BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

USING INHERITANCE DEPENDENCIES TO ACCELERATE ABSTRACTION-BASED SYNTHESIS OF FINITE-STATE CONTROLLERS FOR POMDPS.

VYLEPŠENÍ SYNTÉZY KONTROLÉRŮ PRO POMDP S VYUŽITÍM ABSTRAKCE A PODOBNOSTI.

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **ALEKSANDR SHEVCHENKO**
AUTOR PRÁCE

SUPERVISOR **doc. RNDr. MILAN ČEŠKA, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2024

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Bachelor's Thesis Assignment

157070

Institut:	Department of Intelligent Systems (DITS)
Student:	**Shevchenko Aleksandr**
Programme:	Information Technology
Title:	**Using inheritance dependencies to accelerate abstraction-based synthesis of finite-state controllers for POMDPs.**
Category:	Formal Verification
Academic year:	2023/24

Assignment:

1. Study the state-of-the-art controller synthesis methods for Partially Observable Markov Decision Processes (POMDPs) based on MDP abstraction.
2. Evaluate these methods on practically relevant case studies and identify their limitations.
3. Design possible improvements and extensions of the methods that exploit inheritance dependencies between sub-POMDPs.
4. Implement the improvements and extensions within the tool PAYNT.
5. Carry out a detailed evaluation of the implemented methods including an extension of the existing benchmarks.

Literature:

- Kochenderfer, M.J., Wheeler, T.A., and Wray K.H, Algorithms for Decision Making, MIT Press 2021.
- Milan Češka, Nils Jansen, Sebastian Junges, and Joost-Pieter Katoen. Shepherding hordes of Markov chains. In Proc. of TACAS'19. Springer, 2019.
- Andriushchenko, R., Češka, M., Junges, S., and Katoen, J.P. Inductive synthesis of finite-state controllers for POMDPs. In UAI'22. Proceedings of Machine Learning Research.
- Andriushchenko, R., Češka, M., Junges, S., Katoen, J.P. and Stupinský, Š. PAYNT: A Tool for Inductive Synthesis of Probabilistic Programs. In CAV 2021. Springer.

Requirements for the semestral defence:
Items 1, 2 and partial 3 and 4.

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:	Hanáček Petr, doc. Dr. Ing.
Beginning of work:	1.11.2023
Submission deadline:	9.5.2024
Approval date:	23.4.2024

Abstract

Partially observable Markov decision process is an important model for autonomous planning used in many areas, such as robotics and biology. This work focuses on the Abstraction-Refinement framework for the inductive synthesis of finite-state controllers (FSCs) for POMDPs. The classic version of AR requires model checking of a quotient MDP for an entire set of compatible choices of the subfamily in each iteration. We propose an algorithm that uses inheritance dependencies to reduce the size of the quotient MDP's mask and accelerate model checking for subfamilies of FSCs. We also introduce a smart version of this algorithm, which preserves all its advantages and reduces its weaknesses. During the experiments, it turned out that our approach also affects the operation of other parts of the synthesis, e.g. model building. Depending on the POMDP model, we observe both speedups and slowdowns in comparison to AR. On average, our approach speeds up the overall synthesis time by 1.2 times, and in some cases up to the factor 10.

Abstrakt

Částečně pozorovatelný Markovský rozhodovací proces (POMDP) je důležitým modelem pro autonomní plánování, který se používá v mnoha oblastech, jako je robotika a biologie. Tato práce se zaměřuje na metodu Abstraction-Refinement pro induktivní syntézu konečně stavových kontrolérů (FSC) pro POMDP. Klasická verze AR vyžaduje model checking quotient MDP pro celou množinu kompatibilních akcí podrodiny v každé iteraci. My navrhujeme algoritmus, který využívá dědičné závislosti ke snížení velikosti masky pro quotient MDP a ke zrychlení model checkingu pro podrodiny FSC. Také představujeme chytrou verzi tohoto algoritmu, která zachovává všechny jeho výhody a snižuje jeho slabiny. Během experimentů se ukázalo, že náš přístup také ovlivňuje činnost jiných částí syntézy, jako je např. model building. V závislosti na modelu POMDP, pozorujeme jak zrychlení, tak zpomalení ve srovnání s AR. V průměru naše metoda zrychluje celkovou dobu syntézy 1.2 krát a v některých případech až desetkrát.

Keywords

Partially observable Markov decision process, finite-state controller synthesis, model checking, inheritance dependencies for Markov models.

Klíčová slova

Částečně pozorovatelný Markovský rozhodovací proces, syntéza konečně stavových kontrolérů, model checking, dědičné závislosti pro Markovské modely.

Reference

SHEVCHENKO, Aleksandr. *Using inheritance dependencies to accelerate abstraction-based synthesis of finite-state controllers for POMDPs.*. Brno, 2024. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor doc. RNDr. Milan Češka, Ph.D.

Rozšířený abstrakt

Pravděpodobnost již dlouho slouží jako základní nástroj pro modelování nejistot a přijímání informovaných rozhodnutí. Markovský řetězec (MC, nebo diskrétní MC) je nejpopulárnějším modelem pro integraci pravděpodobnosti do přechodových systémů. Je zvláště vhodný pro modelování náhodných jevů, poskytuje matematický základ pro pochopení evoluce systémů v průběhu času [7]. Markovské řetězce a jejich rozšíření nabízejí mocný nástroj pro analýzu stochastických systémů. Jsou využívány v mnoha oblastech: robotice (plánování strategií pro roboty, vyhýbání se chybám [23]), biologii (vyhynutí populací, šíření epidemií [1]), financích (měnový trh, investiční strategie [25]) atd.

Některé systémy mohou zahrnovat současné procesy se střídavým chováním. Deterministická struktura MC postrádá flexibilitu, která je potřebná pro jejich adekvátní reprezentaci. Z tohoto důvodu Markovský rozhodovací proces (MDP) přichází jako náhrada za MC. MDP umožňuje soužití nedeterministických rozhodnutí a pravděpodobnostních přechodů, poskytuje podrobnější reprezentaci systémů zahrnujících jak náhodnost, tak souběžnost [7]. V MDP se předpokládá, že stav systému je plně pozorovatelný, což znamená, že o něm existuje úplná informace. Bohužel to není vždy pravda, například kvůli nedokonalosti senzorů (nebo jiného nástroje pro monitorování stavů) [20]. Částečně pozorovatelný Markovský rozhodovací proces (POMDP) je obecnější a realističtější model, který předpokládá, že existuje nejistota ohledně účinků akcí a skutečného stavu světa. POMDP jsou výpočetně náročnější než MDP kvůli přidané složitosti částečné pozorovatelnosti.

Rezoluce nedeterminismu je prováděna prostřednictvím plánovačů [7]. Existují dva problémy spojené s analýzou POMDP – jak efektivně reprezentovat plánovače a najít ten optimální. Belief MDP je jednou z možných reprezentací plánovačů. Belief je informační vektor, který reprezentuje distribuci pravděpodobnosti mezi stavy POMDP. Belief MDP je vytvořen na základě všech dosažitelných beliefs [29]. Poté se optimální plánovač hledá model checkingem belief MDP. Nicméně, pokud je belief prostor pro POMDP spojitý, model checking se stává výpočetně neřešitelným. Jinak může být belief MDP vytvořen pomocí aproximačních technik. To vede ke snížení přesnosti a možné ztrátě nejlepšího řešení [22]. Plánovače mohou být také zakódovány konečně stavovým kontrolérem (FSC) s využitím vnitřního paměťového stavu. FSC umožňuje kompaktní reprezentaci plánovačů, proto není potřeba pamatovat si celou historii akcí a pozorování [27]. Výsledkem aplikace FSC na POMDP je induced MC. MDP je vhodný model pro syntézu kontrolérů, protože umožňuje definovat kontrolované akce. Nicméně, každý FSC udržuje velikost plánovače omezenou a ve většině případů je počet všech možných FSC nekonečný. To dělá problém nalezení optimálního plánovače nerozhodnutelným [26].

Induktivní syntéza FSC [5] je moderní metoda syntézy FSC, která pracuje s rodinami plánovačů. To umožňuje prozkoumat plánovače s různými velikostmi paměti, přizpůsobuje se tak složitosti daného POMDP [3]. Hlavní omezení této metody vychází z problému FSC popsaného výše. Nekonečně rostoucí prostor plánovačů dělá problém nalezení optimálního FSC nerozhodnutelným. Abstraction-Refinement framework pro induktivní syntézu [13] pracuje s abstrakcí rodiny. Quotient MDP je společnou abstrakcí celé rodiny, která zachovává chování všech jednotlivých realizací (FSC). Umožňuje přepínat mezi realizacemi a simulovat chování induced MC, který původně nebyl v rodině přítomen. Model checking quotient MDP poskytuje spodní a horní hranice hodnoty FSC [5]. Pokud celý interval mezi těmito hranicemi splňuje specifikované podmínky, jsou přijata všechna možná řešení. Pokud se zcela nachází mimo požadovaný interval, je celá rodina odmítnuta. Jinak je rodina rozdělena na dvě poloviny (podrodiny), které jsou prozkoumávány odděleně. Pokud podrodina není přijata ani odmítnuta, je znovu rozdělena na dvě poloviny. Pro feasibil-

ity synthesis problem, tento proces končí, když je buď nalezeno vhodné řešení, nebo jsou odmítnuty všechny realizace. Rozdělení (pod)rodiny postupně snižuje počet kompatibilních akcí a provádí se omezením quotient MDP. Omezení v podstatě aplikuje masku vybraných akcí na quotient MDP.

Hlavním úkolem této práce je vytvořit metodu, která zrychluje model checking rodin FSC za použití dědičné závislosti pro AR (IDAR). Výsledky model checkingu rodiny (rodiče) mohou být užitečné i pro analýzu jejích přímých podrodin (dětí). Nechť je stav quotient MDP vágní, pokud existuje nenulová pravděpodobnost, že jeho optimální akce z plánovače rodiče nezůstane optimální pro dítě. Stavy, které ztratily svou optimální akci, se stávají vágními. Předchůdci vágních stavů jsou také označeni jako vágní. Poté jsou tyto informace použity k vytvoření masky. Pokud je stav vágní, jsou zachovány všechny jeho dostupné akce. Jinak je zachována pouze optimální akce. IDAR snižuje velikost masky pro každou podrodinu. Pro model checking MDP s PCTL syntaxí je algoritmická složitost polynomiální ve velikosti MDP [28]. Každý nevágní stav quotient MDP významně urychluje jeho model checking eliminací nedeterminismu.

Rozšířená verze IDAR (EIDAR) bere v úvahu umístění optimálních akcí nejen pro stavy, které ztratily nějaké své akce, ale také pro jejich předchůdce. EIDAR vytváří množinu affected stavů tím, že přímo prochází optimálními akcemi k jejich původním stavům. Tím výrazně snižuje počet affected stavů ve srovnání s vágními. Výsledná maska je tedy menší, což přispívá k urychlení model checkingu. Nicméně, popsaná vylepšení nezrychlují syntézu pro každý model POMDP, náš program se proto může rozhodnout, zda použít dědičné závislosti, či nikoliv (Smart EIDAR). SEIDAR, hlavní přínos této práce, je navržen tak, aby uživatele ušetřil manuální volby mezi AR, IDAR a EIDAR. SEIDAR zahajuje svou činnost v EIDAR, sbírá určité statistiky během prvních několika iterací a rozhoduje, zda přejít na AR, nebo zůstat v EIDAR.

Všechny nové přístupy se ukázaly jako konzistentní s klasickým AR. Nezávisle na velikosti paměti FSC se optimální výsledek nezávisí na zvolené metodě. Během experimentů se ukázalo, že velikost masky také ovlivňuje činnost jiných částí syntézy (např. model buildingu). V průměru SEIDAR zrychluje celkovou dobu syntézy 1.2 krát, model building 1.54 krát a model checking 1.61 krát. V některých případech pro feasibility synthesis problem zrychlení přesahuje desetkrát. Na základě experimentů je pro uživatele rozumné vždy volit SEIDAR před AR.

Using inheritance dependencies to accelerate abstraction-based synthesis of finite-state controllers for POMDPs.

Declaration

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

.....
Aleksandr Shevchenko
May 7, 2024

Acknowledgements

I would like to thank my supervisor, doc. RNDr. Milan Češka, Ph.D., and my consultant, Ing. Roman Andriushchenko, for their help and ideas that significantly improved the quality of this thesis. I thank my family, who supported me even from far away throughout my years of study. I would also like to thank my friends and colleagues for their genuine interest in the subject of my thesis.

Contents

List of Figures

Chapter 1

Introduction

Probability has long served as a fundamental tool for modelling uncertainties and making informed decisions. Markov chain (MC, or discrete-time MC) is the most popular model for the integration of probabilities into transition systems. It is particularly suited for modelling random phenomena, providing a mathematical foundation to understand the evolution of systems over time [7]. Markov chains and their extensions offer a powerful tool for the analysis of stochastic systems. They are used in many areas: robotics (planning strategies for robots, error avoidance [23]), biology (population extinction, spread of epidemics [1]), finance (currency market, investment strategies [25]), etc.

However, some systems may involve concurrent processes with interleaving behaviour. MC's deterministic structure lacks the flexibility needed to represent them adequately. For this reason, the Markov decision process (MDP) comes to replace MC. MDP allows for the coexistence of nondeterministic decisions and probabilistic transitions, providing a more detailed representation of systems involving both randomness and concurrency [7]. In MDPs, it is assumed that the system's state is fully observable, meaning there is complete information about it. Unfortunately, this is not always true, for example, due to the imperfection of sensors (or any other tool for monitoring states) [20]. The partially observable Markov decision process (POMDP) is a more general and realistic model, assuming that there is uncertainty about both the effects of actions and the true state of the world. POMDPs are computationally more challenging than MDPs due to the added complexity of partial observability.

The resolution of nondeterminism is performed by policies (schedulers) [7]. There are two problems related to POMDP analysis – how to represent policies and find the optimal one efficiently. Belief MDP is one of the possible representations of policies. Belief is an information vector that represents probability distributions over POMDP's states. Belief MDP is built based on all reachable beliefs [29]. Then, the optimal policy is found by the model checking of belief MDP. However, if the belief state space of POMDP is continuous, model checking becomes computationally unsolvable. Otherwise, the belief MDP can be constructed using approximation techniques. It leads to decreased accuracy and a possible loss of the best solution [22]. Policies can also be encoded by a finite-state controller (FSC) using an internal memory state. FSCs allow policies to be represented compactly, therefore there is no need to remember the entire history of actions and observations [27]. The result of applying FSC over POMDP is induced MC. MDP is a suitable model for controller synthesis, as it allows to define controllable actions. However, each FSC keeps the policy size bounded and in most cases, the number of all possible FSCs is infinite. It makes the problem of finding the optimal policy undecidable [26].

This paper describes two state-of-the-art methods for the synthesis of FSCs. A belief-based approach synthesises FSCs using the concept of beliefs. In this case, FSC can be derived from finite or infinite belief MDP. However, deriving FSC from large or infinite belief MDP requires a finite approximation – cut-offs or clipping [10]. Another approach, inductive synthesis of FSCs [5], works with families of policies. It allows for the exploration of policies with varying memory sizes, adapting to the complexity of the underlying POMDP [3]. The main limitation of this method stems from the FSC problem described above. Infinitely growing policy space makes the problem of finding the optimal FSC undecidable.

There are several approaches for identifying the *best* FSC within the design space. CounterExample-Guided Inductive Synthesis (CEGIS) performs an enumerative search within a family of FSCs [4]. Realisations that violate the specification provide facts (*counterexamples*) and help avoid the consideration of other certainly violating FSCs. Abstraction-Refinement framework for inductive synthesis [13] operates with an abstraction of the family. Quotient MDP is a common abstraction for the entire family preserving the behaviour of all individual realisations (FSCs). It allows to switch realisations and simulate the behaviour of an induced MC, which is not originally presented in the family. Model checking of the quotient MDP provides lower and upper bounds of the FSCs value [5]. If the entire interval between these bounds meets the specified conditions, all candidate solutions are accepted. If it fully lies outside the desired interval, the entire family is rejected. Otherwise, the family is *split* into two halves (subfamilies), which are explored separately. If the subfamily is not accepted or rejected, it is divided in half again. For the feasibility synthesis problem, the synthesis terminates when either a feasible solution is found, or all realisations are rejected. Splitting of a (sub)family gradually decreases the number of compatible actions (*choices*) and is performed by *restriction* of the quotient MDP. Restricting essentially applies a *mask* of selected choices on the quotient MDP.

Contributions

The main task of this work is to create a method that accelerates model checking of families of FSCs using Inheritance Dependencies for AR (IDAR). Model-checking results of a family (*parent*) can also be useful for the analysis of its direct subfamilies (*children*). Let a state of the quotient MDP be *vague* if there is a non-zero probability that its optimal choice from the parent's scheduler does not remain optimal for the child. States that lost their optimal choice become vague. Predecessors of vague states are marked as vague too. Then this information is used to create the mask. If a state is vague, all its available choices are kept. Otherwise, only the optimal choice is preserved. IDAR reduces the size of the mask for each subfamily. For the MDP model checking with the PCTL syntax, the algorithmic complexity is polynomial in the size of MDP [28]. Therefore, each non-vague state of the quotient MDP significantly accelerates its model checking by eliminating the nondeterminism.

The Extended version of IDAR (EIDAR) considers the location of optimal choices not only for states that lost some of their choices but also for their predecessors. EIDAR creates a set of *affected* states by going directly through optimal choices to their origin states. It significantly reduces the number of affected states compared to the vague ones. Therefore, the resulting mask is smaller, contributing to a more accelerated model checking. However, since the described improvements do not speed up the synthesis for each POMDP model, our program can decide whether to use inheritance dependencies or not (Smart EIDAR). SEIDAR, the main contribution of this thesis, is designed to save the user from manually choosing between AR, IDAR and EIDAR. SEIDAR starts its operation in EIDAR,

collects certain statistics within the first few iterations and decides whether to switch to AR or remain in EIDAR.

All new approaches proved to be consistent with classic AR. Regardless of the amount of FSC's memory, the optimal result does not depend on the chosen method. During the experiments, it turned out that the size of the mask also affects the operation of other parts of the synthesis (e.g. model building). On average, SEIDAR speeds up the overall synthesis time by 1.2 times, model building by 1.54 times and model checking by 1.61 times. In some cases for the feasibility synthesis problem, the speedups exceed 10 times. Based on the experiments, it is reasonable for the user to always choose SEIDAR over AR.

Structure of this paper

Chapter 2 introduces the fundamental theory about Markov chains and Markov decision processes. Chapter 3 compares two state-of-the-art methods for the FSC's synthesis – belief-based and inductive synthesis and describes their benefits and limitations. In Chapter 4, we introduce the novel algorithm that uses inheritance dependencies to accelerate the inductive synthesis – IDAR and its extensions. Chapter 5 provides the experimental evaluation of the proposed improvements. Chapter 6 summarises the results and describes ideas for future work. Finally, Appendices A and B describe the contents of the included storage media and the basic information to get started with PAYNT.

Chapter 2

Preliminaries

This chapter covers the fundamental concepts essential for understanding the subject of this paper. It begins with an introduction to the Markov model and Markov chains. Then, it gradually delves into the principles of partially observable Markov decision processes and the role of finite-state controllers.

2.1 Markov Model and Discrete-Time Markov Chains

A *Markov model* is a stochastic model representing dynamic systems characterized by the Markov property. This property, also referred to as a *first-order Markov assumption* or *memorylessness*, means that the probability of the next observation depends only on the current state, regardless of past observations [17].

Definition 1 (DTMC)**.** [7, 8] A *discrete-time Markov chain (DTMC, MC)* $\mathcal{D}$ is a tuple (S, s_0, P), where

- S is a finite, non-empty *set of states*,
- $s_0 \in S$ is the *initial state*,
- $P : S \times S \to [0, 1]$ is the *transition probability matrix*, where

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

and which follows the Markov property:

$$\mathbb{P}[X_{k+1} = s_{k+1} \mid X_k = s_k, \ldots, X_0 = s_0] = \mathbb{P}[X_{k+1} = s_{k+1} \mid X_k = s_k] = P(s_k, s_{k+1}),$$

where $X_k \in S$ is a random variable describing the state of $\mathcal{D}$ in time $k \geq 0$.

The transition probability matrix P is a two-dimensional array representing all possible transitions between the states of S, $n = |S|$:

$$P := \begin{bmatrix} P(s_0, s_0) & P(s_0, s_1) & \ldots & P(s_0, s_{n-1}) \\ P(s_1, s_0) & P(s_1, s_1) & \ldots & P(s_1, s_{n-1}) \\ \vdots & \vdots & \ddots & \vdots \\ P(s_{n-1}, s_0) & P(s_{n-1}, s_1) & \ldots & P(s_{n-1}, s_{n-1}) \end{bmatrix}$$

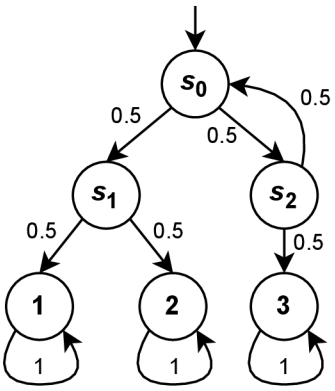


Figure 2.1: A discrete-time Markov chain (MC) for a 3-sided die.

A *transition probability graph* is a commonly used graphical representation of MC, as shown in Figure 2.1. Nodes represent states and arrows indicate non-zero transitions [8]. An arrow from the outside often depicts the initial state. States from which it is possible to transition back to itself with probability 1 are called *absorbing*. A *path* ω is a non-empty sequence of states obtained by execution of an MC [24]. Probability of a finite path ω_{fin} can be computed as follows:

$$\mathbb{P}[\omega_{fin}] := \begin{cases} 1 & \text{if } n = 0 \\ P(\omega_0, \omega_1) \cdots P(\omega_{n-1}, \omega_n) & \text{if } n > 0 \end{cases},$$

where ω_i is the ith state of ω_{fin} and $n = |\omega_{fin}|$. A *transient probability* $t_n(s) := \mathbb{P}[X_n = s \mid X_0 = s_0]$ represents the probability that the system is in state s at the time step n, assuming that the agent started its operation in the initial state s_0 [8]. Transient probability $t_n(s)$ directly depends on the transient probabilities of all states of the system at time $n-1$. Equation (2.1) is known as the *Chapman-Kolmogorov equation for the n-Step Transition Probabilities*.

$$\boldsymbol{t_n(s) = \sum_{s' \in S} t_{n-1}(s') P(s', s)}, \tag{2.1}$$

$$t_0(s) = \begin{cases} 0 & \text{if } s \neq s_0, \\ 1 & \text{if } s = s_0. \end{cases}$$

Let $t_n := [t_n(s) \mid s \in S]$ denote the row vector of transient probabilities at the time step n. It can be computed using the transition probability matrix by (2.2):

$$\boldsymbol{t_n = t_{n-1} P}. \tag{2.2}$$

To calculate a *bounded reachability* $r_{n \leq k}(s)$ of a state s, it is necessary to modify the MC's structure [7]. By making state s absorbing, its transient probability accumulates with each step, resulting in $r_{n \leq k}(s) = t_k(s)$. An essential aspect of MC's analysis is computing a probability of eventually reaching a set of states B. It is also called an *unbounded reachability probability* r_s of B from s. If B is not reachable from s, then $r_s = 0$, and $r_s = 1$ if $s \in B$. Otherwise, r_s is computed using equation (2.3).

$$\boldsymbol{r_s = \sum_{s' \in S} P(s, s') r_{s'}}. \tag{2.3}$$

Example 1. *Simulating a 3-sided die with a fair coin.*

Let us simulate the behaviour of a 3-sided die using coin tosses. The probability of tossing each side has to be equal. The corresponding MC is illustrated in Figure (2.1).

a) What is the probability of completing the simulation within a maximum of 4 coin tosses? The correct approach is to use the equation (2.2) and summarize the values of transient probabilities for states $1, 2, 3$, which are already absorbing:

$$P = \begin{bmatrix} 0 & 0.5 & 0.5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.5 & 0.5 & 0 \\ 0.5 & 0 & 0 & 0 & 0 & 0.5 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$t_0 = [1, 0, 0, 0, 0, 0]$$

$$t_4 = t_0 P^4 = [0.0625, 0, 0, 0.3125, 0.3125, 0.3125]$$

$$r_{n \leq 4}(1, 2, 3) = t_4(1) + t_4(2) + t_4(3) = \mathbf{0.9375}$$

b) Verify the correctness of the created protocol (MC), ensuring that each side of the die is tossed with the same probability of $\frac{1}{3}$. Using equation (2.3) for state 3:

$$x_{s_0} = 0.5x_{s_1} + 0.5x_{s_2}$$

$$x_{s_1} = 0.5x_1 + 0.5x_2$$

$$x_{s_2} = 0.5x_{s_0} + 0.5x_3$$

$$x_1 = 0, x_2 = 0, x_3 = 1$$

The solution to this system of equations is $x_{s_0} = \frac{1}{3}$. Using the same approach, x_{s_0} for eventually reaching states 1 and 2 are also $\frac{1}{3}$. Therefore, the protocol is correct.

□

2.2 Markov Decision Processes

This section is dedicated to the MC's extension called the Markov decision process (MDP). It is a complex model that can cope with more complicated tasks, which MC would not be able to handle. MDP introduces the concept of controlled actions and is a core model for sequential decision-making.

Definition 2 (MDP)**.** [7, 15] A *Markov decision process (MDP)* $\mathcal{M}$ is a tuple (S, s_0, Act, P), where

- S is a finite, non-empty *set of states*,
- $s_0 \in S$ is the *initial state*,
- Act is a finite, non-empty *set of actions*,
- $P : S \times Act \times S \to [0, 1]$ is the *transition probability function*, where

$$\forall s \in S, \forall \alpha \in Act : \sum_{s' \in S} P(s, \alpha, s') \in \{0, 1\}.$$

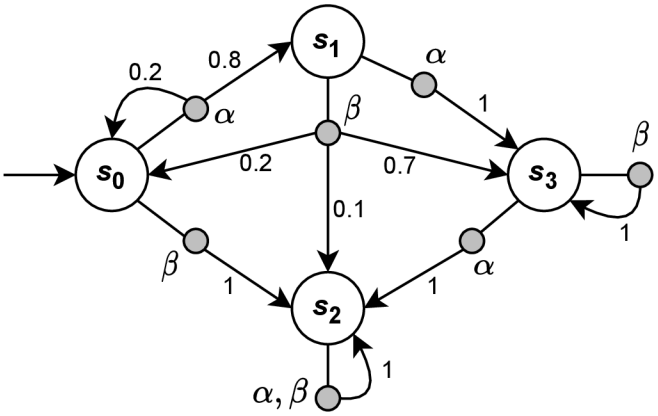


Figure 2.2: An example of Markov decision process (MDP).

If $\sum_{s' \in S} P(s, \alpha, s') = 1$, action α is called *enabled* in s [7]. $Act(s)$ denotes the set of all enabled actions for s and is required to be non-empty. [15] The behaviour of an MPD $\mathcal{M}$ can be described as follows. The agent starts the operation from the initial state s_0. If after $n \geq 0$ steps the current state is s_n, a choice between enabled actions $Act(s_n)$ needs to be done. If the agent lacks additional information about the frequency of available actions, it is selected nondeterministically. The next state s_{n+1} is selected randomly according to the distribution $P(s, \alpha, \cdot)$. An example of MDP is shown in Figure 2.2. This graphical representation is inspired by [15].

Unlike MC, a *path* $\pi = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_{n-1}} s_n$ in MDP also includes selected actions. Its probability can be computed as follows:

$$\mathbb{P}[\pi] = \prod_{k=0}^{n-1} P(s_k, \alpha_k, s_{k+1}).$$

However, transient analysis for MDP is impossible without knowing how actions are selected. Essentially, an MC is a special case of an MDP where only one action can be executed in each state. The resolution of nondeterminism is crucial for determining probability measures, and this resolution is performed by *deterministic schedulers* (*policies*).

Definition 3 (Scheduler)**.** [7]Let $\mathcal{M} = (S, s_0, Act, P)$ be an MDP. A *scheduler* for $\mathcal{M}$ is a function $\sigma : S^+ \to Act$, such that

$$\forall s_0 s_1 \ldots s_n \in S^+ : \sigma(s_0 s_1 \ldots s_n) \in Act(s_n).$$

The path

$$\pi = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} s_2 \xrightarrow{\alpha_2} \ldots$$

is called a σ-path if

$$\forall i \geq 0 : \alpha_i = \sigma(s_0 \ldots s_i).$$

An important note: actions are not preserved in the *history* $s_0 s_1 \ldots s_n$, because each action $\alpha_i, i < n$ is already chosen deterministically by σ. If at some point any path fragment is $s_i \xrightarrow{\alpha_i} s_{i+1}$, and $\sigma(s_0 \ldots s_i) \neq \alpha_i$, this path is not a σ-path.

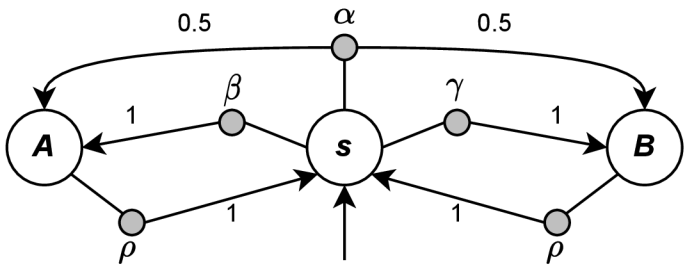


Figure 2.3: MDP for cleaning schedule.

Definition 4 (MC induced by MDP's scheduler)**.** [7] Let $\mathcal{M} = (S, s_0, Act, P)$ be an MDP and σ a scheduler on $\mathcal{M}$. A *MC induced by MDP's scheduler* is a Markov chain $\mathcal{M}_\sigma = (S^+, s_0, P_\sigma)$, where for $\theta = s_0 s_1 \dots s_n$:

$$P_\sigma(\theta, \theta s_{n+1}) = P(s_n, \sigma(\theta), s_{n+1}).$$

Actual state in $\mathcal{M}_\sigma$ depends on the history ω, which can be infinite. It means, that even if $\mathcal{M}$ is finite, $\mathcal{M}_\sigma$ is infinite. Between each σ-path of $\mathcal{M}$ and paths ω_σ in $\mathcal{M}_\sigma$ exists one-to-one correspondence. For a σ-path

$$\pi = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} s_2 \xrightarrow{\alpha_2} \dots,$$

and $\theta_n = s_0 s_1 \dots s_n$ the corresponding ω_σ path is

$$\omega_\sigma = \theta_0 \theta_1 \theta_2 \dots.$$

Example 2. *Cleaning schedule using MDP (inspired by [7]).*

Anna and Bob are planning a cleaning schedule for their apartment. They have created an MDP (Figure 2.3) $\mathcal{M}$ that allows them to either flip a coin (action α) or directly assign cleaning to someone (β and γ). Action ρ represents the cleaning itself. They have proposed two approaches: a) each time, the coin decides who will clean up; b) the coin makes the first decision, after which they alternate. Scheduler σ_a always selects α in s, while scheduler σ_b selects α only as the first action; actions β and γ are chosen depending on s_{n-1}:

$$\sigma_a(s_0 s_1 \dots s_n) = \begin{cases} \alpha & \text{if } s_n = s \\ \rho & \text{otherwise.} \end{cases} \qquad \sigma_b(s_0 s_1 \dots s_n) = \begin{cases} \alpha & \text{if } n = 0 \\ \beta & \text{if } s_{n-1} = B \\ \gamma & \text{if } s_{n-1} = A \\ \rho & \text{otherwise.} \end{cases}$$

By gradually following σ_a and σ_b, $\mathcal{M}_{\sigma_a}$ (Figure 2.4) and $\mathcal{M}_{\sigma_b}$ (Figure 2.5) are obtained.

□

Definition 5 (Finite-memory scheduler)**.** [7] Let $\mathcal{M} = (S, s_0, Act, P)$ be an MDP. A *finite-memory scheduler* for $\mathcal{M}$ is a tuple $\sigma_{fm} = (N, n_0, \eta, \delta)$, where

- N is a finite *set of nodes*,
- n_0 is the *initial node*,

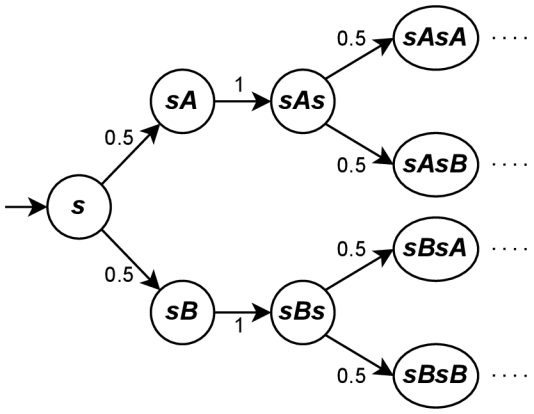


Figure 2.4: Induced MC for coin flipping.

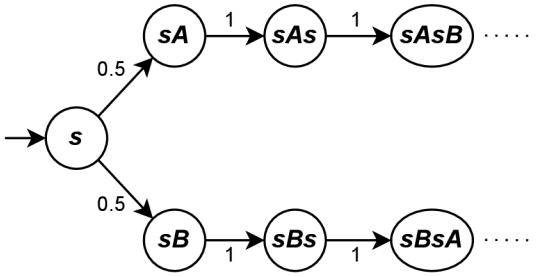


Figure 2.5: Induced MC for coin flipping and alternating.

- $\eta : N \times S \to Act$ is the action selection function, $\forall n \in N, \forall s \in S : \eta(n, s) \in Act(s)$,
- $\delta : N \times S \to N$ is the transition function.

A *memoryless scheduler* is a finite-memory scheduler with just a single node. The behaviour of a finite-memory scheduler σ_{fm} for an MDP $\mathcal{M}$ can be described as follows. The agent starts its work in the initial state s_0 and the initial node n_0. If after $k \geq 0$ steps the current state is s_k and the node is n_k, an action $\alpha_{k+1} = \eta(n_k, s_k)$ is selected and σ_{fm} evolves its node to $n_{k+1} = \delta(n_k, s_k)$. The next state s_{k+1} is selected randomly according to the distribution $P(s, \alpha, \cdot)$. This principle resembles the behaviour of a deterministic finite automaton [7]. Therefore, an MC ${\mathcal{M}'}_\sigma$ induced by σ_{fm} is finite. Its states can be represented as pairs $\langle s, q \rangle$ and the transition probabilities are

$$P'_\sigma(\langle s, q \rangle, \langle s', q' \rangle) = P(s, \eta(q, s), s'),$$

considering that $\delta(q, s) = q'$.

Model-checking algorithms for MDPs provide a systematic way to verify whether a given MDP meets certain criteria. There is a large number of MDP model-checking algorithms, involving linear or dynamic-programming approaches [29], Probabilistic Computation Tree Logic (PCTL) and Linear Temporal Logic (LTL) [15], etc. These algorithms involve analysing the probabilistic behaviour of systems, computing probabilities for specified properties, and leveraging formal logic specifications to ensure correctness.

2.3 Partially Observable Markov Decision Processes

This section introduces the concept of partially observable Markov decision processes. Compared to MDP, POMDP is a more general and realistic model, assuming that there is uncertainty about both the effects of actions and the true state of the world.

Definition 6 (POMDP)**.** [30, 3] A *partially observable Markov decision process (POMDP)* is a tuple $\mathcal{P} = (\mathcal{M}, Z, O)$, where

- $\mathcal{M} = (S, s_0, Act, P)$ is the *underlying MDP*,
- Z is a finite *set of observations*,

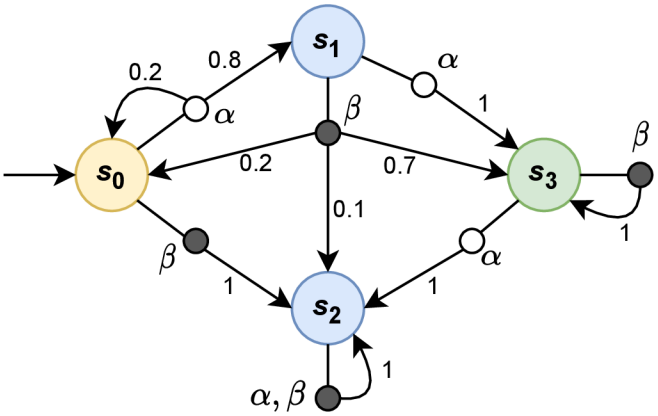


Figure 2.6: An example of partially observable Markov decision process (POMDP).

- $O : S \to Z$ is a *deterministic observation function.*

POMDP consists of an underlying MDP supplemented with observations and the observation function – the behaviour of MDP is also preserved in POMDP. Their main difference lies in the presence of observations Z, corresponding to the properties of the world, which can be detected by the agent's sensor [27]. The observation may be the same across multiple states. The agent lacks the knowledge of its current state and has information solely about the observation [30].

By integrating observations into the MDP from Figure 2.2, the POMDP from Figure 2.6 is obtained. For simplicity, observations are indicated by colours: $Z = \{$○, ○, ○$\}$. The agent receives the same observations in states marked with the same colour. This graphical representation of POMDP is inspired by [3].

Each path $\pi = s_0\alpha_0 \dots s_n$ has its *observation trace* $O(\pi) = O(S_0)\alpha_0 \dots O(s_n)$. Let $last(\pi)$ denote the last state of path π and $Paths^{\mathcal{P}}$ the set of all finite paths of $\mathcal{P}$. Scheduler σ is *observation-based*, if

$$\forall \pi, \pi' \in Paths^{\mathcal{P}} : O(\pi) = O(\pi') \implies \sigma(\pi) = \sigma(\pi').$$

Further in this paper, all POMDP schedulers are assumed to be observation-based. As the length of histories increases with each step, it becomes impractical to represent policies as mappings from histories to actions [27]. Therefore, there are two key problems in POMDP analysis: how to represent policies and find the optimal one efficiently.

2.3.1 Representation of policies by Belief MDP

A *belief* b is a probability distribution over states with the same observation [3]. Let b_0 be the initial belief state of POMDP $\mathcal{P}$ [27, 30]. According to the definition of $\mathcal{P}$ given in Section 2.3, $b_0 = \{s_0 \mapsto 1\}$ (or $b_0(s_0) = 1$). Consider a time-step t, where the agent chooses action α and belief is b_t. By slightly modifying the Kolmogorov equation (2.1) for the POMDP conditions, the probability of reaching state s' is

$$\mathbb{P}[s' \mid b_t, \alpha] = \sum_{s \in S} b_t(s) P(s, \alpha, s').$$

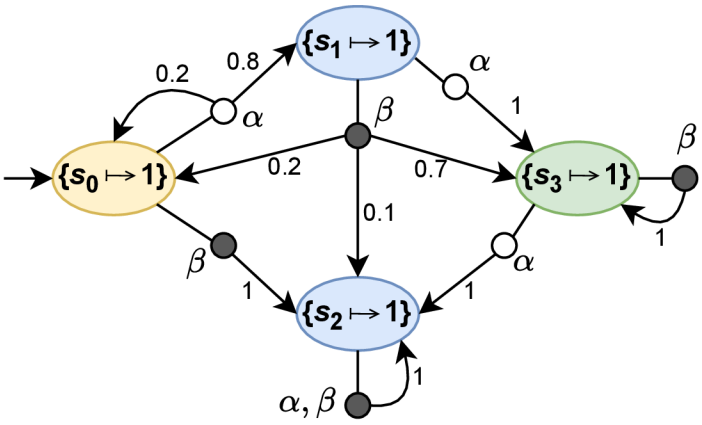


Figure 2.7: Finite belief MDP for POMDP from Figure 2.6.

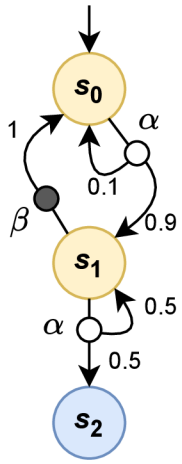


Figure 2.8: POMDP with infinite belief MDP.

Under the same conditions, the probability of reaching a state with observation z' is

$$\mathbb{P}[z' \mid b_t, \alpha] = \sum_{s' \in S, O(s')=z'} \mathbb{P}[s' \mid b_t, \alpha].$$

Updated belief b_{t+1} for z' and s' such that $O(s') = z'$ is obtained using Bayes' rule:

$$\boldsymbol{b_{t+1}(s') = \frac{\mathbb{P}[s' \mid b_t, \alpha]}{\mathbb{P}[z' \mid b_t, \alpha]} = \frac{\sum_{s \in S} b_t(s) P(s, \alpha, s')}{\sum_{s \in S} b_t(s) \sum_{s'' \in S; O(s'')=z'} P(s, \alpha, s'')}.} \tag{2.4}$$

Equation (2.4) counts the probability of reaching s' with the awareness, that observation z' is received. The updated belief b_{t+1} is a mapping of all states in S to their values obtained by equation (2.4):

$$\boldsymbol{b_{t+1} = \{s \mapsto b_{t+1}(s) \mid s \in S\}.} \tag{2.5}$$

Belief encodes necessary information about previous actions and observations. Thus, a scheduler can be represented as a mapping from belief states to actions [27]. However, there is an uncountable amount of possible beliefs. Applying such a scheduler transforms the initial discrete-time POMDP into a continuous-time MDP [30]. One possible solution is to consider only *reachable* beliefs to construct an appropriate *belief MDP*. Let $supp(b) := \{s \in S \mid b(s) > 0\}$ be the *support* of b and $O(b) = O(s)$ for any $s \in supp(b)$. Denote $b' = [b, \alpha, z']$ an updated belief for b after taking action α, $O(b') = z'$.

Definition 7 (Belief MDP)**.** [3] Let $\mathcal{P} = (S, s_0, Act, P, Z, O)$ be a POMDP. A *belief MDP* of $\mathcal{P}$ is the MDP $\mathcal{M}^{\mathcal{B}} = (\mathcal{B}_{\mathcal{M}}, b_0, Act, \mathcal{P}^{\mathcal{B}})$, where

- $\mathcal{B}_{\mathcal{M}}$ is a *set of all beliefs*,
- $b_0 = \{s_0 \mapsto 1\}$ is the *initial belief*,
- Act is the *set of actions*,
- $\mathcal{P}^{\mathcal{B}} : \mathcal{B}_{\mathcal{M}} \times Act \times \mathcal{B}_{\mathcal{M}} \to [0, 1]$ is the transition probability function such that

$$\mathcal{P}^{\mathcal{B}}(b, \alpha, b') = \begin{cases} \mathbb{P}[z' \mid b, \alpha] & \text{if } b' = [b, \alpha, z'] \\ 0 & \text{otherwise.} \end{cases}$$

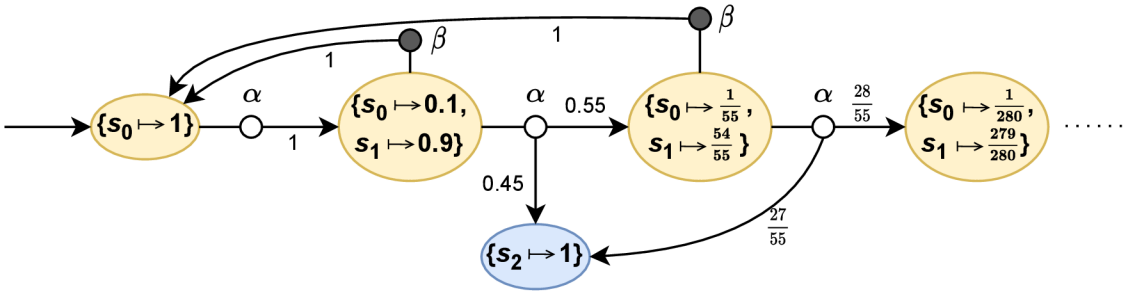


Figure 2.9: Infinite belief MDP for POMDP from Figure 2.8.

For the POMDP depicted in Figure 2.6, the corresponding belief MDP is finite (Figure 2.7). In such cases, the problem of finding the optimal policy is solvable with model checking of belief MDP. However, in some cases, belief MDP for a discrete state POMDP may have an infinite number of reachable beliefs.

Example 3. *POMDP with infinite belief MDP.*

Consider a POMDP $\mathcal{P}$, depicted in Figure 2.8, where states s_0 and s_1 have the same observation ○. Let $\sigma = \{$ ○, ○ $\mapsto \alpha\}$ be a deterministic, observation-based and memoryless scheduler of $\mathcal{P}$. Actions, which are not shown explicitly, are self-loops with probability 1. Using equations (2.4, 2.5) and Definition 7, an infinite belief MDP from Figure 2.9 is obtained.

□

Even for a simple POMDP, belief MDP can be infinite. That means that the belief MDP representation of policies does not fully solve the problem of infinitely growing histories. Nevertheless, approximation techniques can be applied to infinite belief MDPs. This concept is used in a belief-based approach for the finite-state controllers' synthesis, which will be discussed in Section 3.1.

2.3.2 Representation of policies by Finite-State Controllers

This method determines the choice of actions based on a state of internal memory [22]. The mapping from *cyclic* histories to actions can be represented by a *finite-state controller (FSC)* [27]. Each FSC encodes a finite-space policy (scheduler).

Definition 8 (FSC)**.** [5, 3] Let $\mathcal{P} = (S, s_0, Act, P, Z, O)$ be a POMDP. A *finite-state controller (FSC)* for $\mathcal{P}$ is a tuple $\mathcal{F} = (N, n_0, \eta, \delta)$, where

- N is a finite *set of nodes*,
- n_0 is the *initial node*,
- $\eta : N \times Z \to Act$ is the *action selection function*,
- $\delta : N \times Z \to N$ is the *update function*.

FSC can also be specified using Definition 5 of finite-memory scheduler for MDP. The main difference is that transitions between nodes of FSCs are based on observations, not specific states. All FSCs in this paper are considered *deterministic*. Using memory for

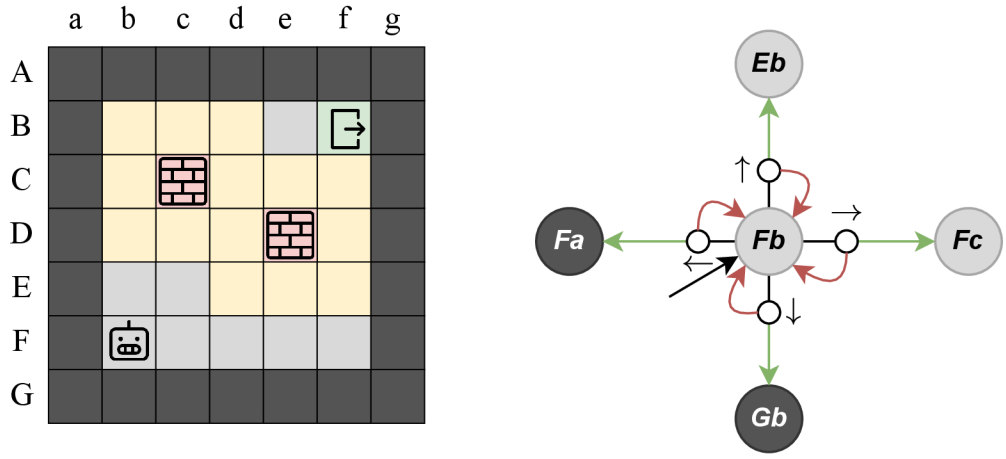


Figure 2.10: Field for the LRV from Example 4, which also represents a POMDP.

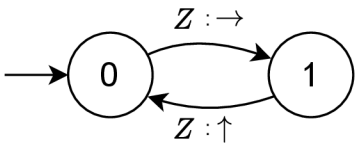


Figure 2.11: FSC for the robot, which ignores the obstacles and goes directly to the target.

POMDP's strategies is crucial. It allows the agent to apply different actions in states with the same observation.

The behaviour of an FSC $\mathcal{F}$ for a POMDP $\mathcal{P}$ can be described as follows. The agent starts its work in the initial state s_0 and the initial node n_0. If after $k \geq 0$ steps the current state is s_k and the node is n_k, an action $\alpha_{k+1} = \eta(n_k, O(s_k))$ is selected, and $\mathcal{F}$ evolves its node to $n_{k+1} = \delta(n_k, O(s_k))$. The next state s_{k+1} is selected randomly according to the distribution $P(s, \alpha, \cdot)$. An FSC is a k-FSC, if $|N| = k$. When $k = 1$, the FSC represents a memoryless policy. Denote $\mathcal{F}^{\mathcal{P}}(\mathcal{F}_k^{\mathcal{P}})$ a family of all (k-)FSCs for $\mathcal{P}$.

Definition 9 (Induced MC for FSC)**.** [3] Let $\mathcal{P} = (S, s_0, Act, P, Z, O)$ be a POMDP and $\mathcal{F} = (N, n_0, \eta, \delta)$ an FSC for $\mathcal{P}$. The *induced MC for FSC* $\mathcal{F}$ is a Markov chain $\mathcal{P}^{\mathcal{F}} = (S \times N, (s_0, n_0), P^{\mathcal{F}})$, where for all $(s, n), (s', n') \in S \times N$:

$$P^{\mathcal{F}}((s,n),(s',n')) := \begin{cases} P(s, \eta(n, O(s)), s') & \text{if } n' = \delta(n, O(s)), \\ 0 & \text{otherwise.} \end{cases}$$

Each FSC in conjunction with its corresponding POMDP generates an induced MC. The number of all possible FSCs for a particular POMDP can be infinite. Therefore, the problem of finding the optimal policy using FSC is undecidable [26]. However, the optimal solution is not always required. Thus, the problem can be simplified to the search for an FSC that satisfies given requirements.

Example 4. *Robot control using POMDP and FSC.*

Consider a Lunar Roving Vehicle (LRV, robot) located on the section (field) of the Moon, consisting of 7×7 cells (Figure 2.10). The robot is in communication with the International Space Station, so its field of view is limited. The LRV can move up ↑, down ↓, left ← or

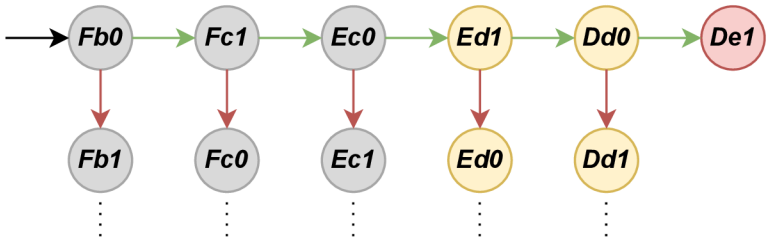


Figure 2.12: Induced MC for the robot, which goes directly to the target.

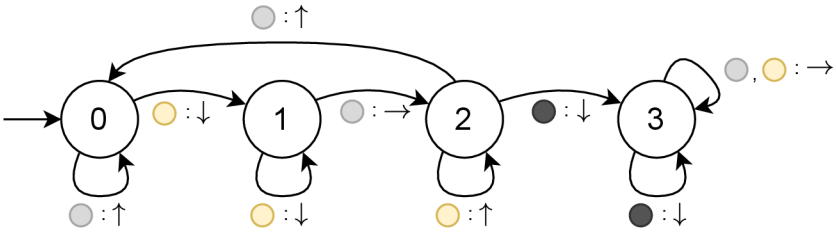


Figure 2.13: FSC for the robot, which takes into account the walls and boundaries.

right → to an adjacent cell. There is a 0.1 probability that it does not perform the specified action. Therefore, after the first step, the robot does not know its exact location. However, its scanner receives observations from the outside world, specifically:

– a normal cell.
– a **wall**. If the robot moves there, it breaks.
– a cell next to the wall. Warns of danger.
– the **final goal**.
– a cell indicating the **boundaries** of the field. If the robot goes beyond it, visibility is lost and it becomes uncontrollable.

Hence, the field can also be interpreted as POMDP. Its states are in format Xx, where X is a row of the field, x is a column. Figure 2.10 shows a part of the POMDP for a cell Fb. For clarity, the red transition means an error (probability 0.1), and the green one means success (probability 0.9). States representing walls are considered absorbing. The task is to create a policy following which the LRV reaches the final goal Bf and does not break.

a) Imagine a situation that NASA has not programmed the LRV to bypass the walls. Suppose the LRV alternates → and ↑ movements. This is realised by an FSC $\mathcal{F}_a$ with 2 nodes (Figure 2.11). In this case, Z means that the transition is made for any given observation.

Since the induced MC is too large for $\mathcal{F}_a$, Figure 2.12 illustrates only a part of $\mathcal{P}^{\mathcal{F}_a}$. For clarity, the states are indicated by the colour of the corresponding field cell. The probability of getting into the ○ state is strictly greater than the probability of reaching De without a single error:

$$\mathbb{P}[\omega] = 0.9^5 \approx \mathbf{0.59},$$

where $\omega = Fb0 \to Fc1 \to Ec0 \to Ed1 \to Dd0 \to De1$. It means that by following $\mathcal{F}_a$, the robot very likely hits the wall. Moreover, this policy does not restrict the robot from going beyond the boundaries.

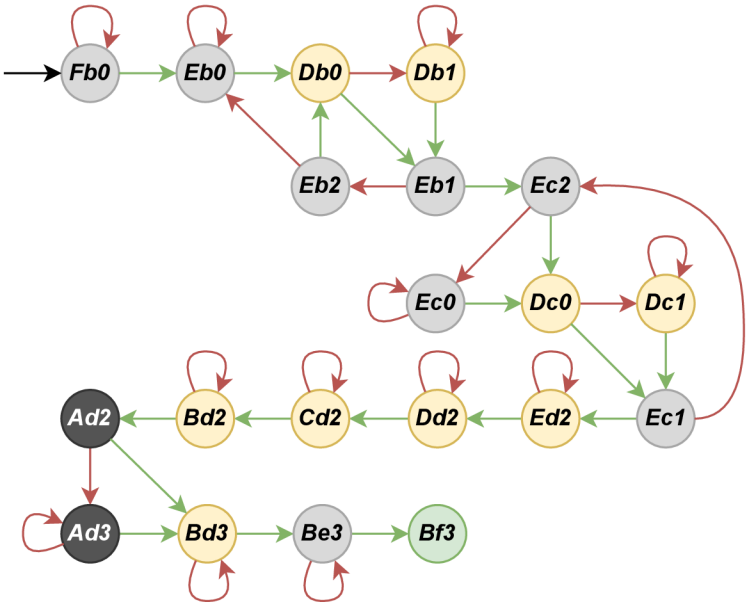


Figure 2.14: Induced MC for the robot, which takes into account the presence of walls and boundaries.

b) Now imagine that the robot takes into account the presence of walls and boundaries. Let the LRV's behaviour be described by the following algorithm, where $O(s)$ represents the current observation:

1. Move $\uparrow$ until $O(s)$ becomes ○.
2. Make a step back $\downarrow$ until success (until $O(s)$ returns to ○).
3. Make one move $\rightarrow$.
4. If $O(s)$ is ○, go to step 1. If it is ○, move $\uparrow$ until $O(s)$ becomes ●.
5. Make a step back $\downarrow$ until success (until $O(s)$ returns to ○).
6. Move $\rightarrow$ until $O(s)$ becomes ○.

Note that the action in step 3 may not always be executed, allowing the robot to move in cycles ($Eb \leftrightarrow Db$ or $Ec \leftrightarrow Dc$) for some time. The algorithm corresponds to the FSC $\mathcal{F}_b$ with 4 nodes (Figure 2.13), the induced MC $\mathcal{P}^{\mathcal{F}_b}$ is shown in Figure 2.14. Following this algorithm, the LRV avoids collisions with walls and stays within the boundaries. The probability of the LRV reaching the target without an error is

$$\mathbb{P}[\omega] = 0.9^{14} \approx \mathbf{0.23},$$

where ω is the path, obtained by following green arrows of $\mathcal{P}^{\mathcal{F}_b}$. Moreover, by following this policy, the robot reaches the goal with a probability of 1, as the task required. Note that this is not the only suitable FSC for this field.

□

Finite-state controllers provide an efficient way of decision-making in partially observable environments. Although determining the optimal policy using FSCs can be undecidable

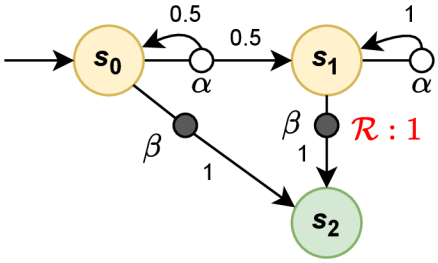


Figure 2.15: POMDP with a reward structure.

in the general case, the focus shifts to identifying FSCs that meet given requirements for particular applications. Example 4 highlights the importance of choosing the right finite-state controller to solve certain tasks.

2.4 Reward Structure and Value Function for POMDP

A *reward structure* is used for analysing the average behaviour of executions in a Markov chain [7]. Reward is a real number describing the costs or bonuses of MC's transitions. It is used to compute quantities like time, energy consumption, queue size, etc.

Definition 10 (Reward Structure)**.** [10] Let $\mathcal{M} = (S, s_0, Act, P)$ be an MDP. A *reward structure* for $\mathcal{M}$ is a function $\mathcal{R} : S \times Act \times S \to \mathbb{R}$ such that either $\forall s, s' \in S, \alpha \in Act : \mathcal{R}(s, \alpha, s') \geq 0$ ($\mathcal{R}$ is *positive*) or $\forall s, s' \in S, \alpha \in Act : \mathcal{R}(s, \alpha, s') \leq 0$ ($\mathcal{R}$ is *negative*).

Intuitively, the value $\mathcal{R}(s, \alpha, s')$ stands for the reward earned on the transition from s to s' after taking action α. Formally, the *total (cumulative) reward* for a finite path π is defined as $\mathbf{rew}_{\mathcal{M},\mathcal{R}}(\pi) := \sum_{i=0}^{|\pi|-1} \mathcal{R}(s_i, \alpha_i, s_{i+1})$. Let $\tilde{\pi}$ denote an infinite path and $\tilde{\pi}[i] = s_0\alpha_0 \ldots s_i$ a finite prefix of $\tilde{\pi}$. A *total reward until reaching a set of goal states* $G \subseteq S$ for $\tilde{\pi}$ is defined as

$$\mathbf{rew}_{\mathcal{M},\mathcal{R},G}(\tilde{\pi}) := \begin{cases} \mathbf{rew}_{\mathcal{M},\mathcal{R}}(\pi) & \text{if } \exists i \in \mathbb{N} : \pi = \tilde{\pi}[i] \wedge last(\pi) \in G \wedge \\ & \quad \forall j < i : last(\tilde{\pi}[j]) \notin G, \\ \mathbf{rew}_{\mathcal{M},\mathcal{R}}(\tilde{\pi}) & \text{otherwise.} \end{cases}$$

Thus, $\mathbf{rew}_{\mathcal{M},\mathcal{R},G}(\tilde{\pi})$ represents the cumulative reward obtained along $\tilde{\pi}$ until the first visit of a goal state $s \in G$. Let μ denote the probability distribution, $\mu_{\mathcal{M}}^{\sigma,s_0}$ a *probability measure* of the MC induced by $\mathcal{M}$, policy σ and initial state s_0 for paths in $\mathcal{M}$. An *expected total reward until reaching* G from s_0 for policy σ is

$$\mathbf{EXP}^{\sigma}_{\mathcal{M},\mathcal{R}}(s \models \Diamond G) := \int_{\tilde{\pi} \in Paths^{\mathcal{M}}_{\text{inf}}} \mathbf{rew}_{\mathcal{M},\mathcal{R},G}(\tilde{\pi}) \cdot \mu_{\mathcal{M}}^{\sigma,s_0}(d\tilde{\pi}), \tag{2.6}$$

where $Paths^{\mathcal{M}}_{\text{inf}}$ is a set of all infinite paths in $\mathcal{M}$. As mentioned earlier, this paper assumes observation-based policies for POMDP. Therefore, the expected total reward calculation algorithm for MDP also works for POMDP.

Definition 11 (Maximal Expected Total Reward)**.** Let $\mathcal{P} = (S, s_0, Act, P, Z, O)$ be a POMDP. A *maximal expected total reward until reaching* $G \subseteq S$ from s in $\mathcal{P}$ is

$$\mathbf{EXP}^{\max}_{\mathcal{P},\mathcal{R}}(s \models \Diamond G) := \sup_{\sigma \in \Sigma^{\mathcal{P}}_{\text{obs}}} \mathbf{EXP}^{\sigma}_{\mathcal{P},\mathcal{R}}(s \models \Diamond G),$$

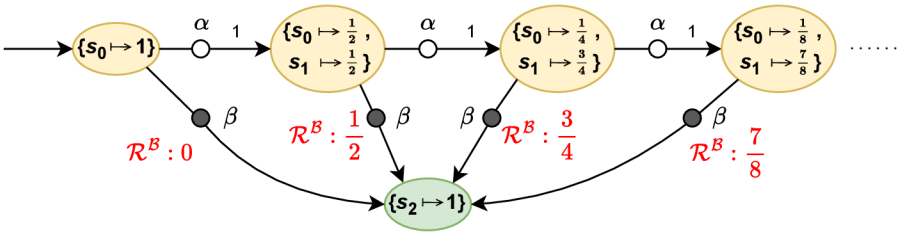


Figure 2.16: Belief MDP with belief reward structure.

where $\Sigma^{\mathcal{P}}_{\text{obs}}$ denotes a set of all observation-based policies for $\mathcal{P}$.

Computing the exact value of the maximal expected total reward for POMDP is undecidable. Consider the definitions of belief and belief MDP, described in Subsection 2.3.1. These concepts are combined with the reward structure into a *belief reward structure*.

Definition 12 (Belief Reward Structure)**.** Let $\mathcal{M}^{\mathcal{B}} = (\mathcal{B}_{\mathcal{M}}, b_0, Act, \mathcal{P}^{\mathcal{B}})$ be a belief MDP of a POMDP $\mathcal{P}$ with an associated reward structure $\mathcal{R}$. A *belief reward structure* $\mathcal{R}^{\mathcal{B}}$ based on $\mathcal{R}$ for $b, b' \in \mathcal{B}_{\mathcal{M}}$ and $\alpha \in Act$ is given by

$$\mathcal{R}^{\mathcal{B}}(b, \alpha, b') := \frac{\sum_{s \in S} b(s) \sum_{s' \in S; O(s') = O(b')} P(s, \alpha, s') \mathcal{R}(s, \alpha, s')}{\mathbb{P}[O(b') \mid b, \alpha]}.$$

Belief MDP induces a function which evaluates the expected total reward in n steps for every given belief b. It quantifies the utility of b considering its potential reward. Denote $G_{\mathcal{B}} := \{b \in \mathcal{B}_{\mathcal{M}} \mid supp(b) \subseteq G\}$ a set of *goal beliefs* for G, $\mathbb{R}^{\infty} := \mathbb{R} \cup \{\infty, -\infty\}$.

Definition 13 (POMDP Value Function)**.** Let $\mathcal{M}^{\mathcal{B}} = (\mathcal{B}_{\mathcal{M}}, b_0, Act, \mathcal{P}^{\mathcal{B}})$ be a belief MDP of a POMDP $\mathcal{P}$ with an associated reward structure $\mathcal{R}$. For $b \in \mathcal{B}_{\mathcal{M}}$, a n-step *POMDP value function* $V_n : \mathcal{B}_{\mathcal{M}} \to \mathbb{R}$ is defined recursively as $V_0(b) := 0$ and

$$V_n(b) := \begin{cases} \max\limits_{\alpha \in Act} \sum\limits_{b' \in \mathcal{B}_{\mathcal{M}}} \mathcal{P}^{\mathcal{B}}(b, \alpha, b') \cdot \big(\mathcal{R}^{\mathcal{B}}(b, \alpha, b') + V_{n-1}(b')\big) & \text{if } b \notin G_{\mathcal{B}}, \\ 0 & \text{otherwise.} \end{cases}$$

The *optimal* value function $V^* : \mathcal{B}_{\mathcal{M}} \to \mathbb{R}^{\infty}$ is defined as $V^*(b) := \lim_{n \to \infty} V_n(b)$. It yields maximal expected total reward in $\mathcal{P}$ for the initial belief $b_0 = \{s_0 \mapsto 1\}$:

$$\mathbf{EXP}^{\max}_{\mathcal{P},\mathcal{R}}(s_0 \models \Diamond G) = \mathbf{EXP}^{\max}_{\mathcal{M}^{\mathcal{B}},\mathcal{R}^{\mathcal{B}}}(b_0 \models \Diamond G_{\mathcal{B}}) = V^*(b_0).$$

Example 5. [10] *Maximal expected total reward and belief reward structure.*

Consider a POMDP $\mathcal{P}$ depicted in Figure 2.15, where $\mathcal{R}(s_1, \beta, s_2) = 1$, other rewards are 0. The policy which selects α at s_0 and β at s_1 would maximize the expected total reward in $\mathcal{P}$. However, since $O(s_0) = O(s_1) = \bigcirc$, that policy is not observation-based. Consider a policy σ which for the first $n \in \mathbb{N}$ steps in $\bigcirc$ selects α and then selects β. The probability of making a transition from s_0 to s_2 in that case is 0.5^n, so the expected total reward until reaching s_2 is computed using equation (2.6):

$$\mathbf{EXP}^{\sigma}_{\mathcal{P},\mathcal{R}}\big(s_0 \models \Diamond\{s_2\}\big) = 0 \cdot 0.5^n + 1 \cdot (1 - 0.5^n) = \mathbf{1 - 0.5^n}.$$

For $n \to \infty$, the maximal expected total reward is $\mathbf{EXP}^{\max}_{\mathcal{P},\mathcal{R},\{s_2\}} = 1$. Corresponding belief MDP with belief reward structure for $\mathcal{P}$ is depicted in Figure 2.16. Note that

$$\mathbf{EXP}^{\max}_{\mathcal{M}^{\mathcal{B}},\mathcal{R}^{\mathcal{B}}}\Big(\{s_0 \mapsto 1\} \models \Diamond\big\{\{s_2 \mapsto 1\}\big\}\Big) = V^*\big(\{s_0 \mapsto 1\}\big) = \mathbf{1}.$$

□

Chapter 3

State-of-the-art Methods for the Synthesis of FSCs

Even though the problem of finding the optimal finite-state controller is undecidable, FSC remains an effective representation of POMDP's policies. Real-world tasks can be solved using FSCs, which satisfy given requirements. State-of-the-art methods for the synthesis of FSCs provide advanced and efficient techniques for the search of such FSCs. This chapter compares two state-of-the-art methods – *belief-based* and *inductive synthesis*, and describes their benefits and limitations.

3.1 Belief-Based FSC Synthesis Method

This approach derives FSCs from finite or infinite belief MDP considering its approximations [3]. The fundamental idea is to construct a finite abstraction of the belief MDP by unfolding its parts and to approximate values of beliefs that will not be explored. Then, model checking computes the under-approximative expected total reward for the resulting finite MDP. To achieve this, finite approximation techniques – *belief cut-offs* and *belief clipping* – are applied. Belief clipping [10] provides a higher approximation quality than belief cut-offs and is not fully described in this paper. The central problem of the belief-based FSC synthesis method is answering the question of whether the maximal expected total reward exceeds a given threshold $\mathbf{EXP}^{\max}_{\mathcal{P},\mathcal{R}}(s_0 \models \Diamond G) \leq \lambda$. This method aims at under-approximating the actual value of the maximal expected total reward. If the lower bound exceeds λ, then $\mathbf{EXP}^{\max}_{\mathcal{P},\mathcal{R}}(s_0 \models \Diamond G) > \lambda$.

The main idea of belief cut-offs is to suspend the exploration of the belief MDP at certain beliefs, called *cut-off beliefs*. Then, it is assumed that the goal state is reached and a sub-optimal reward is collected.

An *under-approximative value function* is $V_{\downarrow} : \mathcal{B}_{\mathcal{M}} \to \mathbb{R}^{\infty}$ such that $V_{\downarrow}(b) \leq V^*(b)$ for all $b \in \mathcal{B}_{\mathcal{M}}$, where $V_{\downarrow}(b)$ is a *cut-off value* of b. In each cut-off belief, only one transition remains, leading to a dedicated goal state b_{cut}. This transition is assigned a reward of $V_{\downarrow}(b)$, which leads to an under-approximation of the exact value of all beliefs. Figure 3.1 shows the updated belief MDP with a modified reward structure $\mathcal{R}'$ for the belief MDP from Figure 2.16 with a single cut-off belief $b = \{s_0 \mapsto \frac{1}{4}, s_1 \mapsto \frac{3}{4}\}$. The key problem is to determine an appropriate under-approximative value function. This function should be computationally efficient yet offer cut-off values close to the optimum. For a positive reward structure, the constant value of 0 is always a valid under-approximation. A more precise

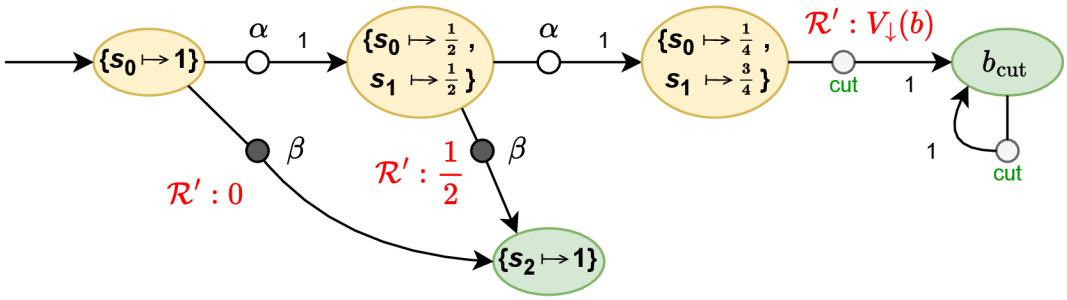


Figure 3.1: Belief MDP with cut-off approximation.

way is to compute suboptimal expected reward values for the states of POMDP using some arbitrary observation-based policy.

In the next step, this method derives an FSC from the obtained finite belief MDP. Consider a belief MDP $\mathcal{M}^{\mathcal{B}}$, which is already finite without applying any approximation techniques. Standard model-checking techniques can be used for $\mathcal{M}^{\mathcal{B}}$ to compute a memoryless policy $\sigma_{ml} : \mathcal{B}_{\mathcal{M}} \to Act$. In each $b \in \mathcal{B}_{\mathcal{M}}$, σ_{ml} selects an action that satisfies

$$\mathbf{EXP}^{\sigma_{ml}}_{\mathcal{M}^{\mathcal{B}},\mathcal{R}^{\mathcal{B}}}(b \models \Diamond G_{\mathcal{B}}) = \mathbf{EXP}^{\max}_{\mathcal{M}^{\mathcal{B}},\mathcal{R}^{\mathcal{B}}}(b \models \Diamond G_{\mathcal{B}}).$$

Then, σ_{ml} can be translated into the corresponding FSC $\mathcal{F}_{\mathcal{B}} = (\mathcal{B}_{\mathcal{M}}, b_0, \eta, \delta)$, where $\delta : \mathcal{B}_{\mathcal{M}} \times Z \times Z \to \mathcal{B}_{\mathcal{M}}$ takes into account the current and the following observations, in contrast to the Definition 8. The action selection function is $\eta(b, O(b)) = \sigma_{ml}(b)$ and the update function is $\delta(b, O(b), z') = b'$, where $b' = [b, \sigma_{ml}(b), z']$, for all $z' \in Z$. For a finite belief MDP $\mathcal{M}^{\mathcal{B}}_{\text{cut}}$ with cut-offs, a detailed explanation is provided in [3].

Belief-based FSC synthesis method integrates the concepts of belief, reward structure and finite approximation techniques. Under-approximation of the maximal expected total reward value brings flexibility in handling large or infinite belief MDPs. But at the same time, determining the exact value in POMDP is undecidable. The choice of under-approximative value function introduces a trade-off between computational efficiency and precision.

3.2 Inductive Synthesis of FSCs

This approach is based on a policy iteration algorithm introduced in [19]. Hansen's algorithm solves infinite-horizon POMDPs by exploring a space of policies, which are encoded as FSCs [27]. The inductive synthesis framework analyses finite families of FSCs by gradually increasing their memory size.

Definition 14 (Family of full k-FSCs)**.** [5, 21] Let $\mathcal{P} = (S, s_0, Act, P, Z, O)$ be a POMDP. A *family of full* k-FSCs $\mathcal{F}^{\mathcal{P}}_k$ is a tuple (N, n_0, K), where

- N is a set consisting of k nodes,
- $n_0 \in N$ is the *initial node*,
- $K = N \times Z$ is a finite *set of parameters* such that the domain of each parameter $k \in K$ is $V_{(n,z)} \subseteq Act \times N$.

Alternatively, $\mathcal{F}_k^{\mathcal{P}}$ often denotes the set of all possible k-FSCs for $\mathcal{P}$ [21]. Each member of the family can be derived by selecting parameters for functions $\eta(n, z)$, $\delta(n, z)$ for all $n \in N$, $z \in Z$. Therefore, the number of different k-FSCs in $\mathcal{F}_k^{\mathcal{P}}$ is $|\mathcal{F}_k^{\mathcal{P}}| = (|Act||N|)^{|N||Z|}$. The main problem of the family's analysis is finding the best controller with a fixed number of nodes [18]. In many experiments, certain observations may occur only in a limited number of states, or in some cases, may be unique (for example, and from Figure 2.10). There is no need to use a large number of memory nodes for such observations. Instead, a *memory restriction* $\mu : Z \to \mathbb{N}$ is introduced, where $\mu(z)$ denotes the number of memory nodes used for the observation z.

Definition 15 (Reduced family of FSCs)**.** [5] Let $\mathcal{F}_k^{\mathcal{P}} = (N, n_0, K)$ be a family of full k-FSCs and μ be a memory restriction model. A *reduced family* $\mathcal{F}_\mu^{\mathcal{P}}$ for μ is a subfamily of $\mathcal{F}_k^{\mathcal{P}}$, where $k = \max_{z \in Z}\{\mu(z)\}$, each $(n, z) \in K$ implies $n \leq \mu(z)$, and the domains $V_{n,z}$ are as in $\mathcal{F}_k^{\mathcal{P}}$. If $\delta(n, z) = n'$ and $n' > \mu(z')$, the memory update is considered invalid for the resulting observation z' and is modified to $\delta(n, z) = n_0$.

The reduced family for $\mathcal{F}_k^{\mathcal{P}}$ decreases the number of parameter domains: $\sum_{z \in Z}\{\mu(z)\} \leq k \cdot |Z|$. Such a family provides a smaller design space of FSCs and may require fewer memory nodes.

Inductive synthesis consists of two stages. The outer stage is also called a *memory injection strategy* [5]. The learner passes a selected subset of FSCs to the teacher and receives the *best* FSC with additional information from the inner synthesis stage. Finally, the learner either accepts the provided FSC or derives a new design space based on the provided information. This process involves the execution of three foundational steps:

1. *Adding memory*: Allows FSCs to store more information by making the growth of the memory size manageable. Generally, FSCs with a larger number of nodes can represent more flexible strategies and yield better results.

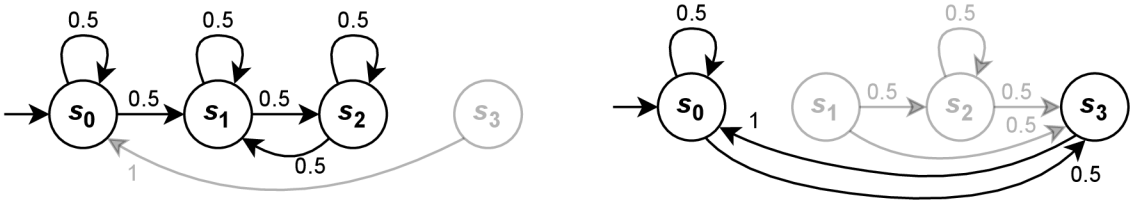
2. *Removing symmetries*: Given the topology properties of FSCs, certain controllers may be *equivalent* – encode the same policy [18]. The elimination of such symmetries reduces the size of the family.

3. *Analysing abstractions*: Guiding the search based on the results obtained from the inner stage, provided by the teacher.

The inner stage, also called the *inductive synthesis loop*, describes the internal processes of the teacher responsible for identifying the best FSC within the design space. In the following subsections, various realisations of the teacher will be presented. However, before delving into these approaches, it is essential to look into their common groundwork. Denote $Distr(X)$ a set of all probability distributions on a finite set X.

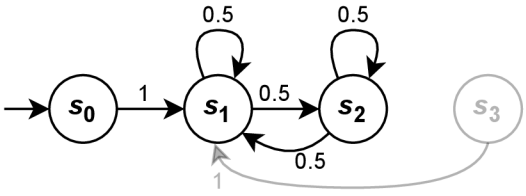
Definition 16 (Family of MCs)**.** [4, 12, 13] A *family of MCs* is a tuple $\mathfrak{D} = (S, s_0, K, \mathfrak{P})$, where

- S is a finite *set of states*,
- $s_0 \in S$ is an *initial state*,
- K is a finite *set of discrete parameters* such that the domain of each parameter $k \in K$ is $V_k \subseteq S$,

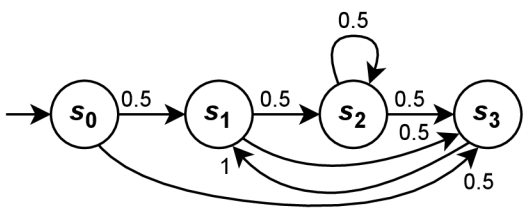


(a) $\mathcal{D}_{r_0}$ with $r_0(k_0) = s_0$, $r_0(k_2) = s_1$.

(b) $\mathcal{D}_{r_1}$ with $r_1(k_0) = s_0$, $r_1(k_2) = s_3$.



(c) $\mathcal{D}_{r_2}$ with $r_2(k_0) = s_1$, $r_2(k_2) = s_1$.



(d) $\mathcal{D}_{r_3}$ with $r_3(k_0) = s_1$, $r_3(k_2) = s_3$.

Figure 3.2: An example of all possible realisations of a family of MCs with different reachable states.

- $\mathfrak{P} : S \to Distr(K \cup S)$ is a *family of transition probability matrices.*

For a single MC, the transition probability matrix maps states to distributions over successor states. For the family of MCs, $\mathfrak{P}$ maps states to distributions over parameters. A concrete MC is obtained by instantiating each parameter with a value from its domain.

Definition 17 (Realisation)**.** [4, 13] Let $\mathfrak{D} = (S, s_0, K, \mathfrak{P})$ be a family of MCs. A *realisation* of $\mathfrak{D}$ is a function $r : K \to S$, such that $r(k) \in V_k$ for all $k \in K$. A realisation r yields an MC $\mathcal{D}_r = (S, s_0, \mathfrak{P}(r))$, where $\mathfrak{P}(r)$ is the transition probability matrix in which each $k \in K$ in $\mathfrak{P}$ is replaced by $r(k)$.

The set of all realisations of $\mathfrak{D}$ is denoted as $\mathcal{R}^{\mathfrak{D}}$. The number of all possible realisations from $\mathcal{R}^{\mathfrak{D}}$ is $|\mathcal{R}^{\mathfrak{D}}| = \prod_{k \in K} |V_k|$, which means that it is exponential in the number of parameters.

There are two fundamental synthesis problems related to families of MCs. *Threshold synthesis problem* is to identify sets of MCs satisfying and violating a given specification, respectively. *Max/min synthesis problem* is to find an MC that maximises/minimises a given objective. *Feasibility synthesis problem* is a special case of the threshold synthesis problem, which aims to find just one realisation that would meet the specification.

Example 6. *Family of MCs with different reachable states.*

Consider a family of MCs $\mathfrak{D} = (S, s_0, K, \mathfrak{P})$, where $S = \{s_0, s_1, s_2, s_3\}$, $K = \{k_0, k_1, k_2\}$ with domains $V_{k_0} = \{s_0, s_1\}$, $V_{k_1} = \{s_2\}$, $V_{k_2} = \{s_1, s_3\}$ and the family of transition probability matrices $\mathfrak{P}$ is defined as follows:

$$\mathfrak{P}(s_0) = 0.5 : k_0 + 0.5 : k_2 \qquad \mathfrak{P}(s_1) = 0.5 : k_1 + 0.5 : k_2$$

$$\mathfrak{P}(s_2) = 0.5 : k_1 + 0.5 : k_2 \qquad \mathfrak{P}(s_3) = 1 : k_0$$

All possible realisations of $\mathfrak{D}$ are shown in Figure 3.2. States, unreachable from s_0, are grayed out. This example demonstrates that MCs, yielded by distinct realisations from the same family, may have different reachable states. Let $\varphi = \mathbb{P}_{\geq 0.8}(\Diamond\{s_3\})$ be a specification. Unbounded reachability probability of s_3 for r_1 and r_3 is 1, s_3 is not reachable for r_0 and r_2. Then, the solution for the threshold synthesis problem is a set of realisations $T = \{r_1, r_3\}$. For $\phi = \Diamond\{s_3\}$, the solution to the max synthesis problem is either r_1 or r_3.

□

3.2.1 One-By-One Synthesis Approach

Within this approach, each member of the family is analysed separately [14]. The teacher receives a family of FSCs $\mathcal{F}_k^{\mathcal{P}}$ ($\mathcal{F}_\mu^{\mathcal{P}}$) and a set of constraints from the outer stage. Then, it solves the threshold or max/min synthesis problem by enumerating through all realisations $r \in \mathcal{R}^{\mathfrak{D}}$ [13]. For each yielded MC $\mathcal{D}_r$, model checking is performed based on the specified constraints. Obtained results are provided to the learner for the next outer stage loop.

However, as mentioned earlier, the number of all possible realisations from $\mathcal{R}^{\mathfrak{D}}$ is exponential in the number of parameters. The total number of states and parameters consequently explodes, making this approach unusable for large problems [4]. This leads to the necessity of applying more advanced techniques that exploit the family structure.

3.2.2 CounterExample-Guided Inductive Synthesis

Similar to the one-by-one approach, this method performs an enumerative search within a family of FSCs (or a family of realisations induced by them). The key difference lies in handling FSCs, which violate the specification. Such FSC provides facts, called *counterexamples*, and helps avoid the consideration of other certainly violating FSCs [5].

The CEGIS approach is illustrated in Figure 3.3 [4, 12]. The learner (*synthesiser*) takes a set of realisations $\mathcal{R}^{\mathfrak{D}}$ and aims to find a realisation satisfying the specification Φ. Let $Q \subseteq \mathcal{R}^{\mathfrak{D}}$ be a set of realisations that need to be checked. The learner selects a realisation r and asks the teacher (*oracle*, or *verifier*) whether it is a solution. If the teacher accepts r, it reports success. Otherwise, it returns a set $\mathcal{V}$ of realisations all violating Φ including r. Then, the learner prunes $\mathcal{V}$ from Q. In terms of parameters K of the family $\mathfrak{D}$, the oracle returns a set K' of parameters such that all realisations obtained by changing only the values assigned to K' violate Φ.

An intuitive visualisation of CEGIS for a family of 16 realisations is presented in Figure 3.4. The currently considered (violated) realisation r is marked in yellow, red indicates the set of other pruned realisations $\mathcal{V} \setminus r$. Once the teacher accepts the given realisation (marked in green), the algorithm returns the result.

The key problem of this algorithm is to compute a set $\mathcal{V}$ of realisations that are all violating Φ. Consider the threshold synthesis problem for a single specification φ. If an MC $\mathcal{D} \not\models \varphi$, a *counterexample* derived from a critical subsystem can provide diagnostic information about the source of the failure.

Definition 18 (Counterexample)**.** Let $\mathcal{D} = (S, s_0, P)$ be an MC and $s_\perp \notin S$. A *sub-MC* of $\mathcal{D}$ induced by $C \subseteq S$ is the MC $\mathcal{D}{\downarrow}C = (S \cup \{s_\perp\}, s_0, P')$, where the transition probability matrix P' is defined as

$$P'(s) = \begin{cases} P(s) & \text{if } s \in C, \\ [s_\perp \mapsto 1] & \text{otherwise.} \end{cases}$$

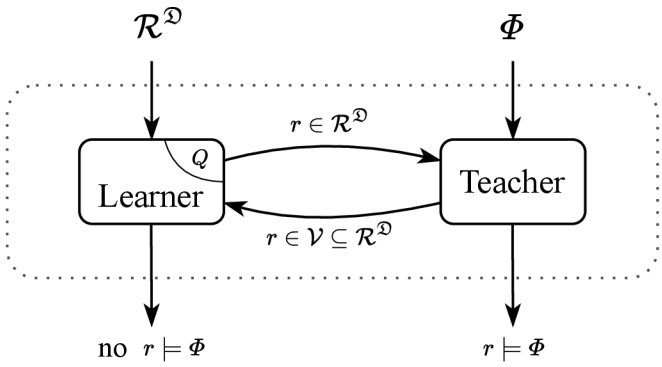


Figure 3.3: Schematic view on CounterExample-Guided Inductive Synthesis approach.

Figure 3.4: An example run of CEGIS.

The set C and the sub-MC $\mathcal{D}{\downarrow}C$ are called a *counterexample (CE)* for the property $\mathbb{P}_{\leq\lambda}[\Diamond T]$ on MC $\mathcal{D}$, if $\mathcal{D}{\downarrow}C \not\models \mathbb{P}_{\leq\lambda}\big[\Diamond(T \cap (C \cup \{s_0\}))\big]$.

Let $\mathcal{D}_r$ be an MC that violates the specification φ. To compute the rest of $\mathcal{V}$, the teacher computes a critical subsystem $\mathcal{D}{\downarrow}C$ that is then used to derive a *conflict*. Then, the set of violating realisations is computed directly from the conflict.

Definition 19 (Conflict)**.** Let $\mathfrak{D} = (S, s_0, K, \mathfrak{P})$ be a family of MCs and $C \subseteq S$. A *conflict* is a set K_C of relevant parameters given by $\bigcup_{s\in C} \mathit{supp}(\mathfrak{P}(s))$.

Definition 20 (Generalisation)**.** Let r be a realisation and $K_C \subseteq K$ be a conflict. A *generalisation* of r induced by K_C is the set $r{\uparrow}K_C = \{r' \in \mathcal{R}^{\mathfrak{D}} \mid \forall k \in K_C : r(k) = r'(k)\}$.

The size of a conflict $|K_C|$ directly impacts the size of a generalisation. Smaller conflicts potentially result in the generalisation of r to larger subfamilies $r{\uparrow}K_C \subseteq \mathcal{R}^{\mathfrak{D}}$. Hence, the CEs must consist of a minimal number of parameterised transitions.

Example 7. *Applying CEGIS on a family of MCs.*

Consider a family of MCs $\mathfrak{D} = (S, s_0, K, \mathfrak{P})$ from Figure 3.5, where $S = \{s_0, s_1, s_2, s_3\}$, $K = \{k_0, k_1\}$ with domains $V_{k_0} = \{s_1, s_2\}$, $V_{k_1} = \{s_0, s_2, s_3\}$ and the family of transition probability matrices $\mathfrak{P}$ is defined as follows:

$$\mathfrak{P}(s_0) = 0.5 : s_1 + 0.5 : k_0 \qquad \mathfrak{P}(s_1) = 1 : k_1$$
$$\mathfrak{P}(s_2) = 0.5 : s_3 + 0.5 : k_0 \qquad \mathfrak{P}(s_3) = 1 : s_3$$

Let $\varphi = \mathbb{P}_{\leq 0.2}(\Diamond\{s_3\})$ be a specification. Consider a realisation $r_a \not\models \varphi$ depicted in Figure 3.6a. Note that a sub-MC $\mathcal{D}_{r_a}{\downarrow}C$ of $\mathcal{D}_{r_a}$ with $C = \{s_0, s_2, s_3\}$, $K_C = \{k_0\}$ from Figure 3.6b also does not satisfy φ. Thus, $\mathcal{D}_{r_a}{\downarrow}C$ serves as a counterexample, covering only

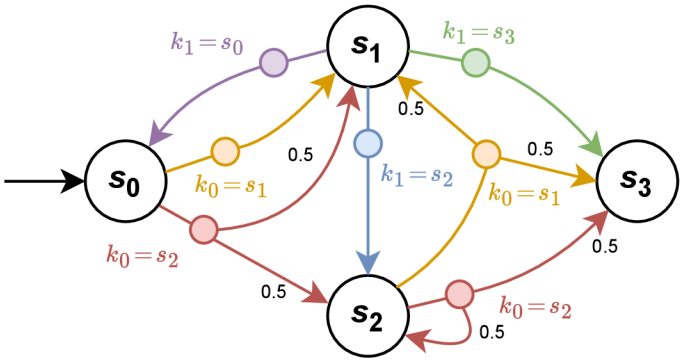
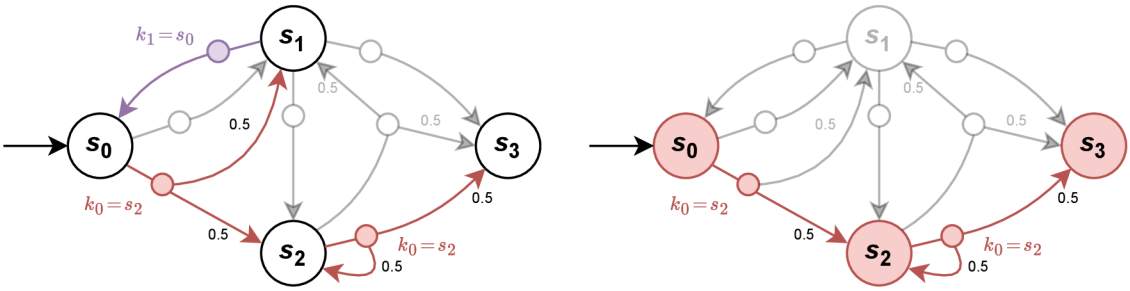


Figure 3.5: A visualisation of a family of MCs, containing all its possible realisations.



(a) $\mathcal{D}_{r_a}$ with $r_a(k_0) = s_2$, $r_a(k_1) = s_0$.

(b) $\mathcal{D}_{r_a}{\downarrow}C$ with $C = \{s_0, s_2, s_3\}$.

Figure 3.6: A violating realisation of the family and the corresponding counterexample.

parameter k_0. Even with different values of k_1, r_a would still violate φ. Consequently, a generalisation $r_a{\uparrow}K_C$ contains $|V_{k_1}| = 3$ realisations, all of which can be rejected.

However, for realisations with k_0 set to s_1, it is not possible to construct a counterexample covering only one parameter. Therefore, each potential conflict would contain both k_0 and k_1, resulting in generalisations having only one realisation each. In that case, compared to the one-by-one method, CEGIS would operate even slower because of the additional time required for searching for CEs.

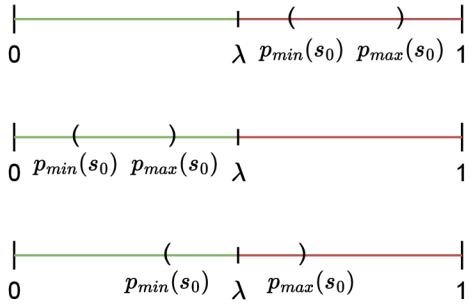
□

This framework also can be generalised to handle multiple-property specifications. It can be achieved by constructing separate conflicts for each violated property. An advanced oracle for computing the set of violating realisations $\mathcal{V}$ is presented in [4]. Its main features are taking into account the position of the parameters and using the model-checking results from an abstraction of the family.

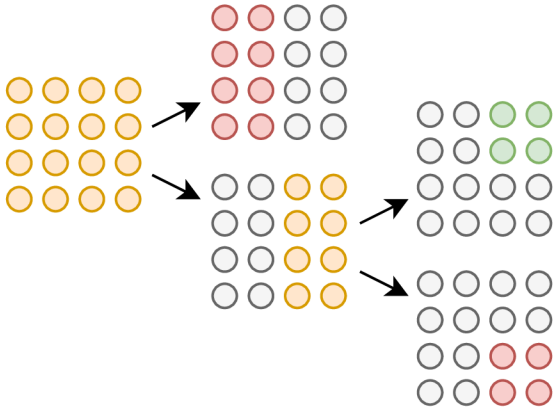
3.2.3 Abstraction-Refinement Framework for Inductive Synthesis

This framework, in comparison to the one-by-one and CEGIS methods, introduces an orthogonal *all-in-one* approach. Instead of considering members of a family of FSCs (MCs) separately, the teacher operates with its *abstraction*, represented by a single *quotient MDP* [5].

Definition 21 (Quotient MDP)**.** [2] Let $\mathfrak{D} = (S, s_0, K, \mathfrak{P})$ be a family of MCs. A *quotient MDP* of $\mathfrak{D}$ is an MDP $\mathcal{M}^{\mathfrak{D}} = (S, s_0, \mathcal{R}^{\mathfrak{D}}, P)$, where $P(\cdot, r) \equiv \mathfrak{P}(r)$.



(a) Various scenarios of the location of upper and lower bounds relative to the threshold λ.



(b) An example run of AR.

Figure 3.7: The principle of Abstraction-Refinement framework for inductive synthesis.

Considering that a POMDP and a family of its FSCs induce a family of MCs, another way to define a quotient MDP is viable [3]. For a POMDP $\mathcal{P}$ and a family of k-FSCs $\mathcal{F}_k^{\mathcal{P}} = \{\mathcal{F}_1, \ldots, \mathcal{F}_m\}$ a quotient MDP is an MDP $\mathcal{M}(\mathcal{F}_k^{\mathcal{P}}) = (S \times N, (s_0, n_0), \{1, \ldots, m\}, P^{\mathcal{F}_k^{\mathcal{P}}})$ with

$$P^{\mathcal{F}_k^{\mathcal{P}}}((s,n),i) = P^{\mathcal{F}_i},$$

where $P^{\mathcal{F}_i}$ is the transition probability matrix of the MC induced by $\mathcal{P}$ and $\mathcal{F}_i$, as introduced in Definition 9. Note that actions in quotient MDP preserve the behaviour of individual realisations (FSCs). Therefore, it allows to switch realisations and simulate the behaviour of an induced MC, which is not originally presented in the family $\mathfrak{D}$. Moreover, multiple realisations may share the same choice of action in some states. In such cases, the action is not duplicated in the quotient MDP and represents several realisations at the same time. A scheduler, which always selects the same realisation, is called *consistent*. Such a scheduler yields a valid member of the family.

Definition 22 (Consistent scheduler)**.** Let $\mathfrak{D} = (S, s_0, K, \mathfrak{P})$ be a family of MCs and $\mathcal{M}^{\mathfrak{D}} = (S, s_0, \mathcal{R}^{\mathfrak{D}}, P)$ be a quotient MDP of $\mathfrak{D}$. A (memoryless) scheduler σ_r for $r \in \mathcal{R}^{\mathfrak{D}}$ is called *r-consistent* iff $\sigma_r(s) = r$ for all $s \in S$. A scheduler is called *consistent* iff it is r-consistent for some $r \in \mathcal{R}^{\mathfrak{D}}$.

Example 8. *Quotient MDP and inconsistent schedulers.*
Consider the family of MCs $\mathfrak{D}$ from Example 7. Note that Figure 3.5 already represents a quotient MDP $\mathcal{M}^{\mathfrak{D}}$ for $\mathfrak{D}$. The number of all possible realisations is $|V_{k_0}| \cdot |V_{k_1}| = 6$, but $\mathcal{M}^{\mathfrak{D}}$ remains compact due to the reduction of duplicate actions, as e.g. in s_0 each action is shared by 3 realisations at once. An example of inconsistent scheduler may be generated by selecting $k_0 = s_2$ in s_0, $k_0 = s_1$ in s_2 and any value of k_1.

□

Consider a quotient MDP $\mathcal{M}^{\mathfrak{D}}$ for a family of MCs $\mathfrak{D}$. Although $\mathcal{M}^{\mathfrak{D}}$ overapproximates the behaviour of $\mathfrak{D}$, model checking of $\mathcal{M}^{\mathfrak{D}}$ still provides useful information for the further analysis of $\mathfrak{D}$ [2, 13]. Let $\varphi = \mathbb{P}_{\leq\lambda}(\Diamond\{G\}), G \subseteq S$ be a specification for the feasibility synthesis problem. Model checking of $\mathcal{M}^{\mathfrak{D}}$ computes maximising and minimising schedulers σ_{max} and σ_{min}, which may not necessarily be consistent. These schedulers yield vectors p_{max}

Algorithm 1 Feasibility synthesis for the Abstraction-Refinement framework

Input: A family of MCs $\mathfrak{D} = (S, s_0, K, \mathfrak{P})$ with the set of realisations $\mathcal{R}^{\mathfrak{D}}$, a specification $\varphi = \mathbb{P}_{\leq\lambda}(\Diamond\{G\}), G \subseteq S$

Output: Realisation $r \in \mathcal{R}^{\mathfrak{D}} : \mathcal{D}_r \models \varphi$, or UNSAT if no such realisation exists

$\mathfrak{X} \leftarrow \{\mathcal{R}^{\mathfrak{D}}\}$ ▷ Set of subfamilies awaiting model checking
$\mathcal{M}^{\mathfrak{D}} \leftarrow$ buildQuotientMDP($\mathfrak{D}$) ▷ Applying Def. 21
while $\mathfrak{X} \neq \emptyset$ **do**
  $\mathcal{X} \leftarrow$ any($\mathfrak{X}$)
  $\mathfrak{X} \leftarrow \mathfrak{X} \setminus \{\mathcal{X}\}$
  $\mathcal{M}^{\mathfrak{D}}[\mathcal{X}] \leftarrow$ restrict($\mathcal{M}^{\mathfrak{D}}, \mathcal{X}$) ▷ Applying Def. 24
  $(p_{min}, \sigma_{min}, p_{max}, \sigma_{max}) \leftarrow$ modelCheck($\mathcal{M}^{\mathfrak{D}}[\mathcal{X}], \varphi$)
  if $p_{max}(s_0) \leq \lambda$ **then**
    return any($\mathcal{X}$)
  end if
  if $p_{min}(s_0) > \lambda$ **then**
    continue
  end if
  if σ_{min} *is r-consistent for some* $r \in \mathcal{X}$ **then**
    return r
  end if
  $(\mathcal{X}_\top, \mathcal{X}_\bot) \leftarrow$ split($\mathcal{X}$) ▷ Applying Def. 23
  $\mathfrak{X} \leftarrow \mathfrak{X} \cup \{\mathcal{X}_\top, \mathcal{X}_\bot\}$
end while
return UNSAT

and p_{min}, containing upper and lower bounds of the reachability probability for all states of the quotient MDP. One of the three possible scenarios from Figure 3.7a may occur. If $p_{min}(s_0) > \lambda$, there is no solution for the feasibility problem – $\mathcal{D}_r \not\models \varphi$ for each realisation $r \in \mathcal{R}^{\mathfrak{D}}$. On the other hand, if $p_{max}(s_0) < \lambda$, all members of the family satisfy φ. If λ lies between the bounds and σ_{min} is consistent, then σ_{min} is a solution. Otherwise, if σ_{min} is not consistent, nothing can be concluded yet due to the too coarse abstraction. In that case, $\mathcal{M}^{\mathfrak{D}}$ is *refined* by *splitting* $\mathcal{R}^{\mathfrak{D}}$ into two subfamilies and each of them is analysed separately using the procedure described above. The refinement loop continues until either a feasible solution is found, or all realisations are rejected. The termination of the procedure is guaranteed due to the finite number of family members. This approach is summarised in Algorithm 1. It can also be modified to solve threshold or max/min synthesis problems.

An intuitive visualisation of AR for a family of 16 realisations is presented in Figure 3.7b. If the corresponding quotient MDP is too coarse, a set of realisations is marked in yellow. Red indicates realisations certainly violating the specification. Once the bounds obtained from model checking allow for accepting the subfamily (marked in green), AR returns the result.

Definition 23 (Splitting)**.** Let $\mathfrak{D} = (S, s_0, K, \mathfrak{P})$ be a family of MCs and $\mathcal{X} \subseteq \mathcal{R}^{\mathfrak{D}}$ a set of realisations. For $k \in K$ and predicate A_k over S, *splitting* partitions $\mathcal{X}$ into

$$\mathcal{X}_\top = \{r \in \mathcal{X} \mid A_k(r(k))\} \quad \text{and} \quad \mathcal{X}_\bot = \{r \in \mathcal{X} \mid \neg A_k(r(k))\}.$$

To avoid rebuilding the quotient MDP in each iteration, splitting is performed on the set of realisations, not on states of the quotient MDP. *Restricting* the actions of the quotient MDP to the particular subfamily is crucial for the AR performance.

Definition 24 (Restricting)**.** Let $\mathcal{M}^{\mathfrak{D}} = (S, s_0, \mathcal{R}^{\mathfrak{D}}, P)$ be a quotient MDP and $\mathcal{X} \subseteq \mathcal{R}^{\mathfrak{D}}$ a set of realisations. A *restriction* of $\mathcal{M}^{\mathfrak{D}}$ wrt. $\mathcal{X}$ is an MDP $\mathcal{M}^{\mathfrak{D}}[\mathcal{X}] = (S, s_0, \mathcal{R}^{\mathfrak{D}}[\mathcal{X}], P)$, where $\mathcal{R}^{\mathfrak{D}}[\mathcal{X}] = \{r \mid r \in \mathcal{X}\}$.

A good splitting strategy involves choosing a parameter $k \in K$ and a predicate A_k, which would reduce the number of model checkings required to classify all $r \in \mathcal{X}$. The two key aspects of a good k are *variance* and *consistency*. These characteristics show how the splitting may narrow the difference between p_{min} and p_{max} and how it may reduce the inconsistency of σ_{min} and σ_{max}. An efficient strategy, proposed in [13], selects k based on a light-weighted analysis of the model-checking results for $\mathcal{M}^{\mathfrak{D}}[\mathcal{X}]$.

Since the AR approach is diametrically opposite to CEGIS, these methods behave differently for various models and specifications. Depending on the topology of the state space, CEGIS may either manage to identify small conflicts and analyse only a few realisations, or be unable to prune the state space and analyse each realisation individually. As for AR, quotient MDP may yield tight bounds, so the synthesis takes only a couple of refinements, or on the contrary, the abstraction could be too coarse and require refining subfamilies up to the level of individual MCs.

3.3 Tools for Inductive Synthesis of Probabilistic Programs

PAYNT[1] (Probabilistic prograAm sYNThesizer [6]) is a tool for automatic synthesis of probabilistic programs, supporting the synthesis of FSCs for POMDPs. It takes a program *sketch*, describing a finite family of finite MCs, a specification, and finds a fitting realisation. A sketch is a probabilistic program with *holes* in the PRISM (or JANI) language, and a *realisation* of the sketch is a function that maps every hole to one of its options [12]. PAYNT implements an oracle-guided synthesis approach, supporting both CEGIS and AR methods and their hybrid combination [3].

The implementation of PAYNT utilises the probabilistic model checker Storm [16], which is able to analyse MDPs. Storm also provides a Python API, which PAYNT flexibly uses to construct the overall synthesis loop. When analysing discrete-time models, Storm focuses on PCTL logic. For SMT-solving, PAYNT uses Z3.

[1] Available at: https://github.com/randriu/synthesis

Chapter 4

Acceleration of Abstraction-Refinement Framework

The Abstraction-Refinement framework for POMDPs stands out among other FSC synthesis methods due to its all-in-one approach. A common abstraction for the entire family of FSCs – quotient MDP – allows preserving the behaviour of all realisations within a single MDP. Splitting of a (sub)family gradually decreases the number of compatible actions (*choices*) and is performed by restriction of the quotient MDP. Restricting essentially applies a *mask* of selected choices on the quotient MDP. At the heart of the refinement loop lies model checking, which is crucial for defining the upper and lower bounds of the abstraction.

Model checking, as a powerful tool, can offer even more useful information for the analysis of the family. Vectors p_{max} and p_{min}, derived from σ_{max} and σ_{min}, respectively, provide bounds for all states of the quotient MDP. However, only $p_{max}(s_0)$ and $p_{min}(s_0)$ impact the synthesis scenario. In a feasibility synthesis problem $\varphi = \mathbb{P}_{\leq\lambda}(\Diamond\{G\}), G \subseteq S$, σ_{min} serves as a memoryless and potentially inconsistent scheduler, providing the optimal action for each quotient MDP's state. Given that masks of families applied to the quotient MDP may share common choices, can σ_{min} be reused for the analysis of other families as well? Is it feasible to avoid recalculating model checking multiple times for individual parts of the quotient MDP? In this chapter, we propose improvements to the AR method that accelerate the synthesis of FSCs using *inheritance dependencies (IDAR)*. We also introduce an *extended* version of this algorithm (*EIDAR*) and a final product of this thesis, *smart* version of EIDAR – *SEIDAR*. Figures 4.1 and 4.5 provide a schematic overview of these algorithms, which are described in more detail in the following sections.

4.1 Inheritance Dependencies within Families of FSCs

The main objective of this approach is to reduce the size of the mask for each family so that the optimum obtained from model checking does not change in comparison with AR. Model-checking results of a family (*parent*) can be also useful for the analysis of its direct subfamilies (*children*). Each child is obtained by replacing a parent's hole with one of its options or their interval. Therefore, the number of children in a family may be 2 or greater. In this section, we consider the first scenario.

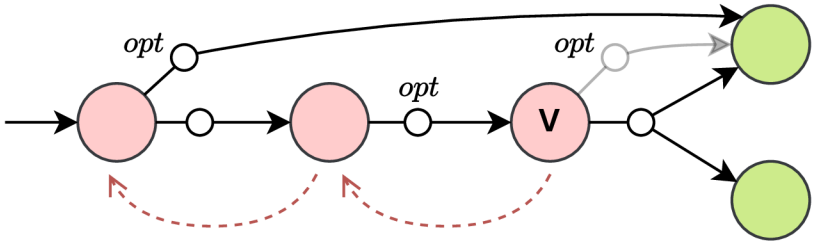


Figure 4.1: The principle of IDAR. When the optimal choice for some state is omitted, it and its predecessors are marked as vague.

Definition 25 (Vague state)**.** Let $\mathcal{M}^{\mathfrak{D}} = (S, s_0, \mathcal{R}^{\mathfrak{D}}, P)$ be a quotient MDP, $\mathcal{X} \subseteq \mathfrak{D}$ a subfamily of MCs, $\mathcal{C}_{\mathcal{X}}$ a set of compatible choices for $\mathcal{X}$ and σ_{opt} the optimal memoryless scheduler for the parent of $\mathcal{X}$. A state $s \in S$ is *vague* if at least one of the following conditions is met:

1. The optimal choice for s is omitted in $\mathcal{C}_{\mathcal{X}}$:

$$\sigma_{opt}(s) \notin \mathcal{C}_{\mathcal{X}},$$

2. At least one of the direct successors of s is vague:

$$\exists s' \in S : s' \in supp(P(s, \cdot)) \wedge \texttt{vague}(s').$$

According to Algorithm 1, AR initially analyses the entire family $\mathcal{R}^{\mathfrak{D}}$. Therefore, this family is the only one that does not have a parent. Model-checking results provide the optimal memoryless scheduler σ_{opt}, which allows for accessing the optimal choice for each state of $\mathcal{M}^{\mathfrak{D}}$. Assume that initially, the abstraction is too coarse and AR splits the family, for convenience, into 2 subfamilies $\mathcal{X}$ and $\mathcal{Y}$ according to a hole $\mathcal{H}$. Sets of compatible choices $\mathcal{C}$ (for the superfamily) and $\mathcal{C}_{\mathcal{X}}$ are equal, except for the choices connected with $\mathcal{H}$ and left in $\mathcal{C}_{\mathcal{Y}}$.

Initially, all states of $\mathcal{M}^{\mathfrak{D}}$ are considered non-vague. Our approach involves two stages of classifying states from non-vague to vague. In the first stage, we examine states where the number of actions decreased after splitting. Such a state remains non-vague for a child inheriting the optimal parent choice. Otherwise, if the optimal choice is omitted, the state becomes vague (the first condition from Definition 25). In the second stage, we iterate through all vague states obtained in the first stage and mark their predecessors as vague (the second condition). When we are no longer certain about the optimal choice in a state s, the uncertainty propagates to all states, where s is reachable. Figure 4.1 illustrates the described principle of marking states as vague.

Once the set of vague states $\mathcal{V}$ is identified, we need to use this information to create a mask based on the child's set of compatible choices. For vague states, all available choices are kept. For non-vague states, retaining all available choices is unnecessary when the optimal one is known for certain. In such cases, only the optimal choice is preserved. After splitting, each child becomes a parent, and the algorithm repeats the process. It is important to note that splitting of the subfamily has to be performed based on the full set of its compatible choices, not on the mask. Otherwise, there is a risk of overlooking some realisations.

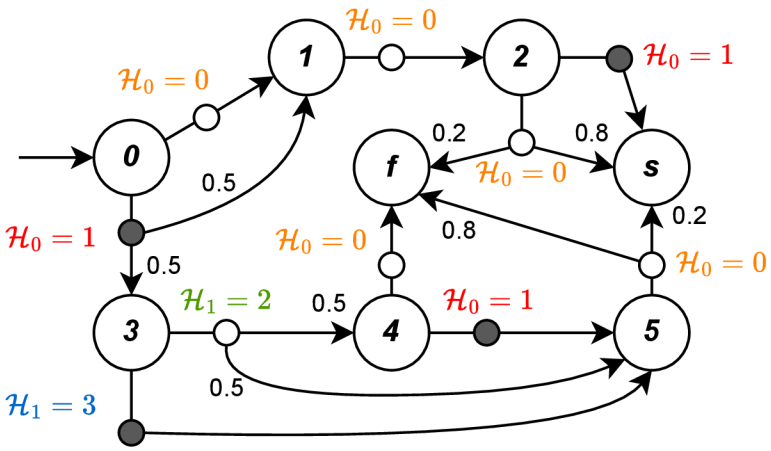


Figure 4.2: An example of quotient MDP with holes.

Algorithm 2 Using inheritance dependencies within families of FSCs for AR

Input: A family of MCs $\mathcal{X}$, a set of compatible choices $\mathcal{C}_\mathcal{X}$
Output: A mask $\mathfrak{M}$, or $\mathcal{C}_\mathcal{X}$ for the first iteration

1: **if** parent($\mathcal{X}$) **is** `None` **then**
2: **return** $\mathcal{C}_\mathcal{X}$ ▷ The mask for the superfamily is not changed
3: **end if**
4: $\sigma_{opt} \leftarrow$ parentScheduler($\mathcal{X}$)
5: $\mathcal{V} \leftarrow$ findVagueStates($\mathcal{X}$, $\mathcal{C}_\mathcal{X}$, σ_{opt}) ▷ A set of vague states
6: $\mathcal{V} \leftarrow$ findVagueReachable($\mathcal{V}$) ▷ Works with predecessors of each state
7: $\mathfrak{M} \leftarrow$ vagueToChoices($\mathcal{V}$, $\mathcal{X}$, $\mathcal{C}_\mathcal{X}$, σ_{opt})
8: **return** $\mathfrak{M}$

Algorithm 2 can be considered an extension of the *restrict* procedure from Algorithm 1 (row 6). It summarizes the described method and presents procedures resembling its possible implementation. As input, the algorithm takes a (sub)family $\mathcal{X}$ and a set of compatible choices $\mathcal{C}_\mathcal{X}$. If $\mathcal{X}$ is the superfamily, the mask is $\mathcal{C}_\mathcal{X}$. Otherwise, *parentScheduler* retrieves the parent's optimal scheduler. Procedures *findVagueStates* and *findVagueReachable* correspond to the two stages of classifying states from non-vague to vague. Finally, *vagueToChoices* reduces $\mathcal{C}_\mathcal{X}$, and the result is returned as a mask $\mathfrak{M}$.

This improvement reduces the size of the mask compared to the set of compatible choices and accelerates model checking for children. In general, the fewer vague states there are, the smaller the mask size is. For the MDP model checking with the PCTL syntax, the algorithmic complexity is polynomial in the size of MDP [28]. Therefore, each non-vague state of the quotient MDP significantly accelerates its model checking by eliminating the nondeterminism. The consistency of IDAR and AR follows directly from Theorem 1, proved in the following section.

Example 9. *Applying IDAR on a quotient MDP.*

Consider a hypothetical quotient MDP $\mathcal{M}^\mathfrak{D}$ from Figure 4.2 with two holes $\mathcal{H}_0, \mathcal{H}_1$ with domains $V_{\mathcal{H}_0} = \{0, 1\}$, $V_{\mathcal{H}_1} = \{2, 3\}$ and a feasibility synthesis problem $\varphi = \mathbb{P}_{\leq 0.3}(\Diamond\{f\})$. For states 1 and 5, $\mathcal{H}_0 = 1$ is a self-loop. In the first iteration of IDAR, a mask is the set of compatible choices $\mathcal{C}$. Model checking of $\mathcal{M}^\mathfrak{D}$ for $\mathcal{C}$ returns minimising and maximising

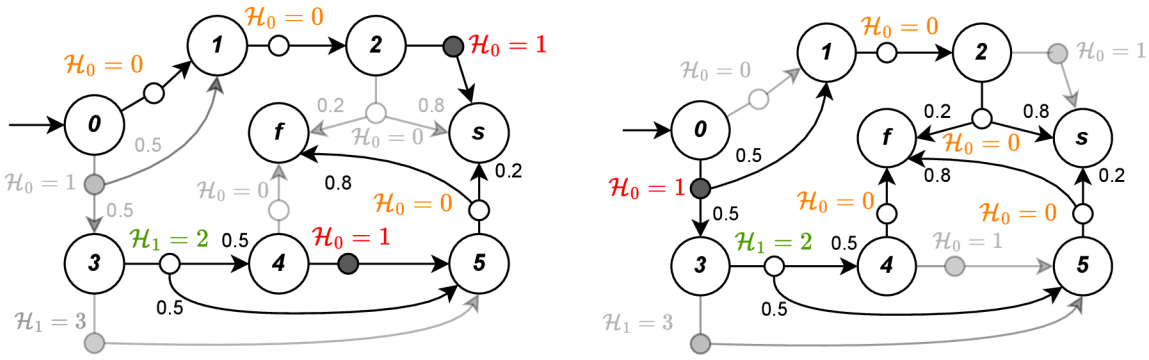


Figure 4.3: MCs induced by σ_{min} and σ_{max} for the quotient MDP from Figure 4.2.

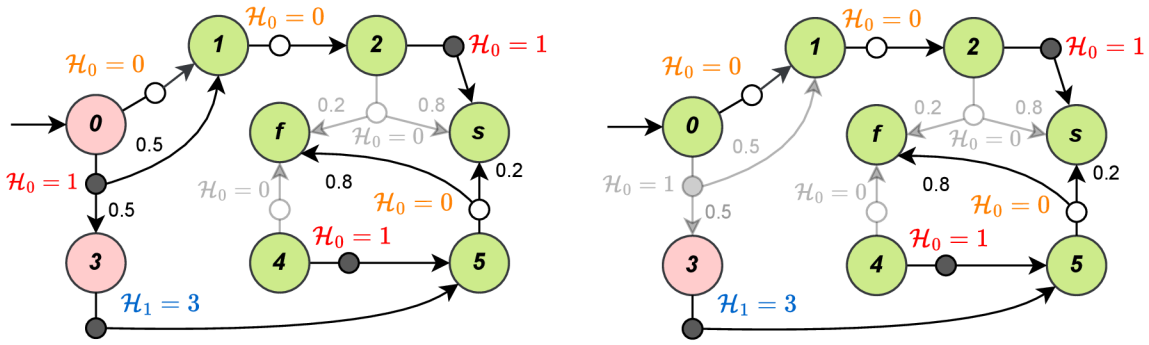


Figure 4.4: Classification of quotient MDP's states into vague/non-vague (IDAR, left) or affected/non-affected (EIDAR, right) and the resulting masks.

schedulers σ_{min} and σ_{max}. Figure 4.3 shows induced MCs $\mathcal{M}_{\sigma_{min}}$ (left) and $\mathcal{M}_{\sigma_{max}}$ (right). For state 3 in σ_{min}, both values of $\mathcal{H}_1$ are optimal, so the choice is selected randomly. The resulting bounds are $p_{min}(0) = 0$, $p_{max}(0) = 0.55$, and σ_{min} is not consistent, therefore, the abstraction is too coarse. Let $\mathcal{M}^{\mathfrak{D}}$ be split in $\mathcal{H}_1$ into $\mathcal{X}$ and $\mathcal{Y}$, where $\mathcal{C}_{\mathcal{X}} = \mathcal{C} \setminus \{3 : \mathcal{H}_1 = 2\}$ and $\mathcal{C}_{\mathcal{Y}} = \mathcal{C} \setminus \{3 : \mathcal{H}_1 = 3\}$.

For the child $\mathcal{X}$, the optimal choice was omitted only in state 3. Thus, in the first stage of IDAR, only state 3 is marked as vague. State 0 is its only predecessor, so after both stages of classification, there are two vague states, as shown in Figure 4.4 (left). For non-vague states, only choices from σ_{min} are preserved in $\mathfrak{M}_{\mathcal{X}}$. Note that the new upper bound $p_{max}(0) = 0.4$ is less than the previous one.

Consider that AR is a DFS algorithm (as implemented in PAYNT) and children of $\mathcal{X}$ are analysed earlier than $\mathcal{Y}$. Let $\mathcal{X}$ be split in $\mathcal{H}_0$ into $\mathcal{X}'$ (preserves $\mathcal{H}_0 = 0$ in each state) and $\mathcal{X}''$ ($\mathcal{H}_0 = 1$). All holes in $\mathcal{X}'$ are substituted by some value and $\mathfrak{M}_{\mathcal{X}'} = \mathcal{C}_{\mathcal{X}'}$. Model checking for $\mathcal{X}'$ provides $\mathbb{P}(\Diamond\{f\}) = 0.2$, satisfying φ. The resulting realisation is obtained by substitutions $\mathcal{H}_0 = 0$ and $\mathcal{H}_1 = 3$.

□

IDAR accelerates model checking by reducing the number of compatible choices. Because of that, sometimes IDAR can narrow the model-checking bounds. If the upper bound passes down the border λ after applying the mask, the number of AR iterations can also decrease. However, narrower bounds do not guarantee that all realisations of the subfamily

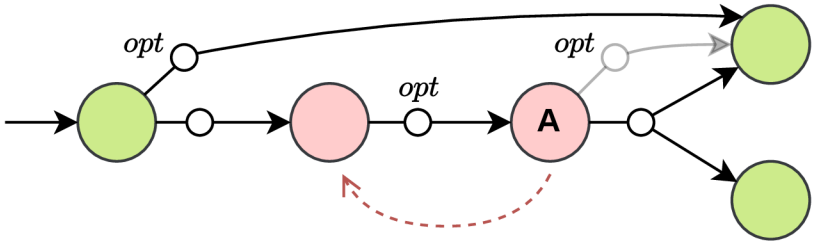


Figure 4.5: The principle of EIDAR. When is some state the optimal choice is omitted or leads to an affected state, it is marked as affected.

meet the specification. For the feasibility synthesis problem, a consistent σ_{min} is certainly a solution. Otherwise, we cannot accept the entire subfamily based on the bounds, provided by the mask. Each solution for the threshold or max/min synthesis problems should be double-checked.

4.2 Extended IDAR

In the second stage of classic IDAR, a state is considered vague if some of its successors lose the parent's optimal choice. In this section, we propose an extended version of IDAR (*EIDAR*). It takes into account the location of optimal choices not only for states that lost their optimal choices but also for their predecessors.

Definition 26 (Affected state and affected choice)**.** Let $\mathcal{M}^{\mathfrak{D}} = (S, s_0, \mathcal{R}^{\mathfrak{D}}, P)$ be a quotient MDP, $\mathcal{W} \subseteq \mathfrak{D}$ a subfamily of MCs, $\mathcal{X} \subset \mathcal{W}$ a child of $\mathcal{W}$, $\mathcal{C}_{\mathcal{W}}$ and $\mathcal{C}_{\mathcal{X}}$ the corresponding sets of compatible choices and σ_{opt} the optimal memoryless scheduler for $\mathcal{W}$. A *state* $s \in S$ is *affected* if its optimal choice for the parent $\sigma_{opt}(s)$ is affected. A *choice* $q \in \mathcal{C}_{\mathcal{W}}$ is *affected* if $\exists s \in S : \sigma_{opt}(s) = q$ and at least one of the following conditions is met:

1. q is omitted in $\mathcal{C}_{\mathcal{X}}$:
$$q \notin \mathcal{C}_{\mathcal{X}},$$
2. q leads to an affected state:
$$\exists s \in S : s \in \mathit{supp}(P(s_o, q)) \wedge \texttt{affected}(s),$$
where s_o is the origin state of q.

Initially, all choices and states are considered non-affected. The EIDAR approach also consists of two stages. Let $\mathcal{Q}$ be a set of affected choices, initially empty. In the first stage, all optimal and omitted choices are added to $\mathcal{Q}$ (the first condition from Definition 26). In the second stage, EIDAR takes a choice q from $\mathcal{Q}$ and marks its origin state s_o as affected. Then, all *unique* optimal choices leading to s_o are added to $\mathcal{Q}$ (the second condition). When $\mathcal{Q}$ becomes empty, we obtain a set $\mathcal{A}$ of all affected states. Figure 4.5 illustrates the described principle of marking states as vague. The mask of result choices $\mathfrak{M}$ is created in the same way as in Algorithm 2. Unlike IDAR, its extended version creates the set of affected states by going directly through optimal choices to their origin states. EIDAR significantly reduces the number of affected states compared to the vague ones. Therefore, the resulting mask for EIDAR is smaller, contributing to a more accelerated model checking.

Algorithm 3 Extended IDAR

Input: A family of MCs $\mathcal{X}$, a set of compatible choices $\mathcal{C}_\mathcal{X}$
Output: A mask $\mathfrak{M}$, or $\mathcal{C}_\mathcal{X}$ for the first iteration

if parent($\mathcal{X}$) **is** `None` **then**
  return $\mathcal{C}_\mathcal{X}$ ▷ The mask for the superfamily is not changed
end if
$\mathcal{A} \leftarrow \emptyset$ ▷ A set of affected states
$\sigma_{opt} \leftarrow$ parentScheduler($\mathcal{X}$)
$\mathcal{Q} \leftarrow$ optOmittedChoices($\mathcal{X}$, $\mathcal{C}_\mathcal{X}$, σ_{opt}) ▷ A set of affected choices
while $\mathcal{Q} \neq \emptyset$ **do**
  $q \leftarrow$ any($\mathcal{Q}$)
  $\mathcal{Q} \leftarrow \mathcal{Q} \setminus \{q\}$
  $s_o \leftarrow$ stateOrigin(q)
  $\mathcal{A} \leftarrow \mathcal{A} \cup \{s_o\}$
  $\mathcal{L} \leftarrow$ optLeadingChoices(s_o) ▷ A set of all optimal choices leading to s_o
  while $\mathcal{L} \neq \emptyset$ **do**
    $l \leftarrow$ any($\mathcal{L}$)
    $\mathcal{L} \leftarrow \mathcal{L} \setminus \{l\}$
    if not wasInQ(l) **then**
      $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{l\}$
    end if
  end while
end while
$\mathfrak{M} \leftarrow$ affectedToChoices($\mathcal{A}$, $\mathcal{X}$, $\mathcal{C}_\mathcal{X}$, σ_{opt}) ▷ Similar to vagueToChoices from IDAR
return $\mathfrak{M}$

Algorithm 3 summarises the described changes. The input and output remain the same as in Algorithm 2. Procedure *optOmittedChoices* corresponds to the first stage of EIDAR. The set of all optimal choices leading to the state s_o is obtained using *optLeadingChoices*. The resulting mask is computed using *affectedToChoices* that is identical to *vagueToChoices* from IDAR. It is important to note that each choice can only enter $\mathcal{Q}$ once; otherwise, the algorithm might get into an endless loop.

Observation 1. After applying EIDAR for a subfamily of MCs, the MC induced by its optimal scheduler cannot have a transition from a non-affected state to an affected state. Each non-affected state has a single choice in the mask – the optimal one from the parent's scheduler. According to Algorithm 3, the origin state of such choice should have become affected earlier.

Theorem 1. *Model checking of a quotient MDP for a given subfamily of MCs finds the same optimal result (probability or reward) for classic AR and EIDAR.*

Proof. Let $\mathcal{M}^\mathfrak{D} = (S, s_0, \mathcal{R}^\mathfrak{D}, P)$ be a quotient MDP, $\mathcal{W} \subseteq \mathfrak{D}$ a subfamily of MCs, $\mathcal{X} \subset \mathcal{W}$ a child of $\mathcal{W}$, $\sigma_\mathcal{W}$ the optimal scheduler for $\mathcal{W}$. Denote $\sigma^\mathrm{A}_\mathcal{X}$ and $\sigma^\mathrm{E}_\mathcal{X}$ the optimal schedulers for $\mathcal{X}$ obtained by AR and EIDAR, respectively. According to Observation 1 there are two possible scenarios for EIDAR:

I. Only non-affected states are reachable in the MC $\mathcal{M}_{\sigma^\mathrm{E}_\mathcal{X}}$ induced by $\sigma^\mathrm{E}_\mathcal{X}$ (Figure 4.6, left). It means, that $\sigma^\mathrm{E}_\mathcal{X}$ and $\sigma_\mathcal{W}$ are identical and the optimum does not change.

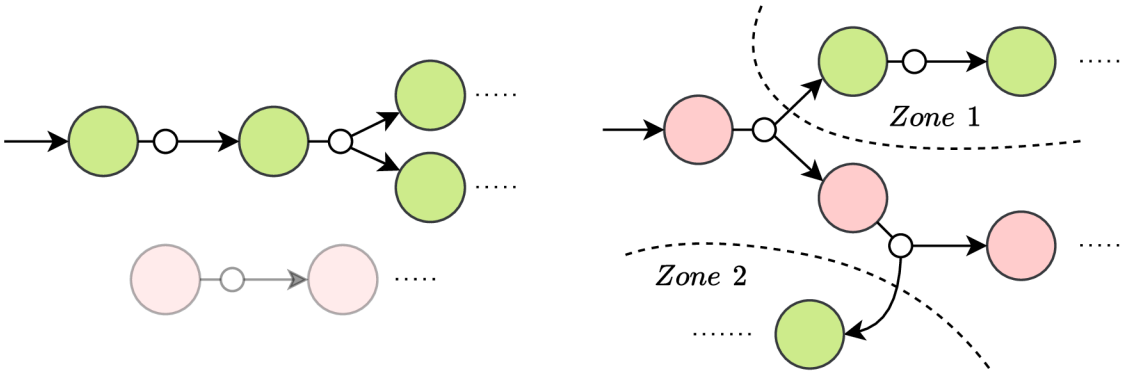


Figure 4.6: Different scenarios of the affected states' location for the MC induced by the optimal scheduler using EIDAR.

Since all choices of the resulting mask for EIDAR are also available in the AR's mask, both these methods provide the same optimum.

II. The initial state s_0 is affected, and there is an unlimited number of „non-affected zones" in $\mathcal{M}_{\sigma_{\mathcal{X}}^{\mathrm{E}}}$ (Figure 4.6, right). Each zone contains only non-affected states. Similar to scenario I., bounds p_{max} and p_{min} for states in these zones are preserved from $\sigma_{\mathcal{W}}$. It means, that each non-affected state for EIDAR and AR contains the same optimal results. Since all compatible choices are preserved in each affected state, the final optimum for s_0 is the same for EIDAR and AR. Note that although the optimal results are equal, the provided FSCs are not necessarily equal. AR could find an FSC with the same value, but different selected choices – transitions from non-affected states to the affected ones are not blocked for AR.

Thus, EIDAR is consistent with AR. Since the mask for EIDAR is a subset of the IDAR's mask, IDAR is also consistent with AR. □

Example 10. *Applying EIDAR on a quotient MDP.*

Consider the quotient MDP $\mathcal{M}^{\mathfrak{D}}$ and the feasibility synthesis problem $\varphi = \mathbb{P}_{\leq 0.3}(\Diamond\{f\})$ from Example 9. The first iteration of EIDAR is similar to IDAR and AR. It produces the same schedulers σ_{min}, σ_{max} and the same subfamilies $\mathcal{X}$ and $\mathcal{Y}$. The only optimal and not compatible choice for $\mathcal{X}$ is $3 : \mathcal{H}_1 = 2$. Therefore, after the first stage of EIDAR, $\mathcal{A}$ contains only state 3. Given that σ_{min} from Figure 4.3 (left) is the optimal scheduler and it does not include the choice $0 : \mathcal{H}_0 = 1$, there are no states marked as affected during the second stage. As a result, $\mathcal{A} = \{3\}$ (Figure 4.4, right) and the choice $0 : \mathcal{H}_0 = 1$ is omitted from the mask $\mathfrak{M}_{\mathcal{X}}$.

The result of applying $\mathfrak{M}_{\mathcal{X}}$ on $\mathcal{M}^{\mathfrak{D}}$ is an MC, yielding $\mathbb{P}(\Diamond\{f\}) = 0$, thus satisfying φ. However, as mentioned in the previous section, this does not guarantee that all realisations of $\mathcal{X}$ meet φ. There are two possible realisations in $\mathcal{X}$: $r_1(\mathcal{H}_1) = r_2(\mathcal{H}_1) = 3$, $r_1(\mathcal{H}_0) = 0$ and $r_2(\mathcal{H}_0) = 1$. Model-checking results for r_1 and r_2 are 0.2 and 0, respectively. They both meet given φ and can be accepted as possible solutions. If the value of λ was 0.1 instead of 0.3, accepting both r_1 and r_2 would be inappropriate. Compared to IDAR, EIDAR found a suitable realisation after just one splitting.

□

Extended IDAR better minimises the mask's size than IDAR. As will be shown in more detail in Chapter 5, in some models the number of affected states is 37 times less than the number of vague states. Further in this work, the phrase „(E)IDAR“ serves to avoid referring to a specific version of the inheritance dependencies algorithm.

4.3 Smart EIDAR

The experiments described in Chapter 5 show that (E)IDAR does not always accelerate the inductive synthesis. Sometimes, due to the different topology of POMDP models, the number of vague/affected states of its quotient MDP is so large (around 95%) that the classic AR terminates faster. The time spent on classifying states is often longer than the time saved by (E)IDAR compared to AR. And yet, models with a small percentage of vague/affected states and a large average number of choices per vague/affected state showed positive results for (E)IDAR. An extension called Smart EIDAR (*SEIDAR*) is designed to save the user from manually choosing between AR, IDAR and EIDAR when working with any POMDP model.

IDAR and EIDAR complement the *restrict* procedure from Algorithm 1. Since all three described methods produce equally valid masks, although, with different sizes, the inductive synthesis can *switch* the method in different iterations during its runtime. The main parameters for making this decision are:

1. *The size of the superfamily.* For smaller families of FSCs, using (E)IDAR is not profitable, since their model checking time is already short, compared to the overall synthesis time.

2. *Percentage of vague/affected states.* The lower the percentage – the smaller the mask.

3. *The number of selected choices per vague/affected state.* Effectively combines the first two parameters. The larger this number, the larger the quotient MDP size and the smaller the number of marked states.

IDAR and EIDAR require auxiliary structures for their run, so switching between these methods would take some time to initialize the structures. The synthesis also cannot be initialised with classic AR, as it is impossible to determine the above parameters in AR. Switching from EIDAR to AR showed the best results among all available options. Implementation of SEIDAR in PAYNT first analyses 20% of family members (or runs a maximum of 100 iterations) on EIDAR. Collected statistics of the parameters are used to decide if PAYNT should switch to AR or remain in EIDAR. For convenience, switching takes place only once. The values of the parameters for PAYNT were determined based on the conducted experiments. In this paper, SEIDAR from Section 5.5 is assumed, but there is an unlimited number of possible variations for its realisation.

Chapter 5

Experimental Evaluation

This chapter provides the experimental evaluation of the AR improvements described in Chapter 4. First, the main aspects of implementation and the POMDP models, used for the testing, are described. Then follows a series of experiments comparing AR, IDAR, EIDAR and SEIDAR. As a result, the following research questions are answered:

Q1: Do conducted experiments confirm the consistency of (E)IDAR and AR? Before studying how our methods accelerate the synthesis of FSC, it is necessary to perform the test of correctness for our implementation. Our methods must yield FSCs with equivalent optimal probabilities/rewards compared to AR.

Q2: What impact do our methods have on the synthesis? As shown in Examples 9 and 10, (E)IDAR affects the inductive synthesis in a specific way. How do the narrowed bounds provided by model checking of a quotient MDP affect the number of iterations of the selected algorithm?

Q3: Does EIDAR perform better compared to IDAR? Although in theory, EIDAR outperforms IDAR, this is not always the case in practice. Computationally, EIDAR is more complex and in some cases, it affects its efficiency. The overall speedup, along with the speedup of model building and model checking, was measured on an existing benchmark containing several models. The answer to this question also affects which method should form the basis of SEIDAR.

Q4: Does SEIDAR outperform AR? SEIDAR saves the user from choosing which method to use for a particular model. Therefore, the main contribution of this work is SEIDAR. Is it reasonable to always prefer SEIDAR over AR?

5.1 Implementation

All described improvements are implemented in PAYNT (see Section 3.3) using Python with C++ bindings for efficiency. The mask of selected choices $\mathfrak{M}$ must be a C++ bit vector due to the existing implementation of PAYNT. Therefore, when the program needs to manipulate a bit vector directly, calling the appropriate function from C++ is more profitable. Also, as practice shows, frequent access to the transition matrix of the quotient MDP is time-consuming, since it is necessary to iterate through each state, each row (choice of the state) and each column of the matrix (destination states). Hence, the transition matrix is also accessed mostly in C++ code.

Model	$\|S\|$	$\|A\|$	$\|Z\|$	Spec.
web-mall	8	20	5	R_{max}
grid-avoid-4-0	17	59	4	P_{max}
4x3-95	22	82	9	R_{max}
mini-hall2	27	77	12	R_{max}
query-s2	36	70	6	R_{max}
refuel-06	208	565	50	P_{max}

Model	$\|S\|$	$\|A\|$	$\|Z\|$	Spec.
drone-4-1	1226	2954	384	P_{max}
hallway2	1500	7492	20	R_{max}
rocks-12	6553	32k	1645	R_{min}
refuel-20	6834	25k	174	P_{max}
rocks-16	11k	54k	2761	R_{min}
LRV	18k	105k	2242	R_{min}
network-prio	19k	34k	4909	R_{max}

Table 5.1: Summary about the selected benchmark of POMDPs.

To make our algorithms work, it is necessary to initialise auxiliary structures. For IDAR, this is a vector storing sets of direct predecessors for each quotient MDP's state. For EIDAR, this is a mapping from choices to their corresponding state. In classic PAYNT, all available choices are numbered from zero and only a mapping from states to choices is available. Since accessing the transition matrix is necessary to create both structures, the corresponding functions are called from the C++ binding.

The second stage of IDAR, which includes iterating through all vague states and marking their predecessors as vague, is implemented as a DFS procedure. As for EIDAR, the set of affected choices $\mathcal{Q}$ from Algorithm 3 is represented by a queue. An auxiliary bit vector stores the information about choices that already visited $\mathcal{Q}$ to ensure the uniqueness of the choices in the queue.

The key procedure of (E)IDAR is *affectedToChoices* (*vagueToChoices*). Initially, the mask $\mathfrak{M}$ is created as a bit vector of ones. For non-affected (non-vague) states, the value of all non-optimal choices in the mask is reset to zero. Since of all the choices, only those compatible with the family are needed, a bitwise AND operation is performed between the mask and the vector of compatible choices. The resulting mask is used later for model building and model checking of the family. A detailed manual on running PAYNT, including new flags for implemented improvements, is provided in Appendix B.

5.2 Selected Benchmark

The benchmark was run on a single core on Intel i5-10300H @2.5GHz CPU and 16GB of RAM. It includes models of varying complexity and size (one model can have multiple instances) to check whether SEIDAR can handle any problem no worse than AR. All selected POMDP models, except for the *LRV*, were taken from [9, 11]. Table 5.1 summarises the number of states $|S|$, the total number of actions $|A| := \sum_s |Act(s)|$, the number of observations $|Z|$ and the specification for each included POMDP model. Unless mentioned otherwise, we consider either max/min reachability probability P or max/min expected total reward R (in the PRISM notation). In this and the following tables, all measurements are rounded to a maximum of three decimal places for convenience.

The synthesis problem is given by the model instance, which is formed by the topology of the POMDP model, the selected specification and the amount of memory k. All model instances included in the benchmark can be divided into 2 groups – those where we can find the best FSC for a given memory in a reasonable time (within a few minutes) and those where we cannot do so (the synthesis lasts up to several years or more). For experimental

Model	Memory	Optimum P/R			
		AR	IDAR	EIDAR	SEIDAR
web-mall	3	6.685	6.685	6.685	6.685
grid-avoid-4-0	5	0.929	0.929	0.929	0.929
4x3-95	2	1.468	1.468	1.468	1.468
mini-hall2	2	2.687	2.687	2.687	2.687
query-s2	2	486.694	486.694	486.694	486.694

Table 5.2: Comparison between optimal values for model instances, whose synthesis takes a short time (without explicitly specifying the number of iterations).

purposes, handling the second group of instances requires manually limiting the number of iterations.

As will be seen from the results of the experiments, our methods consistently accelerate models with a large design space. To better reveal the potential of SEIDAR, an *LRV* model was created, inspired by Example 4. The total number of actions in *LRV* is 3 times greater than in *network-prio-2-8-20*, which leads to better performance of SEIDAR. In this interpretation of the Lunar Roving Vehicle, there are no walls and boundaries, but minerals have to be collected. There are 3 minerals in a 9×9 field, each of which is „good“ with a probability of 0.6. The robot has a sensor that allows scanning of each mineral, and the closer they are to each other, the more accurate the measurements. The main task of the LRV is to collect at least two good minerals, otherwise, it gets a penalty (reward). It is also penalised for every collected „bad“ mineral. According to the specification, the robot has to reach the final cell with minimal penalty.

5.3 Consistency to AR and Impact on the Synthesis

Since the number of iterations needed to terminate (E)IDAR and AR is not always the same, its explicit restriction may lead to different resulting FSCs. Therefore, the consistency check of (E)IDAR can only be carried out on the first group of model instances. Table 5.2 lists the results obtained for all four compared approaches (parameters for SEIDAR will be described in Section 5.5). Different values of k were tested for each included model, and only one is shown in the table.

Q1: All new approaches proved to be consistent with classic AR, thereby confirming Theorem 1. The optimal result does not depend on the chosen method – the implementation can be considered correct.

The impact of inheritance dependencies on the synthesis can be explored in two ways: i) how the number of iterations changed for model instances from the first group, and ii) how the optimal result changed on model instances with a limited number of iterations (second group). Table 5.3 answers the first question. For each small model instance, there is a change in the number of necessary iterations. Values obtained by EIDAR are always different from the AR's values. SEIDAR, since it switches to AR in these instances, preserved its behaviour. The biggest change is observed in the *grid-avoid-4-0* model: the number of iterations for EIDAR is reduced by 30%.

Even though model checking for (E)IDAR can provide narrower bounds, on some models (*4x3-95*, *mini-hall2*, *query-s2*) the number of iterations increased. The reason for this lies

Model	Memory	Iterations			
		AR	IDAR	EIDAR	SEIDAR
web-mall	3	43747	35029	35029	43747
grid-avoid-4-0	5	107202	107202	75452	107202
4x3-95	2	911	913	913	911
mini-hall2	2	54403	55111	55111	54403
query-s2	2	665	675	667	665

Table 5.3: Comparison between the number of iterations for model instances, whose synthesis takes a short time.

Model	Spec.	Memory	Iterations	Optimum P/R			
				AR	IDAR	EIDAR	SEIDAR
refuel-06	P_{max}	3	20000	0.051	0.051	0.032	0.032
hallway2	R_{max}	1	200	0.002	0.023	0.023	0.002
			2000	0.025	0.026	0.026	0.025
refuel-20	P_{max}	1	1000	0.001	0.001	0.019	0.002
			10000	0.001	0.001	0.019	0.002
rocks-16	R_{min}	3	100	46	no	no	no

Table 5.4: Comparison between optimal values for model instances, whose number of iterations is explicitly limited. Experiments, where no P/R value satisfying the specification was found within the specified number of iterations, are marked with „no“.

in the specifics of Storm and is partially described in Theorem 1. If, as a result of model checking, several FSCs are optimal at once, Storm takes any of them. The change in the AR algorithm at a deep level influenced this choice. At some point, Storm chooses a different FSC, which changes the further splitting of the family and the entire outcome of the synthesis. Therefore, we cannot be sure how the number of iterations would change without running the experiment.

Table 5.4 demonstrates the impact of limiting the number of iterations on the obtained optimum for larger instances. In *refuel-20*, EIDAR finds an FSC that is 19 times better than the one found by AR. However, (E)IDAR does not always find a more optimal FSC compared to AR. For *rocks-16*, it didn't manage to find any suitable FSC. In other experiments not listed in this table (e.g. *rocks-12*, *LRV*), the optimal result is equal for all methods. Hence, we also cannot be sure that each method would return the same optimal FSC within the same number of iterations.

Q2: Experiments show that the impact of inheritance dependencies on the synthesis is ambiguous. We cannot claim that each approach would produce the same optimum for the same number of iterations and vice versa.

5.4 Evaluation of the Synthesis Time for IDAR and EIDAR

Initially, the main task of this work was to create a method that accelerates model checking (MC) of families of FSCs. However, during the experiments it turned out that the size of

Model	k	Iter.	AR time (s)			M.	Speedup			t_{M} (s)
			overall	MB	MC		overall	MB	MC	
grid-avoid	5	-	39.4	4.94	6.51	I	0.68	1.03	1.01	19.13
						E	1.11	1.49	1.46	9.27
4x3-95	2	-	0.42	0.05	0.12	I	0.98	1.14	1.16	0.05
						E	0.89	1.09	1.14	0.08
mini-hall2	2	-	22.09	2.32	4.71	I	0.92	1.08	1.09	3.3
						E	0.86	1.04	1.08	4.67
refuel-06	3	20k	31.13	5.33	8.7	I	0.79	1.19	1.06	11.82
						E	1.07	1.61	1.57	9.04
drone-4-1	1	1k	12.75	0.64	5.94	I	0.92	1.07	1.04	1.74
						E	0.95	1.11	1.07	1.47
hallway2	1	200	18.01	1.88	12.51	I	1.02	1.12	1	0.87
						E	0.89	1.13	1.02	3.59
		2k	120.3	14.61	83.22	I	0.77	0.93	0.76	9.07
						E	0.64	0.93	0.76	37.68
rocks-12	1	1k	14.67	2.16	5.97	I	1.08	1.61	1.73	3.09
						E	1.59	3.32	3.73	3.53
		10k	139.64	21.12	59.46	I	1.16	1.76	1.82	29.72
						E	1.7	3.6	4.09	34.78
refuel-20	1	1k	22.04	0.9	1.66	I	0.78	1.01	1	6.09
						E	0.75	0.75	0.63	5.62
		10k	191.78	9.2	15.52	I	0.81	1.14	1.08	55.12
						E	0.66	0.75	0.6	57.85
rocks-16	1	1k	25.22	3.61	9.88	I	0.94	1.21	1.22	5.37
						E	0.95	1.53	1.65	9.78
LRV	1	500	38.89	6.37	18.67	I	1.5	1.83	1.99	4.51
						E	1.38	1.82	1.92	8.17
	3	40	42.43	5.03	15.1	I	1.84	1.99	2.19	1.22
						E	1.75	2.01	2.14	3.26
network-prio	1	100	19.14	1.11	1.27	I	1.01	1.17	1.25	1.75
						E	0.96	1.18	1.28	3.08
	5	40	52.01	7.3	11.14	I	1.24	1.48	1.59	4.04
						E	1.18	1.54	1.62	8.39

Table 5.5: Evaluation of speedups for IDAR and EIDAR. Hyphen (-) indicates that the experiment was completed without a limit on iterations. „M.“ represents a method, t_{M} refers to the overall time spent by the method. Speedups are calculated as the AR time divided by the time of the corresponding method. A speedup less than 1 implies a slowdown. Models *web-mall* and *query-s2* are not listed in the table, as their result is similar to *4x3-95*.

the mask also affects the operation of other parts of the synthesis: model building (MB) and various optimisations in PAYNT. Table 5.5 shows the resulting speedups. In most examples, time t_{M} for EIDAR is longer than for IDAR. However, this does not prevent EIDAR from achieving better speedups (*rocks-12*). The worst overall speedups (slowdowns) are 0.68 (*grid-avoid-4-0*) for IDAR and 0.64 (*hallway2*) for EIDAR. Since the overall synthesis time

Model	k	Iter.	% affected		SC/AS	
			IDAR	EIDAR	IDAR	EIDAR
grid-avoid-4-0	5	-	97	46.62	2.32	3.76
4x3-95	2	-	32.59	31.26	4.87	4.98
mini-hall2	2	-	32.3	30.38	4.46	4.51
refuel-06	3	20k	55.04	40.23	8.45	13.89
drone-4-1	1	1k	94	86	1.55	1.54
hallway2	1	200	43.55	43.55	1.83	1.83
		2k	42.81	42.81	1.8	1.8
rocks-12	1	1k	35.19	19.99	122.74	604.19
		10k	36.77	19.17	106.97	538.44
refuel-20	1	1k	66.93	50.25	2.87	3.56
		10k	65.95	46.31	2.96	3.29
rocks-16	1	1k	37.98	2.54	180.68	852.84
LRV	1	500	27.36	0.74	262.18	1849.88
	3	40	9.41	1.15	5899.88	46792.19
network-prio-2-8-20	1	100	52.28	52.84	78.33	54.41
	5	40	43.05	39.31	3148.47	7896.19

Table 5.6: Calculating the percentage of vague/affected states („% affected“) and the number of Selected Choices per vague/Affected State („SC/AS“) for IDAR and EIDAR. Columns „% affected“ and „SC/AS“ show the average values for all run iterations.

includes t_M, the speedups for MB and MC are generally better than the overall acceleration. The best overall speedup (1.84) is observed in *LRV*. Although the overall speedup for *rocks-12* is only 1.7, MB and MC terminated 3.6 and 4 times faster, respectively. Increasing the number of iterations enhances the effect of the corresponding experiment with fewer iterations. If there is a slowdown (speedup) at 1000 iterations, the synthesis terminates even slower (faster) at 10000 iterations (*hallway2*, *rocks-12*, *refuel-20*). Table 5.6 compares the percentage of vague/affected states and the number of selected choices per vague/affected state for the same set of model instances. In the absolute majority of results, the number of affected states is less than vague ones. For *LRV* with $k = 1$, „% affected“ decreased by 37 times. But in *network-prio-2-8-20*, there is a slight increase in the percentage of affected states, since Storm at some point chooses a different FSC. Generally, the value of „SC/AS“ for EIDAR is greater than for IDAR, especially in larger models.

Q3: Both methods accelerate the synthesis. On average, IDAR speeds up the overall time by 1.03 times, MB by 1.31 times and MC by 1.33 times. For EIDAR, the average speedups are 1.09, 1.59, and 1.65, respectively. However, we cannot claim that one of the methods is better than the other. Even though there are model instances on which IDAR (or even AR) copes faster, EIDAR performs better on average. Therefore, it was EIDAR that formed the basis of SEIDAR.

5.5 Selection of Parameters for SEIDAR

SEIDAR allows PAYNT to decide whether to switch to AR or remain in EIDAR after a few iterations. The more iterations we perform to collect statistics, the more precise

Model	k	Iter.	Overall sp.	Thresh.	Fam. size	% affected	SC/AS
grid-avoid	5	-	1.11	100	$\leq$	52.1	5.5
4x3-95	2	-	0.89	100	$\leq$	28.13	5.2
mini-hall2	2	-	0.86	100	$\leq$	40.34	4.28
refuel-06	3	20k	1.07	100	$>$	42.29	13.87
drone-4-1	1	1k	0.95	100	$>$	86	1.62
hallway2	1	2k	0.89	100	$\leq$	42.3	1.84
rocks-12	1	1k	1.59	100	$>$	8.19	1144.07
refuel-20	1	1k	0.75	100	$>$	52.86	4.91
rocks-16	1	1k	0.95	100	$>$	2.09	1553.16
LRV	1	500	1.38	100	$>$	0.74	2074.62
	3	40	1.75	8	$>$	1.43	40644.47
network-prio	1	100	0.96	20	$>$	44.11	3.42
	5	40	1.18	8	$>$	73.71	13.35

Table 5.7: Calculating the size of the superfamily („Fam. size“), the percentage of affected states and the number of Selected Choices per Affected State for EIDAR with the number of iterations, given by the threshold. A common logarithm of the size of the superfamily is compared with 15 ($\leq$ or $>$). Its full value is not provided due to its enormous ranges (up to 21822 after applying the common logarithm for *network-prio-2-8-20* with $k = 5$). Columns „% affected“ and „SC/AS“ show the average values for all run iterations.

the results become. On the other hand, the fewer iterations we perform before switching, the greater the effect it will have on the synthesis. Switching occurs at the threshold of 100 iterations, or after analysing 20% of family members (PAYNT counts this value). In the larger model instances where even 100 iterations take a lot of time and memory, we lowered this threshold. For experimental purposes, where we manually limit the number of iterations, the program also switches after performing 20% of this limit. The measured parameters are described in Section 4.3. If the superfamily's size is too small, SEIDAR switches to AR right after the first iteration. This prevents wasting extra time in EIDAR and allows immediately following the optimal method.

Table 5.7 lists the obtained measurements. In an ideal scenario, we want to switch to AR in those examples where the overall speedup is less than 1. Otherwise, it is more profitable to remain in EIDAR. However, there are slowed-down instances (e.g. *rocks-16*), where all the conditions are met to stay in EIDAR: a big superfamily, a small percentage of affected states and a large value of „SC/AS“. On the contrary, there are also accelerated instances with small family sizes (16M for *grid-avoid-4-0*, which is 100 times smaller than that for *4x3-95*). For this reason, it is impossible to choose the values of parameters so that the best approach is selected for each model. The main task of SEIDAR is to preserve the advantages of EIDAR (e.g. a big speedup of *rocks-12*) and eliminate its disadvantages (a slowdown of *hallway2*).

In *refuel-20* and *network-prio-2-8-20* ($k = 1$), a switch to AR is necessary despite their large family size. Therefore, we need the third parameter, „SC/AS“, as its value is relatively small in these examples. The percentage of affected states is the highest for *drone-4-1*, and its „SC/AS“ is low. However, there are also other models not presented in this chapter, such as *web-mall* ($k = 40$), where a high „% affected“ (96) was obtained together with a

Model	k	Iter.	SEIDAR speedup			Switched?
			overall	MB	MC	
grid-avoid-4-0	5	-	1.01	1.01	1.02	Yes
4x3-95	2	-	1	1	1.03	Yes
mini-hall2	2	-	1	1	1	Yes
refuel-06	3	20k	1.04	1.56	1.53	No
drone-4-1	1	1k	0.98	0.99	1	Yes
hallway2	1	200	1.03	1.02	1.03	Yes
		2k	1.03	1.03	1.03	
rocks-12	1	1k	1.6	3.4	3.76	No
		10k	1.72	3.65	4.13	
refuel-20	1	1k	1.32	1.06	0.95	Yes
		10k	1.39	1.14	0.99	
rocks-16	1	1k	0.95	1.54	1.66	No
LRV	1	500	1.39	1.84	1.95	No
	3	40	1.65	1.91	2.04	
network-prio-2-8-20	1	100	0.96	1	0.99	Yes
	5	40	1.17	1.53	1.57	No

Table 5.8: Evaluation of speedups for SEIDAR. The last column shows whether the switch to AR occurred.

high value of „SC/AS“ (36). Hence, including „% affected“ as a parameter is also crucial for achieving better efficiency.

As a result, the following thresholds are selected: 85 for „% affected“ and 5.5 for „SC/AS“. The whole algorithm for SEIDAR can be described as follows. It starts its operation in EIDAR and if the value of a common logarithm of the size of the superfamily is less than or equal to 15, it immediately switches to AR. Otherwise, after 100 iterations or a completed analysis of 20% of family members, the values of other parameters are examined. If the percentage of affected states is greater than 85 or the number of selected choices per affected state is less than 5.5, PAYNT switches to AR; if not, it remains in EIDAR. Following this algorithm, only *grid-avoid-4-0* and *rocks-16* are not behaving according to the ideal scenario.

Table 5.8 shows the final comparison of SEIDAR and AR. In most experiments, higher speedups are observed compared to Table 5.5. The most interesting result is obtained on *refuel-20*. For EIDAR, this model gave one of the worst slowdowns (0.66), but switching to AR sped it up to 1.39. Changing the methods yielded a better result than applying them separately. So far, it is unclear what led to this result, but the reason again may lie in the algorithm for choosing the optimal FSC in Storm.

The worst slowdown, obtained by EIDAR for *hallway2* (0.64), was converted into a slight speedup in SEIDAR. In *LRV*, where switching did not occur, the speedup decreased by 0.1 compared to EIDAR due to the experiments' accuracy. In experiments where switching to AR happened after the first iteration due to the small size of the superfamily, the acceleration is close to 1. For *4x3-95* and *mini-hall2*, this helped to eliminate the slowdown, but for *grid-avoid-4-0* it reduced the speedup instead. Figure 5.1 compares the overall

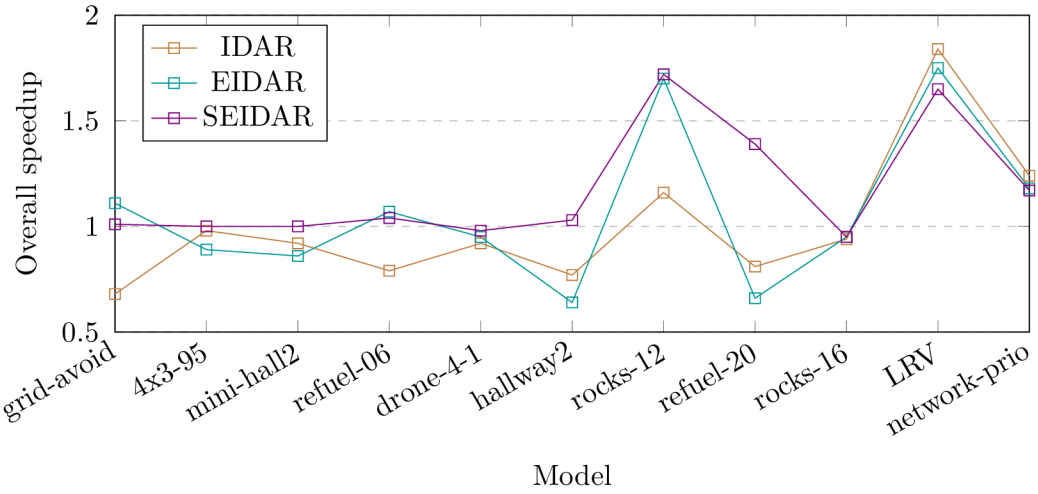


Figure 5.1: Comparison of overall speedups for IDAR, EIDAR and SEIDAR.

speedups for all described approaches. In models with several model instances, the value obtained for a larger number of iterations (or larger k) is used in the plot.

Q4: On average, SEIDAR speeds up the overall synthesis time by 1.2 times, MB by 1.54 times and MC by 1.61 times. Based on the experiments, it is reasonable for the user to always choose SEIDAR over AR. The worst overall slowdown (0.95) is negligible compared to the best speedup (1.72).

5.5.1 Solving Feasibility Synthesis Problem with SEIDAR

As Table 5.4 shows, in some model instances with a limited number of iterations, our method can find an FSC, significantly better than the one found by AR. This led us to the idea of conducting additional experiments, where the feasibility synthesis problem would replace the max/min synthesis problem. For *hallway2*, the corresponding specification is $R_{\geq\lambda}$, for *refuel-20* – $P_{\geq\lambda}$. The current implementation of SEIDAR so far does not support feasibility specifications. Therefore, we made PAYNT to stop the synthesis when the required optimum was reached.

Table 5.9 lists the measurements. For *hallway2*, the specified λ was obtained in IDAR and EIDAR within just 80 iterations. Due to this, we managed to achieve record speedups: 10.78 for the overall synthesis time, 15.23 for the MB and 10.57 for the MC. However, since this model has a small family size, in SEIDAR there was an instant switch to AR, which did not lead to any speedup. Even though in *refuel-20* SEIDAR also has to switch to AR (because of „SC/AS“), its speedup is high (more than 6). This happens since EIDAR needs only 56 iterations for its termination, which is insufficient for switching. Therefore, in this case, the results of EIDAR and SEIDAR are similar.

5.5.2 Discussion

Even though SEIDAR accelerates the synthesis of almost every model instance, in theory we expected a better result. Earlier in this paper we stated that for the MDP model checking with the PCTL syntax, the algorithmic complexity is polynomial in its size. Therefore, each

Model	k	λ	AR time (s)			AR it	M.	Iter.	Speedup		
			ov.	MB	MC				ov.	MB	MC
hall	1	0.02149	93.33	11.18	64.5	1464	I	80	10.78	14.87	10.16
							E	80	9.76	15.23	10.57
							S	1464	1.02	1.01	1.03
refuel	1	0.00083	18.54	0.73	1.32	775	I	775	0.82	1.09	1.05
							E	56	6.82	6.99	3.47
							S	56	6.68	6.89	3.47

Table 5.9: Speedups of IDAR, EIDAR and SEIDAR for the feasibility synthesis problem with a threshold given by λ. „AR it“ stands for the number of iterations needed for terminating AR, while „Iter.“ shows the actual number of iterations, run by our methods.

model instance should be accelerated at least as many times as the number of vague/affected states is less than the total number of states. However, as Tables 5.5 and 5.8 show, we could not speed up the overall synthesis time even by a factor of 2. The results obtained in Subsection 5.5.1 for the feasibility synthesis problem can be considered as luck rather than our algorithm's efficiency. Most likely, in the model instances from Table 5.9 STORM accidentally chose different FSCs and significantly affected the further splitting of subfamilies, so we do not take these results as a reference.

The most controversial result was obtained on the *LRV* model. The results from Table 5.6 show that we keep all compatible choices in only 1% of states. Then, in theory, the overall speedup should be large (around 100). However, as Tables 5.5 (EIDAR) and 5.8 (SEIDAR) show, in reality, the acceleration does not even reach factor 2. The reason for this lies in the value of „SC/AS“, which is 8 times higher for EIDAR than for IDAR. It follows that non-affected states are those with a small number of compatible choices. The highest concentration of choices gathered in affected states. Therefore, „% affected“ does not show the real picture of how much we reduced the size of the mask. In addition, the time t_{M} spent by the selected method takes a considerable part of the total synthesis time.

Chapter 6

Final Considerations

6.1 Conclusion

The main goal of this work was to accelerate the abstraction-based synthesis of finite-state controllers for POMDPs using inheritance dependencies of FSCs' families. Primarily, we intended to accelerate model checking, a key computational component of synthesis. On the one hand, the result exceeded our expectations because model building and other optimisations in PAYNT accelerated in addition to model checking. On the other hand, this did not lead to a very significant speedup of the overall synthesis time.

We created the Inheritance Dependencies for the Abstraction-Refinement approach (IDAR) and its extended, more efficient version – EIDAR. One of the goals of the experiments was to find out which of these methods would form the basis of the final product of this thesis – Smart EIDAR. As a result, SEIDAR starts its operation in EIDAR and after a few iterations decides whether to switch to AR or remain in EIDAR. SEIDAR is designed to save the user of PAYNT from choosing which method to use for a particular POMDP model.

All new approaches proved to be consistent with classic AR. The optimal result, regardless of the amount of memory, does not depend on the chosen method. Experiments also show that the impact of inheritance dependencies on the synthesis is ambiguous. We cannot claim that each approach would produce the same optimum for the same number of iterations and vice versa. On average, SEIDAR speeds up the overall synthesis time by 1.2 times, model building by 1.54 times and model checking by 1.61 times. Based on the experiments, it is reasonable for the user to always choose SEIDAR over AR. The worst overall slowdown (0.95) is negligible compared to the best speedup (1.72). We also conducted additional experiments, where the feasibility synthesis problem replaced the max/min synthesis problem. The record speedups were achieved: 10.78 for the overall synthesis time, 15.23 for the model building and 10.57 for the model checking.

6.2 Future Work

This thesis opens up a great horizon for possible future work. We applied the idea of inheritance dependencies only for the inductive synthesis, in particular AR. Probably, a similar approach could also be used in CEGIS or even in the belief-based FSC synthesis method. Also, while writing this paper, we came up with the idea to look at inheritance dependencies from the other side. What if we use this approach between siblings (families

with a common parent)? Unfortunately, it is still unclear what information from one sibling can be used to speed up the synthesis of the other one. Also, in the current version of PAYNT, this is difficult to implement due to the DFS algorithm used in AR.

In some experiments, we observed a change in the required number of iterations to complete the synthesis. We explained this phenomenon by the features of the model checking in Storm. In the future, conducting a more in-depth analysis of the reasons that caused this result would be appropriate. In addition, more experiments could be done and more suitable parameters could be selected for switching in SEIDAR. An updated version of SEIDAR would also support feasibility specifications.

Bibliography

[1] Allen, L. J. *An introduction to stochastic processes with applications to biology.* 2nd ed. Lubbock, Texas, USA: CRC press, 2010. ISBN 978-1-4398-9468-2.

[2] Andriushchenko, R. *Computer-Aided Synthesis of Probabilistic Models.* Brno, CZ, 2020. Master's thesis. Brno University of Technology, Faculty of Information Technology. Available at: https://www.fit.vut.cz/study/thesis/22997/.

[3] Andriushchenko, R.; Bork, A.; Češka, M.; Junges, S.; Katoen, J.-P. et al. Search and Explore: Symbiotic Policy Synthesis in POMDPs. In: Enea, C. and Lal, A., ed. *Computer Aided Verification.* Cham, DE: Springer Verlag, 2023, p. 113–135. LNCS. ISBN 978-3-031-37709-9.

[4] Andriushchenko, R.; Češka, M.; Junges, S. and Katoen, J.-P. Inductive synthesis for probabilistic programs reaches new horizons. In: Groote, J. F. and Larsen, K. G., ed. *International Conference on Tools and Algorithms for the Construction and Analysis of Systems.* Luxembourg City, Luxembourg: Springer, March 2021, p. 191–209. ISBN 978-3-030-72015-5.

[5] Andriushchenko, R.; Češka, M.; Junges, S. and Katoen, J.-P. Inductive Synthesis of Finite-State Controllers for POMDPs. In: *Conference on Uncertainty in Artificial Intelligence.* Eindhoven, NL: Proceedings of Machine Learning Research, 2022, vol. 180, no. 180, p. 85–95. Proceedings of Machine Learning Research. ISSN 2640-3498.

[6] 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 (CAV).* Cham, DE: Springer Verlag, 2021, vol. 12759, p. 856–869. Lecture Notes in Computer Science. ISBN 978-3-030-81684-1.

[7] Baier, C. and Katoen, J.-P. *Principles of model checking.* Cambridge, Massachusetts, London, England: MIT press, 2008. ISBN 978-0-262-02649-9.

[8] Bertsekas, D. and Tsitsiklis, J. N. *Introduction to probability.* 2nd ed. Belmont, Massachusetts: Athena Scientific, 2008. ISBN 978-1-886529-38-0.

[9] Bork, A.; Junges, S.; Katoen, J.-P. and Quatmann, T. Verification of indefinite-horizon POMDPs. In: *International Symposium on Automated Technology for Verification and Analysis.* Hanoi, Vietnam: Springer, October 2020, p. 288–304. ISBN 978-3-030-59151-9.

[10] Bork, A.; Katoen, J.-P. and Quatmann, T. Under-approximating expected total rewards in POMDPs. In: Fisman, D. and Rosu, G., ed. *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Munich, Germany: Springer, March 2022, p. 22–40. ISBN 978-3-030-99526-3.

[11] Cassandra, A. R. POMDP Example Domains. *Tony's POMDP File Repository Page* online. Available at: https://pomdp.org/examples/. [cit. 2024-04-23].

[12] Češka, M.; Hensel, C.; Junges, S. and Katoen, J.-P. Counterexample-guided inductive synthesis for probabilistic systems. *Formal Aspects of Computing*. Springer, 2021, vol. 33, 4-5, p. 637–667. ISSN 0934-5043.

[13] Češka, M.; Jansen, N.; Junges, S. and Katoen, J.-P. Shepherding hordes of Markov chains. In: Vojnar, T. and Zhang, L., ed. *Tools and Algorithms for the Construction and Analysis of Systems: 25th International Conference, TACAS 2019...* Prague, Czech Republic: Springer, April 2019, p. 172–190. ISBN 978-3-030-17464-4.

[14] Chrszon, P.; Dubslaff, C.; Klüppelholz, S. and Baier, C. ProFeat: feature-oriented engineering for family-based probabilistic model checking. *Formal Aspects of Computing*. Springer, 2018, vol. 30, p. 45–75. ISSN 0934-5043.

[15] Clarke, E. M.; Henzinger, T. A.; Veith, H.; Bloem, R. et al. *Handbook of model checking*. Springer, 2018. ISBN 978-3-319-10574-1.

[16] Dehnert, C.; Junges, S.; Katoen, J.-P. and Volk, M. A storm is coming: A modern probabilistic model checker. In: *Computer Aided Verification: 29th International Conference Proceedings, Part II 30*. Heidelberg, Germany: Springer, July 2017, p. 592–600. ISBN 978-3-319-63389-3.

[17] Fosler Lussier, E. Markov models and hidden Markov models: A brief tutorial. *International Computer Science Institute*. Berkeley, California: [b.n.], december 1998.

[18] Grzes, M.; Poupart, P. and Hoey, J. Isomorph-free branch and bound search for finite state controllers. *IJCAI*. Beijing, China: AAAI Press, august 2013, p. 2282–2290.

[19] Hansen, E. A. Solving POMDPs by searching in policy space. *UAI'98: Proceedings of the Fourteenth conference on Uncertainty in artificial intelligence*. University of Massachusetts Amherst: Morgan Kaufmann Publishers Inc., july 1998, p. 211–219.

[20] Hansen, E. A. and Zhou, R. Synthesis of hierarchical finite-state controllers for POMDPs. *ICAPS*. Mississippi State University, USA: AAAI Press, april 2003, p. 113–122.

[21] Junges, J.; Jansen, N.; Wimmer, R.; Quatmann, T.; Winterer, L. et al. Finite-state controllers of POMDPs via parameter synthesis. *UAI 2018: Proceedings of the Thirty-Fourth Conference on Uncertainty in Artificial Intelligence*. Monterey, California, USA: Corvallis: AUAI Press, 2018, p. 519–529.

[22] Kochenderfer, M. J. *Decision making under uncertainty: theory and application*. Cambridge, Massachusetts, London, England: MIT press, 2015. ISBN 978-0-262-02925-4.

[23] Kurniawati, H. Partially observable markov decision processes (pomdps) and robotics. *ArXiv preprint arXiv:2107.07599*. Australian National University, Canberra, Australia: [b.n.], july 2021.

[24] Kwiatkowska, M.; Norman, G. and Parker, D. Stochastic model checking. *Formal Methods for Performance Evaluation: 7th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2007, Bertinoro, Italy, May 28-June 2, 2007, Advanced Lectures 7*. Berlin, Heidelberg: Springer, 2007, p. 220–270.

[25] Mamon, R. S. and Elliott, R. J. *Hidden Markov models in finance: Further Developments and Applications, Volume II*. Adelaide, Australia, London, Canada: Springer, may 2014. ISBN 978-1-4899-7441-9.

[26] Pajarinen, J. and Peltonen, J. Periodic finite state controllers for efficient POMDP and DEC-POMDP planning. *Advances in neural information processing systems*. Aalto, Finland: NIPS, 2011, vol. 24.

[27] Poupart, P. *Exploiting structure to efficiently solve large scale partially observable Markov decision processes*. University of Toronto, Canada: Citeseer, 2005. ISBN 978-0-494-02727-1.

[28] Puggelli, A.; Li, W.; Sangiovanni Vincentelli, A. L. and Seshia, S. A. Polynomial-time verification of PCTL properties of MDPs with convex uncertainties. In: *Computer Aided Verification: 25th International Conference, CAV 2013. Proceedings 25*. Saint Petersburg, Russia: Springer, July 2013, p. 527–542. ISBN 978-3-642-39798-1.

[29] Smallwood, R. D. and Sondik, E. J. The optimal control of partially observable Markov processes over a finite horizon. *Operations research*. Stanford University, Stanford, California: INFORMS, october 1973, vol. 21, no. 5, p. 1071–1088.

[30] Spaan, M. T. Partially observable Markov decision processes. In: Wiering, M. and Otterlo, M. V., ed. *Reinforcement learning: State-of-the-art*. Berlin: Springer, 2012, p. 387–414. ISBN 978-3-642-27644-6.

Appendix A

Contents of the included storage media

- `docs/` – the text report and its sources:
  - `xshevc01-POMDP.pdf` – this text report.
  - `xshevc01-POMDP-print.pdf` – the print version of the text report.
  - `src/` – the source form of the text report.
- `src/` – the source codes of programs, including PAYNT, installation script, POMDP models and C++ binding.
- `README.md` – contains installation instructions and user manual for running PAYNT.

Appendix B

Manual

Installation

Folder `src/` contains the extended version of PAYNT, which also requires Storm[1] and Stormpy[2]. If you do not have them installed, use the installation script `install.sh` to install Storm, Stormpy and other required dependencies. Complete compilation might take up to an hour. The Python environment will be available in `prerequisistes/venv`.

Running PAYNT

main default flags:

`--project PROJECT`	The path to the benchmark folder [required].
`--pomdp-memory-size INTEGER`	Implicit memory size for POMDP FSCs [default: 1].
`--profiling`	Run profiling.

new flags:

`--use-inheritance`	Use inheritance dependencies (IDAR).
`--use-inheritance-extended`	Use extended inheritance dependencies (EIDAR).
`--use-smart-inheritance`	Use smart inheritance dependencies (SEIDAR).
`--iterations INTEGER`	Limit the number of iterations for the synthesis of FSCs (experimental purposes).

Running selected model instances

If you want to run e.g. a *4x3-95* model with $k = 2$ on AR, use

```
python3 paynt.py --project models/archive/cav23-saynt/4x3-95/
--pomdp-memory-size 2 --profiling
```

To run e.g. *LRV* with a limited number of iterations, write

```
python3 paynt.py --project models/archive/cav23-saynt/lrv/
--pomdp-memory-size 3 --profiling --iterations 40
```

To use another method, use the corresponding flag.

[1] Available at: https://github.com/moves-rwth/storm
[2] Available at: https://github.com/moves-rwth/stormpy