/ ............... Source files for synthesizer
  payntbind/ ................... Source files for C++ bindings
  paynt.py ..................... PAYNT Synthesizer entry point
  README.md .................... README file
```

The `docs/` directory houses the LaTeX source files for this thesis. An `install.sh` script is provided to facilitate dependency setup on Ubuntu-based systems. Benchmark models, used for the experimental evaluations (Section 5), are located in the `models/` directory, specifically within the `archive/atva24-policy-trees/` subdirectory. The core Python implementation of the synthesizer resides within the `paynt/` directory. The `payntbind/` directory contains performance-critical parts of the synthesizer together with the C++ bindings for PAYNT. Finally, `paynt.py` script serves as the entry point for synthesizer execution.

Appendix B

Installation and Running

The project is run using the source files provided in the external attachment. `install.sh` script automates dependency setup for Ubuntu-based systems. For other systems, dependencies require manual installation, adapting the instructions within the script. The synthesizer is executed from the command line via the `paynt.py` script. A typical invocation is:

```
python3 paynt.py <model-path> --export-synthesis <name> [options]
```

Key Command-Line Arguments

`<model-path>` (Required) Directory path to model files. This directory should contain *sketch.templ*, *sketch.props*, and optionally *sketch_alt.props* for dtNESt.

`--export-synthesis <name>` (Required) Base name for output files and the directory for synthesis results.

`--add-dont-care-action` Includes a “don’t care” action in the MDPs, essential for the dtNESt feature’s optimization process.

Thesis-Specific Features

To activate the novel synthesis and optimization techniques developed in this thesis, the `--ldokoupi` parameter must be used. If omitted, the synthesizer utilizes its original synthesis algorithm but still incorporates the enhanced family merging strategy (Section 4.3) developed herein. This parameter accepts one of the following arguments:

`unpruned` Constructs the Unified Decision Tree (UDT) (Section 4.5) using policies derived directly from the game-theoretic abstraction, without subsequent processing.

`prune` Activates the heuristic-based policy state pruning techniques (detailed in Section 4.2) before constructing the UDT.

`dtNESt` Employs the MDP transformation method (Section 4.4) and consequently dtNESt.

Further configuration options are available in the `README.md` file in the repository or via the help command.