RWTH AACHEN UNIVERSITY

MASTER THESIS

Human-readable Scheduler Representation for Markov Decision Processes

---

by
Lea HIENDL

Supervisor:
Tim Quatmann

1st Examiner:
Prof. Dr. Ir. Dr. h. c. Joost-Pieter Katoen

2nd Examiner:
Prof. Dr. Erika Ábrahám

The present work was submitted to

Chair for Software Modeling and Verification
RWTH Aachen University

May 9, 2018

Zentrales Prüfungsamt/Central Examination Office

RWTH AACHEN UNIVERSITY

Eidesstattliche Versicherung
Statutory Declaration in Lieu of an Oath

___________________________ ___________________________

Name, Vorname/Last Name, First Name

Matrikelnummer (freiwillige Angabe)
Matriculation No. (optional)

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Arbeit/Bachelorarbeit/Masterarbeit* mit dem Titel

I hereby declare in lieu of an oath that I have completed the present paper/Bachelor thesis/Master thesis* entitled

____

____

____

selbstständig und ohne unzulässige fremde Hilfe (insbes. akademisches Ghostwriting) erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

independently and without illegitimate assistance from third parties (such as academic ghostwriters). I have used no other than the specified sources and aids. In case that the thesis is additionally submitted in an electronic format, I declare that the written and electronic versions are fully identical. The thesis has not been submitted to any examination body in this, or similar, form.

___________________________ ___________________________

Ort, Datum/City, Date

Unterschrift/Signature

*Nichtzutreffendes bitte streichen

*Please delete as appropriate

Belehrung:
Official Notification:

§ 156 StGB: Falsche Versicherung an Eides Statt

Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

Para. 156 StGB (German Criminal Code): False Statutory Declarations

Whoever before a public authority competent to administer statutory declarations falsely makes such a declaration or falsely testifies while referring to such a declaration shall be liable to imprisonment not exceeding three years or a fine.

§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt

(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.

(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.

Para. 161 StGB (German Criminal Code): False Statutory Declarations Due to Negligence

(1) If a person commits one of the offences listed in sections 154 through 156 negligently the penalty shall be imprisonment not exceeding one year or a fine.

(2) The offender shall be exempt from liability if he or she corrects their false testimony in time. The provisions of section 158 (2) and (3) shall apply accordingly.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:

I have read and understood the above official notification:

___________________________ ___________________________

Ort, Datum/City, Date

Unterschrift/Signature

RWTH AACHEN UNIVERSITY

Abstract

Chair for Software Modeling and Verification
Computer Science Department

Master of Science

Human-readable Scheduler Representation for Markov Decision Processes

by Lea HIENDL

Markov decision processes model problems of decision-making under uncertainty. A lot of work has been devoted to solving them for optimal schedulers that maximize the probability of reaching an objective. Such schedulers represent decision making strategies and diagnostic information, but their representation lacks in explanatory qualities and conciseness, making them hard for humans to comprehend. Alternative representations are desired to facilitate readability.
This research thesis concerns itself with human-readable scheduler representations, which are both compact and explanatory, starting with a recently proposed approximative representation as decision trees. This approach has the advantage of representing only the most important decisions of a strategy, resulting in a compact tree-like structure that is intuitive to understand. The importance of decisions is computed via a heuristic and approximated by experimental simulation, an approach that yields different results upon each execution. We propose an alternative deterministic means of computing an importance function via linear programs and show empirically that the resulting trees are roughly of the same size.
From the decision tree representation we derive a scheduler representation in the PRISM modelling language, which has the advantage of being familiar to many users of model checkers, and show that it can be used to correctly schedule a PRISM program to produce a Markov chain.

Acknowledgements

I want to thank my thesis advisor, Tim Quatmann, who proposed the topic to me and provided me with invaluable advice, feedback and encouragement over the course of the project.

I also want to thank my parents, whose constant support enabled me to pursue a degree.

Contents

Abstract v

Acknowledgements vii

1 Introduction 1
- 1.1 Background and Motivation . . . 1
- 1.2 Related Work . . . 2
- 1.3 Thesis Contribution . . . 2
- 1.4 Outline . . . 5

2 Preliminaries 7
- 2.1 Notation . . . 7
- 2.2 Probability Theory . . . 7
- 2.3 Probabilistic Model Checking . . . 8
  - 2.3.1 Models . . . 8
  - 2.3.2 Schedulers . . . 9
  - 2.3.3 Probability Measure of Markov Chains . . . 10
  - 2.3.4 Reachability Properties . . . 10

3 Scheduler Representation as Decision Trees 13
- 3.1 Introduction . . . 13
- 3.2 Original Algorithm . . . 13
  - 3.2.1 Decision Tree Learning . . . 13
    - Decision Tree . . . 13
    - Training . . . 14
  - 3.2.2 Scheduler Learning . . . 15
    - Training Data . . . 15
    - Decision Tree Scheduler . . . 16
  - 3.2.3 Approximative Representation and Error Computation . . . 16
- 3.3 Adaptations . . . 18
  - 3.3.1 Action Trees . . . 18
  - 3.3.2 State Importance with Linear Programs . . . 19
    - LP Formulation on MDPs . . . 20
    - LP Formulation on MCs . . . 21

4 Scheduler Representation in the PRISM Language 23
- 4.1 Introduction . . . 23
- 4.2 PRISM Language . . . 23
  - 4.2.1 Syntax . . . 23
  - 4.2.2 Semantics . . . 24
    - MDP Probabilistic Transition Function . . . 25
    - MC Probabilistic Transition Function . . . 25
  - 4.2.3 Simple PRISM Program . . . 27

4.3 PRISM Scheduler Representation . . . 29
4.3.1 Scheduler Module . . . 30
4.3.2 Deadlocks . . . 31
4.4 Correctness . . . 33
4.5 Alternative Error Computation . . . 35

5 Experiments **37**
5.1 Overview . . . 37
5.2 Implementation . . . 37
Simulation . . . 37
Pruning Paramters . . . 37
Linear Programs . . . 38
5.3 Results . . . 39
5.3.1 Benchmarks . . . 39
5.3.2 Error And Tree Size . . . 40
5.3.3 Runtime . . . 41
Runtime for Error Computation . . . 42

6 Conclusion **45**
6.1 Discussion . . . 45
6.2 Outlook . . . 46

Bibliography **47**

Chapter 1

Introduction

1.1 Background and Motivation

Model checking is an automated verification technique that describes system behaviour as state models and performs an exhaustive exploration of the resulting state space. This way, it can be rigorously guaranteed that a system satisfies a given property. [3]
Many systems incorporate an element of uncertainty, either through unpredictable behaviour, such as failure rates and loss of communication data, or by design, in the case of randomised algorithms. Therefore, the model checking approach has been extended to verification of probabilistic systems, which find a wide variety of application in security and network protocols or systems biology, to name a few, and the modelling of stochastic processes in general. The goal of probabilistic model checking is to provide a guarantee that a system satisfies a property with a given probability bound, e.g. that an error state is only reached with a sufficiently small probability.
The standard models to represent probabilistic systems are Markov models, transition systems whose behaviour in a state is determined by a probability distribution. *Markov decision processes* are one variant which allow a non-deterministic choice between distributions in a state. Non-determinism can be used to e.g. model that behaviour in a state is underspecified and left up to to an (abstract) decision maker who has the choice between several actions.
Non-determinism arises naturally from e.g. user input or concurrency of processes, but can also be used to express decision-making problems explicitly, where the best choice in each state is unknown, leaving it up to the decision maker to find a strategy that maximizes the probability of a desired outcome, subject to additional constraints like minimal resource consumption and step or time bounds.
Many of these problems can be formulated as reachability problems on the state graph of Markov decision processes. For this class of properties, model checkers already incorporate methods for "solving Markov decision processes", i.e. generating optimal *schedulers*, which map states to choices and thus describe a decision strategy. Many AI planning problems are modelled as Markov decision processes [17] with the goal of finding such an optimal strategy, which also details *how* an objective can be reached. For the same reason, schedulers are valuable debugging tools because they identify how error states are reached within a program. In both of these cases, comprehension of the strategy by humans plays an important role.
However, the explicit representation of schedulers consists of lists of state-action pairs, which scale with model size, and which are in general too large to be readily understood. It therefore seems expedient to find alternative, *readable representations* of schedulers. The goal of this thesis is to investigate such representations.

1.2 Related Work

The explicit representation of models and schedulers enumerates all states and actions. In response to the state explosion problem in model checking, it has become popular to use symbolic representation as binary decision diagrams (BDDs) for state sets and transition functions [18, 8, 14], as well as schedulers [24, 5]. BDDs are tree structures representing boolean functions, and as such are more compact than explicit representations, but not necessarily easy to read. However, while the generation of optimal strategies has been extensively studied, compact representation structures have received less attention.

A related topic to scheduler representation is that of finding small counterexamples, which are similar to schedulers in that they provide diagnostic information as a description of how a given property is *not* satisfied by a model. In probabilistic model checking, they usually take the form of sets of paths in Markov chains whose cumulative probability exceeds the given bound. The representation of these path sets as AND/OR trees has been proposed by [2] and was also used by [16] to generate fault trees for counterexamples which explain the causality relationships leading to a system failure. An alternative notion of counterexamples as minimal critical subsystems has been used by e.g [23] who use SMT/MILP solvers to compute such subsystems. [9] avoid an explicit representation by computing a smallest set of guarded commands for PRISM programs [21] which induce a minimal critical subsystem. [1] gives a more exhaustive overview of methods for counterexample generation and representation.

More recently, it has been proposed to use decision trees to represent schedulers [5]. Decision trees can be trained to memorize only the important features of a strategy, and need not represent the unimportant bits. The algorithms for their construction are based on information gain, therefore yielding more compact representations than BDDs, whose size depends on finding an optimal variable ordering, a problem which is known to be hard.
Decision trees were also used to represent strategies for two player games on graphs [6] with a modified training algorithm that produces an error-free representation. For the readability of schedulers however, an approximative representation is often sufficient and even advantageous because of its decreased complexity.

1.3 Thesis Contribution

The goal of this thesis is to investigate scheduler representations which are easily understandable by humans and algorithms to compute them effectively. Therefore we are interested in what qualities make a scheduler readable. The first criterion that volunteers itself is that it should be *compact*, i.e. the amount of presented information should not be too large, which is the primary problem with explicit representations. However this does not always suffice, as in the case of BDDs, who are much smaller than explicit representations, but still hard to understand. Therefore, the representation should also be *explanatory*, i.e. it should convey an intuitive notion of the most important decisions made by the strategy, and ignore the unimportant details.
As an approach that promises to improve on both of these qualities over other representations, we are using the decision tree representation of schedulers [5] as the starting point for this thesis. To demonstrate how it helps to gain insight into a strategy, consider the following scenario illustrated by Figure 1.1. A mountain climber is scaling a mountain which is represented symbolically as a 3x3 grid. The climber

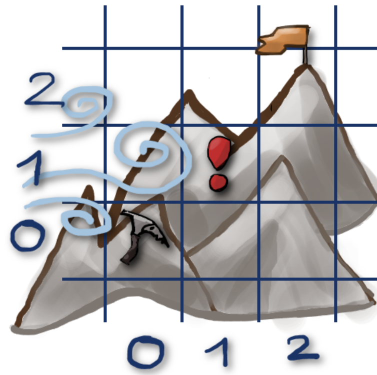


FIGURE 1.1: Illustration of the climbing route problem.

starts at position (0,0) and his or her goal is to reach the mountain top at position (2,2). To this end, the climber can move one tile at a time (up, down, left or right). Position (1,1) is a dangerous location, which will cause the climber to fall and should be avoided. The task is complicated by a strong eastward wind: Every time the climber tries to move upward, there is a chance that he or she has to move to the right instead. To maximize the chance of reaching the mountain top, his or her strategy should be to first climb to the right, and then move upwards. This way he or she avoids the danger of being blown onto the dangerous position by the wind.
This strategy is represented by the decision tree in Figure 1.2 with x and y representing the position of the climber on the grid. It is interpreted as follows. Starting with the root node, every inner node poses a question either about the position of the climber, or about an action he can take. If the question is answered with yes, proceed to the left child (indicated by a straight line), and if it is answered with no, proceed with the right child (indicated by a dotted line). At the end of this process, a leaf node is reached which is labelled either with "good", if the particular combination of position and action is considered good by the strategy, or labelled with "bad" otherwise. For the tree in Figure 1.2, we have that the action "right" is considered good in any position, the action "up" is good only in a position with x-coordinate equal to 2, and both "left" and "down" are not good in any position. This represents the strategy we had in mind: The climber moves to the right until he or she can no longer do so because the rightmost edge of the grid has been reached ($x = 2$), and then moves upward to the target position.

Note that technically, the tree also gives an (unoptimal) decision for positions on the mountain that do not lie on the optimal climbing route, i.e. according to the tree, the climber should move right in position (0,1). However, for an optimal strategy to reach the mountain top from (0,0), decision in these states are irrelevant since they are never reached. The algorithm that computes the decision tree representation takes this into account by not learning decisions in these *unimportant* states. By contrast, decisions in *important* states are stressed. The algorithm uses simulation to compute an importance function, i.e. it lets the climber scale the mountain a set number of times according to the strategy, and records how often a position was visited and what actions were taken in it. A decided disadvantage of this approach is that every execution could yield a different tree, and depending on model and strategy, potentially has to be run many times to produce the best outcome.
Therefore we propose a different means of computing an importance function via

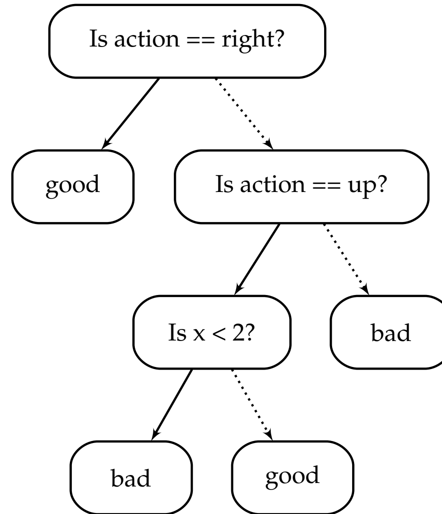


FIGURE 1.2: Decision tree representation of an optimal scheduler for the climbing route problem.

linear programs. In particular, we formulate a maximum flow problem over the state graph of the model. This is similar to simulation in that we can envision it as sending a number of climbers up the mountain, but we rely on a linear optimizer to maximize the amount of climbers that reach the mountain top, and take the importance of a position to be the number of climbers that have visited it. This gives us both an optimal scheduler and an importance function to train a decision tree representation. Also, we can use a simplified version of the linear program to compute an importance function for any given scheduler.
An empirical evaluation of the approaches shows that the resulting trees are comparable in size, and that no method is clearly superior with respect to tree size. However, simulation tends to be slower for larger models.

Departing from decision trees as representation medium, we can also consider other factors that could improve readability of schedulers. Usually, humans process information faster that is presented in a familiar format. Therefore we could add a third entry to the list of properties desirable for scheduler representations: familiarity with the representation language.
We therefore propose to represent schedulers in a language used to describe input models for model checkers. A popular language for symbolic descriptions of Markov models is the PRISM language [21] which is supported by state of the art model checkers [14, 8]. A PRISM program is made up of modules which synchronize over common commands. We represent a strategy as an additional scheduler module which schedules the rest of the modules.
For our construction, we use a decision tree representation as starting point. Decision trees lend themselves well for finding succinct PRISM representations, because they are already compact and branch over predicates, which can be used to compute the conditions under which a command (action) should be enabled. For example, for the climbing route problem, we can use the tree representation in Figure 1.2 to compute the PRISM representation in Listing 1.1. The guards of the commands right, up, left and down exactly reflect when the action is good according to the strategy: right is always good (true), left and down are never good (false) and up is good if x is not smaller than 2, i.e. $x = 2$.

LISTING 1.1: PRISM scheduler representation of an optimal strategy for the climbing route problem.

```
1
2
3 module scheduler
4         [right] true -> true;
5         [up]    !(x < 2)-> true;
6         [left]  false -> true;
7         [down]  false -> true;
8 endmodule
```

As we will show, the representation in Listing 1.1 does not completely suffice. In particular, it relies on the input program being of a particular form, and does not take into account the possibility of deadlocks being introduced by the scheduling. We solve these problems in Chapter 4 and furthermore show that our construction is correct: The program scheduled by the scheduler module produces a model which is equivalent to the model scheduled according to the tree representation of the scheduler.

In summary, the contributions of this thesis are

- implementation and evaluation of the decision tree representation algorithm as proposed by [5],
- improvement of the algorithm through an alternative means of computing the importance of states,
- a scheduler representation in the PRISM language,
- a proof of correctness of its construction,
- and experimental evaluation and comparison of all implemented algorithms.

1.4 Outline

In Chapter 2 we first give a brief overview of concepts from probabilistic model checking which are helpful in understanding the content of the thesis.
Chapter 3 starts with detailing the algorithm to compute a decision tree representation of a scheduler as in [5]. Modifications, such as our alternative method of computing importance functions via linear programs, are discussed afterwards. Chapter 4 details our construction of the PRISM scheduler representation and the proof of its correctness.
All versions of the algorithm we describe have been implemented by us and tested on several benchmarks. Implementation details and results can be found in Chapter 5.
We end with a conclusion and outlook in Chapter 6.

Chapter 2

Preliminaries

2.1 Notation

We represent the set of natural numbers including zero, the set of integers and the set of rational numbers as $\mathbb{N}$, $\mathbb{Z}$ and $\mathbb{Q}$ respectively. For a finite set S, we denote its power set by 2^S and its cardinality by $|S|$.
A variable is an expression x that takes values from a fix domain $\mathcal{D}(x) \subseteq \mathbb{R}$. For a set $\mathcal{V}$ of variables, a variable valuation is a function $v : \mathcal{V} \to \mathcal{D}(\mathcal{V})$ that assigns to a variable x a value in its domain, i.e. $v(x) \in \mathcal{D}(x)$.
A predicate over a variable x is an expression $[x \sim c]$ with $c \in \mathcal{D}(x)$ and $\sim \in \{\leq, <, \geq, >, =\}$. A variable valuation for x satisfies the predicate $v \models [x \sim c]$ if $[v(x) \sim c]$ evaluates to true. The satisfaction relation $\models$ is distributive over conjunction and disjunction of predicates: If $p_0, \ldots, p_n$ are predicates over variables in $\mathcal{V}$ with valuation v, then $v \models (p_0 \wedge \ldots \wedge p_n)$ iff $v \models p_i$ for all $i \in \{0, \ldots, n\}$ and $v \models (p_0 \vee \ldots \vee p_n)$ iff $v \models p_i$ for some $i \in \{0, \ldots, n\}$. For negation it holds that $v \models \neg p$ iff $v \not\models p$.

2.2 Probability Theory

A random experiment can be modelled via a probability space.

Definition 2.2.1. *A **probability space** is a three-tuple $(\Omega, \mathcal{F}, Pr)$. Ω is a nonempty set that is called the sample space and represents all possible outcomes in the experiment. $\mathcal{F} \subseteq 2^\Omega$ is the event space, which defines a σ-algebra on Ω, i.e.*

- $\emptyset \in \mathcal{F}$.
- *$\mathcal{F}$ is closed under complement: if $A \in \mathcal{F}$ then also $(\Omega \setminus A) \in \mathcal{F}$.*
- *$\mathcal{F}$ is closed under countable unions: if $A_i \in \mathcal{F}$ for $i = 1, 2, \ldots$, then $(\cup_{i=1}^{\infty} A_i) \in \mathcal{F}$.*

The probability measure Pr is a function $Pr : \mathcal{F} \to [0,1]$ s.t. $Pr(\Omega) = 1$ and for all disjoint events $E, F \in \mathcal{F}, E \cap F = \emptyset : Pr(E \cup F) = Pr(E) + Pr(F)$.

If Ω is a countable set and the event space is the full power set $\mathcal{F} = 2^\Omega$, then a probability measure can always be obtained by a function $\mu : \Omega \to [0,1]$ s.t. $\sum_{a \in \Omega} \mu(a) = 1$. The probability measure Pr is defined as $Pr(A) = \sum_{a \in A} \mu(a)$ for all sets $A \in \mathcal{F}$.

Example 2.2.1. *A fair coin is tossed two times. There are 4 possible outcomes: $\Omega = \{HH, HT, TH, TT\}$ where e.g. HT stands for the event that the first toss lands heads and the second lands tails. We define a function $\mu : \Omega \to [0,1]$ s.t. $\mu(HH) = \mu(HT) = \mu(TH) = \mu(TT) = \frac{1}{4}$, since every outcome is equally likely. If $\mathcal{F} = 2^\Omega$ then for example $Pr(\{HH, HT\}) = \mu(HH) + \mu(HT) = \frac{1}{2}$.*

Definition 2.2.2. *A* ***probability distribution*** *over a countable set S is a function $\mu : S \to [0,1]$ such that*

$$\sum_{s \in S} \mu(s) = 1.$$

The ***support*** *of μ is the set $Supp(\mu) = \{s \in S \mid \mu(s) > 0\}$.*
A probability distribution $\mu \ldots$

- *is called a* ***Dirac*** *distribution if $Supp(\mu) = 1$.*
- *is* ***uniform*** *if it holds that*

$$\mu(s) = \frac{1}{|S|} \quad \forall s \in S.$$

We use $Dist(S)$ for the set of all distributions over S and use $\mu_{uni}(S)$ to denote the uniform distribution over S.

2.3 Probabilistic Model Checking

This section gives a brief overview over central topics of probabilistic model checking, more detailed explanations of which can be found in [3, Chapter 10].

2.3.1 Models

For checking probabilistic systems, some of the most popular models are Markov models, which exhibit the Markov property (also: memorylessness); the next state only depends on the current state and the probability distribution associated with it. The simplest of the Markov models is the Markov chain, which has only probabilistic transitions.

Definition 2.3.1. *A* ***(discrete-time) Markov chain*** *(DT)MC is a tuple $M = (S, P, \bar{s})$ where S is a finite set of states, $\bar{s} \in S$ is the initial state, and $P : S \times S \to [0,1]$ is a probabilistic transition function such that for all states s:*

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

P defines a probability distribution $\mu_s : S \to [0,1]$ for every $s \in S$: $\mu_s(s') = P(s, s')$.

The possible behaviours of a Markov chain are formalised as paths.

Definition 2.3.2. *A* ***path*** *of a Markov chain M is an (infinite) sequence $\pi = s_0 s_1 \ldots$ of states such that $s_0 = \bar{s}$ is the initial state and $s_{i+1} \in Supp(\mu_{s_i})$ for all $i \in \mathbb{N}$.*

Markov chains suffice for purely probabilistic systems, but cannot capture non-deterministic behaviour, for example the choice which of two concurrent processes performs the next action. Non-determinism is also the natural way to model user input, or any decision whose outcome cannot be adequately described by a stochastic distribution.
An extension of Markov chains that also incorporates non-determinism is given by Markov decision processes, which allow a choice between multiple actions in each state, each associated with its own probability distribution.

Definition 2.3.3. *A* ***Markov decision process*** *(MDP) is a tuple* $\boldsymbol{\mathcal{M}} = (S, A, Act, P, \bar{s})$ *where S is a finite set of states, A is a finite set of actions,* $Act : S \to 2^A$ *assigns each state a set of actions enabled in that state,* $\bar{s} \in S$ *is the initial state, and* $P : S \times A \times S \to [0,1]$ *is the probabilistic transition function such that for all states* $s \in S$ *and actions* $a \in A$*:*

$$\sum_{s' \in S} P(s, a, s') = \begin{cases} 1 & \text{if } a \in Act(s) \\ 0 & \text{if } a \notin Act(s) \end{cases}.$$

P defines a probability distribution $\mu_{s,a} : S \to [0,1]$ *for every state-action pair* (s, a) *with* $a \in Act(s)$*:* $\mu_{s,a}(s') = P(s, a, s')$*.*

An MC can be viewed as a special case of an MDP with $|Act(s)| = 1$ for all $s \in S$. For MDPs, possible executions are also defined as paths, denoted by alternating sequences of states and actions.

Definition 2.3.4. *A* ***path*** *of an MDP* $\mathcal{M}$ *is an (infinite) sequence* $\pi = s_0 a_0 s_1 a_1 \ldots$ *of states and actions such that* $s_0 = \bar{s}$ *is the initial state,* $a_i \in Act(s_i)$ *and* $s_{i+1} \in Supp(\mu_{s_i, a_i})$ *for all* $i \in \mathbb{N}$*.*

Let M be an MC or an MDP. $Paths(M)$ is the set of all paths of M and for a given path $\pi \in Paths(M)$, let $pref(\pi)$ be the set of all finite prefixes of π that end with a state. Then we define the set of all finite paths for M as:

$$Paths_{fin}(M) = \bigcup_{\pi \in Paths(M)} pref(\pi)$$

The combined probability of path sets is computed to verify certain properties (Section 2.3.4). For MDPs this requires resolving non-determinism by reasoning over so called schedulers.

2.3.2 Schedulers

A scheduler (also: strategy, policy) for an MDP replaces the non-deterministic choice in every state by a probabilistic one.

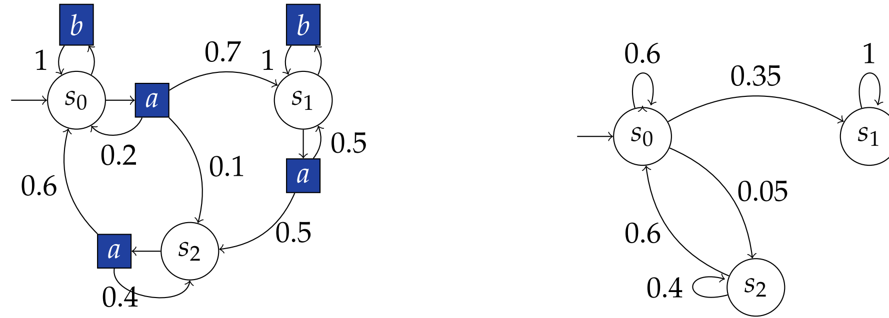
Definition 2.3.5. *Let* $\mathcal{M} = (S, A, Act, P, \bar{s})$ *be a Markov decision process. A* ***scheduler*** *is a function* $\sigma : S \to Dist(Act)$ *that assigns to every state a distribution over the actions enabled in that state, i.e.* $\sigma(s) \in Dist(Act(s))\ \forall s \in S$*.*
A scheduler is ***deterministic*** *if it assigns to every argument a Dirac distribution. The notation then simplifies to* $\sigma : S \to Act$*.*

If a scheduler is applied to a Markov decision process, the result is a Markov chain.

Definition 2.3.6. *Let* $\mathcal{M} = (S, A, Act, P, \bar{s})$ *be a Markov decision process and* $\sigma : S \to Dist(Act)$ *a scheduler. The* ***induced*** *Markov chain is* $\mathcal{M}_\sigma = (S, P_\sigma, \bar{s})$ *where*

$$P_\sigma(s, s') = \sum_{a \in Act(s)} P(s, a, s') \cdot \sigma(s)(a).$$

Example 2.3.1. *Figure 2.1a shows a graph representation of a Markov decision process* $\mathcal{M}$ *with state set* $S = \{s_0, s_1, s_2\}$*, initial state* $\bar{s} = s_0$ *and actions* $A = \{a, b\}$*. The sets of enabled actions are* $Act(s_0) = \{a, b\}$*,* $Act(s_1) = \{a, b\}$*,* $Act(s_2) = \{a\}$*. The transition*



(A) An example MDP. States are represented as circles and actions as rectangles.

(B) An example MC which is produced by scheduling the MDP on the left.

FIGURE 2.1

function is depicted as arrows from states to actions and actions to states.
One possible scheduler $\sigma : S \to Dist(A)$ for $\mathcal{M}$ is:

$$\sigma(s_0) = \{a \mapsto 0.5, b \mapsto 0.5\}, \sigma(s_1) = \{a \mapsto 0, b \mapsto 1\}, \sigma(s_2) = \{a \mapsto 1\}.$$

It induces the Markov chain $\mathcal{M}_\sigma$ which is depicted in Figure 2.1b.

2.3.3 Probability Measure of Markov Chains

To reason formally over the likelihood of model behaviours, we need to define a probability measure over sets of paths of Markov chains. As the set of possible behaviours of a Markov chain M, $Paths(s)$ serves as the sample space in the corresponding probability space (Definition 2.2.1). The σ-algebra is defined via the *cylinder sets* of finite prefixes of paths.

Definition 2.3.7. *The **cylinder set** of $\hat{\pi} = s_0 \dots s_n \in Paths_{fin}(M)$ is*

$$Cyl(\hat{\pi}) = \{\pi \in Paths(M) \mid \hat{\pi} \in pref(\pi)\}.$$

The cylinder set of a finite path $\hat{\pi}$ consists of all infinite paths which start with $\hat{\pi}$. We define the σ-algebra $\mathcal{F}^M$ to be the smallest σ-algebra which contains all cylinder sets.
The probability measure over the events $Cyl(\hat{\pi})$ is then given as

$$Pr^M(Cyl(s_0 s_1 \dots s_n)) = P(s_0 \dots s_n) := \begin{cases} 1 & \text{if } n = 0 \\ \Pi_{0 \le i \le n} P(s_i, s_{i+1}) & \text{else.} \end{cases}$$

Intuitively, we can now specify the probability of model behaviours by describing them with paths, which define an event (cylinder sets).

2.3.4 Reachability Properties

In probabilistic model checking, it is often of particular interest to compute the probability of reaching a particular set of states.
Given a Markov chain $M = (S, P, \bar{s})$ and such a target set $T \subseteq S$, the set of paths containing a target state is

$$\Diamond T = \{\pi \mid \pi = s_0 s_1 \dots \in Paths(M) \text{ and } \exists s_i \in T\} \subseteq Paths(M).$$

To compute the probability of $\Diamond T$ we need to characterize it by a cylinder set of paths. In particular, we want the union of cylinder sets spanned by finite paths ending in T:

$$Paths_{fin,\Diamond T}(M) = \{s_0 s_1 \ldots s_n \in Paths_{fin}(M) \mid s_0 s_1 \ldots s_{n-1} \notin T \text{ and } s_n \in T\}.$$

For two distinct prefixes $\hat{\pi}_1, \hat{\pi}_2 \in Paths_{fin,\Diamond T}(M)$, the cylinder sets are disjoint $Cyl(\hat{\pi}_1) \cap Cyl(\hat{\pi}_2) = \emptyset$. Therefore we can compute the probability of the union of cylinder sets as

$$\begin{aligned} Pr^M(\Diamond T) &= \sum_{s_0 \ldots s_n \in Paths_{fin,\Diamond T}(M)} Pr^M(Cyl(s_0 \ldots s_n)) \\ &= \sum_{s_0 \ldots s_n \in Paths_{fin,\Diamond T}(M)} P(s_0 \ldots s_n). \end{aligned}$$

If we want to compute reachability probabilities for an MDP $\mathcal{M}$, we need to resolve non-determinism by reasoning over schedulers. In particular, we are looking for the maximum reachability probability over all possible schedulers:

$$Pr^{\mathcal{M}}_{\max}(\Diamond T) = \max_{\sigma} Pr^{\mathcal{M}_\sigma}(\Diamond T).$$

We also write $Pr^{\sigma}(\Diamond T)$ as the probability of reaching T in the induced Markov chain M_σ if it is clear from the context. Notably, it suffices for σ to range over all deterministic schedulers [4].

One method to compute $Pr_{\max}(\Diamond T)$ and a corresponding deterministic scheduler is *value iteration*, based on backwards reachability analysis of the state graph. Let $\mathcal{M}_s$ be the MDP which is obtained from $\mathcal{M}$ by making s its only initial state. Then let $x_s := Pr^{\mathcal{M}_s}_{\max}(\Diamond T)$ be the maximal probability to reach T from state s. For all $s \in S$ the value x_s is approximated iteratively $x_s^{(0)}, x_s^{(1)}, \ldots$ such that

$$\lim_{n \to \infty} x_s^{(n)} = x_s.$$

States t in the target set are initialized to $x_t^{(0)} := 1$. For all other states $s \in S \setminus T$: $x_s^{(0)} := 0$. The algorithm updates the values for all states successively;

$$x_s^{(n+1)} = \max_{a \in Act(s)} \sum_{s' \in S} P(s, a, s') \cdot x_{s'}^{(n)}$$

and terminates when

$$x_s^n - x_s^{n-1} < \varepsilon$$

for a given precision parameter ε. Optionally, the algorithm computes a sequence of schedulers $\sigma^0, \sigma^1, \ldots$. In every iteration, the set

$$\Sigma_s^{(n)} = \arg \max_{a \in Act(s)} \sum_{s' \in S} P(s, a, s') \cdot x_{s'}^{(n)}.$$

contains the schedulers which maximize the probability to reach a target state from state s. In the $(n+1)^{th}$ iteration of the algorithm, we update the scheduler only if

the previous scheduler is not still the best (i.e. in set $\Sigma_s^{(n+1)}$):

$$\sigma^{(n+1)}(s) = \begin{cases} \sigma^{(n)}(s) & \text{if } \sigma^{(n)}(s) \in \Sigma_s^{(n+1)} \\ \sigma' & \text{for } \sigma' \in \Sigma_s^{(n+1)} \text{ else.} \end{cases}$$

For every iteration of the algorithm it holds that

$$x_s^{(n)} \leq Pr^{\mathcal{M}_{\sigma^{(n)},s}}(\diamond T) \leq Pr_{\max}^{\mathcal{M}}(\diamond T) = x_s.$$

The precision parameter ε controls the gap between $x_s^{(n)}$ and x_s after termination.

Chapter 3

Scheduler Representation as Decision Trees

3.1 Introduction

Schedulers describe strategies for fulfilling (reachability) objectives, and therefore are of interest in many domains, such as AI planning and debugging, where it is also desirable that a human programmer gains an understanding of the strategy. The explicit representation of schedulers $\sigma : S \to Act$ are lists of state-action pairs that scale in size with the typically large state spaces of probabilistic models. The task of understanding schedulers therefore amounts to a search for meaning in a large amount of data. This is the stated goal of machine learning, a field of computer science concerned with extracting information from large data sets and representing it in an understandable format.
A widely used tool in machine learning is decision tree learning, which trains a model (a decision tree) to predict the value of a target variable based on several inputs. While used in machine learning predominantly as a predictive model, its visualization resembles a flowchart and makes it a very readable representation of the underlying decision process, and decision strategies in general.

This is the primary idea that Brázdil et al. capitalize on in their work on *Counterexample Explanation by Learning Small Strategies in Markov Decision Processes* [5]. They propose an algorithm for transforming a scheduler into an approximate decision tree representation and show that often, an approximation up to a certain error is sufficient to capture the essence of a strategy, but can significantly reduce its complexity and thereby improve its readability.
As the starting point for this thesis, the following sections aim to convey a good understanding of this algorithm, before going into detail on the adaptations that were made to improve it.

3.2 Original Algorithm

This section gives an explanation of the strategy learning approach with decision trees as presented in [5].

3.2.1 Decision Tree Learning

Decision Tree

A decision tree can be used to visually represent a decision making algorithm or strategy. It uses a hierarchical structure of decisions. Every inner node poses a question, and depending on the answer, refers to one of its child nodes, culminating in a

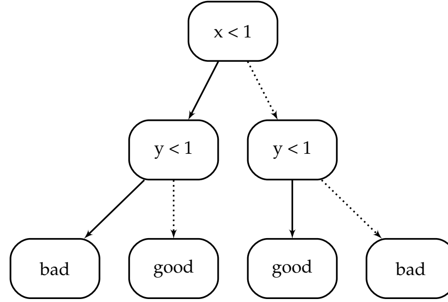


FIGURE 3.1: Decision tree for a binary x XOR y.

leaf node which is labelled with the result of the algorithm.
In the following we only define a binary tree with a single (binary) leaf label, but other variants with arbitrary branching factors and label attributes exist.

Definition 3.2.1. *A decision tree for a set of variables $\mathcal{V} = \{x_1, \dots x_m\}$ with domains $X_i = \mathcal{D}(x_i)$ and fix variable ordering, is a tuple $\mathcal{T} = (T, \rho, \theta)$ where T is a finite binary tree with a set of inner nodes N, root node n_0 and a set of leaf nodes L, ρ assigns to every inner node a predicate $[x_i \sim c]$ where $i \in \{1, \dots, m\}$, $c \in X_i$, $\sim \in \{\leq, <, \geq, >, =\}$ and θ assigns to every leaf a value in $\{good, bad\}$.*

An instance for a decision tree $\mathcal{T}$ is an explicit ordered variable valuation $\bar{x} = (\bar{x}_1, \dots, \bar{x}_m)$ where $\bar{x}_i \in X_i$. An instance satisfies a predicate $\bar{x} \models [x_i \sim c]$ if $\bar{x}_i \sim c$ evaluates to true.
A path of $\mathcal{T}$ is a sequence $\tau = n_0 n_1 \dots n_k n_{k+1}$ of inner nodes $n_i \in N$ for $0 \leq i \leq k$, starting with the root node n_0 and ending in a leaf node $n_{k+1} \in L$ and for every $1 \leq i \leq k$ node n_{i+1} is a child of n_i.

Definition 3.2.2. *The language $\mathcal{L}(\mathcal{T})$ of the tree is a set of instances. An instance $\bar{x}$ is in the language iff there exists a path $\tau = n_0 \dots n_k n_{k+1}$ such that for every inner node n_i, n_{i+1} is the left child of n_i exactly when $\bar{x} \models \rho(n_i)$, and the leaf is labelled with "good": $\theta(n_{k+1}) = good$.*

Example 3.2.1. *Figure 3.1 shows a decision tree for variables $\{x, y\}$ with common domain $\{0, 1\}$. The left child of a node is always indicated by a straight line, the right child by a dotted line. Let n_0 be the root node, n_l its left child and n_r its right child, as well as l_0, l_1, l_2, l_3 be the leaf nodes from left to right, then we can describe the unique path for instance $(0, 1)$ as $n_0 n_l l_1$. Since $\theta(l_1) = good$ the instance is in the language of $\mathcal{T}$. If we repeat this for all other candidate instances, we obtain $\mathcal{L}(\mathcal{T}) = \{(0, 1), (1, 0)\}$.*

Training

A decision tree can be constructed from a *training set* of data, consisting of instances labelled with either "good" or "bad". Formally, a training set is a set $\{(\bar{x}_1, label_1), \dots, (\bar{x}_n, label_n)\}$ where $label_i \in \{good, bad\}$.
The goal is a tree that reflects the training set, i.e. an instance in the training set that was labelled with "good" should be in the language of the tree. The training algorithm constructs the tree node-wise; at each node it determines the best predicate to split the training set into its children, such that instances labelled with "good" are successively separated from instances labelled with "bad". When a node only has instances with one label, a leaf with the corresponding label is created.

The following pseudo-code illustrates the training according to the well-known C4.5 algorithm, which uses information gain to decide which predicate to split on in each node [19]. Intuitively, information gain is a measure for how well a split separates the "good" instances from the "bad" instances.

1. The entire training set is assigned to the root node.
2. Given a node with associated training (sub)set:
   - If the instances in the set are all labelled good or all labelled bad, create a leaf node labelled correspondingly.
   - Otherwise
     - choose the predicate with the highest information gain to split on,
     - assign the data subset that satisfies the predicate to the left child and the subset that does not satisfy it to the right child,
     - perform step 2 for each child.

The resulting tree is a precise representation of the training data in the sense that instances with label "good" are in the language of the tree, and instances with the label "bad" are not.
The final tree can optionally be *pruned*, which means that some of its subtrees are cut or merged. This process is steered by so called *pruning parameters*, which vary between different implementations, and provide a way to control tree size at the cost of accuracy.

3.2.2 Scheduler Learning

Training Data

In order to learn a scheduler with a decision tree, it is necessary to first create a *training set* of data. Let $\mathcal{M} = (S, A, Act, P, \bar{s})$ be an MDP, $\Diamond T$ a reachability property and $\sigma : S \to Act$ a scheduler that optimizes said property for $\mathcal{M}$, i.e. $Pr_{\max}(\Diamond T) = Pr^{\sigma}(\Diamond T)$.
Furthermore, we assume that there is a set of variables $\mathcal{V} = \{x_1, \dots x_m\}$ with domains $Dom(\mathcal{V}) = \{X_1, \dots, X_m\}$ associated with $\mathcal{M}$ such that every state s corresponds to a variable valuation for $\mathcal{V}$ (such as is the case if the MDP was constructed from a symbolic model description such as a PRISM program (Chapter 4)).
We train a decision tree $\mathcal{T}$ over the variables $\mathcal{V} \cup \{action\}$, with $\mathcal{D}(action) = A$, which means that the entries of the training set are of the form $(\nu(s), a, label)$, where $\nu : S \to v(\mathcal{V})$ is an encoding of a state s as its variable valuation, $a \in Act(s) \subseteq A$ and $label \in \{good, bad\}$. Intuitively, a state-action instance $(\nu(s), a)$ is considered *good* if the scheduler chooses action a in state s, and *bad* otherwise. So what the tree ultimately learns is the behaviour of the scheduler.

Technically, σ on its own may already serve as sufficient input for the algorithm. However, to obtain more compact representations, Brázdil et al. [5] perform one additional step based on a heuristic they call the *importance* of a state $s \in S$ conditioned on σ;

$$Imp^{\sigma}(s) := Pr^{\sigma}[\Diamond s \mid \Diamond T], \tag{3.1}$$

the probability of visiting s conditioned on the event that a state in the target set T is reached afterwards. The idea is that decisions made in states with a high importance

have a greater impact on whether the target is reached, and more emphasis should be placed on learning them. This is achieved by repeating the training instances corresponding to states with high importance multiple times.
Imp^{σ} is not computed exactly, but rather estimated using simulation of σ on $\mathcal{M}$. Here, a simulation is a random experiment which determines a (finite) path $\pi = s_0 a_0 \dots s_n \in Paths_{fin}(\mathcal{M})$ by state-wise evaluation via a random number generator of the probability distributions $\sigma(s_i) \in Dist(A)$ and (for result a_i) μ_{s_i,a_i}. The simulation terminates in a state s_n with either $s_n \in T$ or a state with $Pr^{\mathcal{M}_{s_n}}_{\max}(\Diamond T) = 0$ probability to reach a target state. The resulting finite path is also called a run of the simulation.

If $runs_{\Diamond T}$ is the total number of simulation runs that reach a target state, and $runs_{\Diamond s \cap \Diamond T}$ is the number of runs where s and a target state were reached, then the importance function can be approximated as

$$\frac{runs_{\Diamond s \cap \Diamond T}}{runs_{\Diamond T}} = Imp^{\sigma}(s). \tag{3.2}$$

Given that a set number of target runs $c := runs_{\Diamond T}$ are simulated, decisions in important states can then be stressed in the learning algorithm by repeating the corresponding training instances $runs_{\Diamond s \cap \Diamond T}$ times.

Decision Tree Scheduler

The result of the training algorithm is a decision tree which represents a scheduler. We can compute an explicit scheduler representation from the decision tree by determining which state-action instances are in the language of the tree. For every state s, compute the set

$$goodActions(s) = \{a \in Act(s) \mid (\nu(s), a) \in \mathcal{L}(\mathcal{T})\}.$$

This is achieved as follows. For every instance $(\nu(s), a)$ traverse the tree on the unique path $\tau = n_0 \dots n_d n_{d+1}$ with leaf $n_{d+1} \in L$ where for every inner node $n_i \in N$; n_{i+1} is the left child of n_i iff $(v(s), a) \models \rho(n)$, and otherwise n_{i+1} is the right child of n_i. If $\theta(n_{d+1}) = good$, then $a \in goodActions(s)$.

Definition 3.2.3. *The scheduler represented by decision tree* $\mathcal{T}$ *is* $\sigma^{\mathcal{T}} : S \to Dist(Act)$ *with*

$$\sigma^{\mathcal{T}}(s) = \begin{cases} \mu_{uni}(goodActions(s)) & \text{if} \quad goodActions(s) \neq \emptyset \\ \mu_{uni}(Act(s)) & \text{else} \end{cases}.$$

It is possible that not every state has exactly one good action, since not all states are learned equally and variable valuations necessarily have overlaps. In such a case, the action will be chosen uniformly from either the pool of good actions, or if there are no good actions, from the full set of enabled actions.

3.2.3 Approximative Representation and Error Computation

An unpruned tree is still likely to be too large to qualify as readable which will be shown in Chapter 5. To understand the represented strategy, it is not necessary to have an exact representation, since it is likely to contain a lot of unnecessary detail. In such a case, pruning can be used to reduce the tree's size. To ensure that the represented scheduler does not veer too widely from its original, the error between

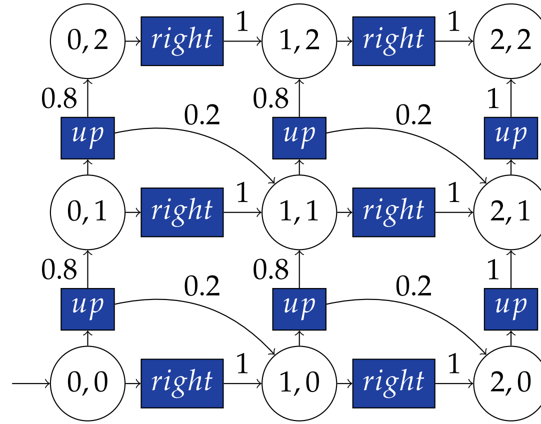


FIGURE 3.2: MDP for the climbing route problem.

the two presentations is required to be smaller than some $\varepsilon > 0$.
The error is computed by converting the decision tree to an explicit representation of $\sigma^{\mathcal{T}}$, which is used to compute the induced Markov Chain $\mathcal{M}_{\sigma^{\mathcal{T}}}$. A model checker can then determine $\Pr^{\sigma^{\mathcal{T}}}[\Diamond T]$. The relative representation error of the tree compared to the original scheduler is

$$\frac{|\Pr^{\sigma}[\Diamond T] - \Pr^{\sigma^{\mathcal{T}}}[\Diamond T]|}{\Pr^{\sigma}[\Diamond T]}.$$

To obtain a good degree of pruning (i.e. small tree size with error $< \varepsilon$), binary search on the pruning parameter in question is used.

Example 3.2.2. *Consider again the scenario of a mountain climber scaling a mountain represented by a 3x3 grid. He starts from position (0,0) and his goal is to reach position (2,2). Whenever he moves upward, there is a 20% that the wind forces him to climb to the right instead. We can model this scenario as the MDP in Figure 3.2. (For simplicity, we omit the actions left and down from the graph.) The state set S is given by the valuations of two variables x, y with domain $\mathcal{D}(x) = \mathcal{D}(y) = \{0, 1, 2\}$ that indicate the climber's position on the mountain. We are interested in the (maximum) reachability problem $\Diamond T$ with target state set $T = \{(2, 2)\}$.*
We can specify an optimal (deterministic) scheduler $\sigma : S \to A$ for the problem as

$$\sigma((x, y)) = \begin{cases} right & \text{if } x < 2 \\ up & \text{if } x = 2 \end{cases}$$

Note that the scheduler also defines a behaviour for states like $(0, 1)$ which do not lie on the climber's optimal route. Since these states are not reachable in an DTMC induced by an optimal scheduler, they can be assigned arbitrary actions. Simulation of σ on the MDP will never reach such a state and results in it being assigned state importance 0, which indicates that it is irrelevant for the strategy: Decisions in this state are not be learned by the decision tree.
A decision tree representation for σ is depicted in Figure 3.3. It exactly represents the expected strategy with a relative representation error of 0%.

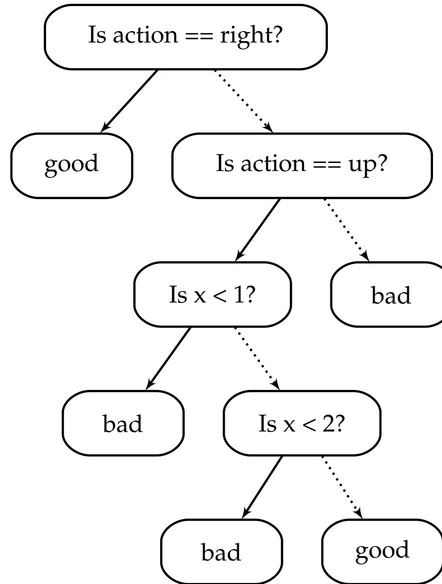


FIGURE 3.3: Decision tree representation of an optimal scheduler for the climbing route problem.

3.3 Adaptations

In the following, we first present adaptations to the decision tree representation and algorithm. First are action trees, a special class of subtrees for decision tree representations, followed by an alternative method of computing importance functions via linear programs.

3.3.1 Action Trees

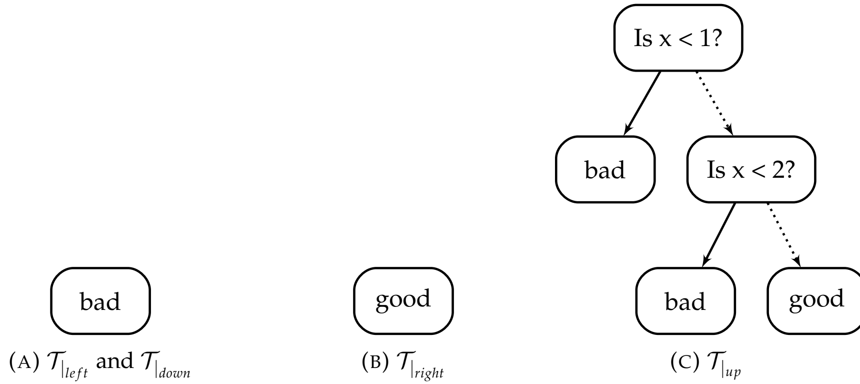
Sometimes it is interesting to know when one action of the MDP in particular is considered good according to the strategy. For this purpose, we can compute a subtree of the entire decision tree scheduler representation whose language is the subset of states in which the action is good.

Definition 3.3.1. *Let $\mathcal{M} = (S, A, Act, P, \bar{s})$ be an MDP and $\mathcal{T}$ a decision tree representation of a scheduler for $\mathcal{M}$ over the variables $\mathcal{V} \cup \{action\}$. The action-restricted decision tree $\mathcal{T}_{|_a}$ for an action $a \in A$ is the unique tree over the variables $\mathcal{V}$ that replaces every inner node $n \in N$ with predicate $\theta(n) = [action = act]$ by its left subtree if $\theta(n)$ is satisfied by the substitution $act = a$, and by its right subtree otherwise.*

Example 3.3.1. *Consider again the decision tree in Figure 3.3 which represents a scheduler for an MDP with action set $A = \{right, left, up, down\}$. The action-restricted trees $\mathcal{T}_{|_{left}}$, $\mathcal{T}_{|_{down}}$, $\mathcal{T}_{|_{right}}$ and $\mathcal{T}_{|_{up}}$ are depicted in Figure 3.4.*
The tree for right consists only of a single leaf node with label "good" because the climber should always move right when possible according to the strategy. Similarly, the trees for $down$ and $left$ consist only of a leaf node labelled with "bad" because the climber should never move down or to the left. The action up is only good in states with $x \geq 2$, i.e. on the far right side of the grid.

Lemma 3.3.1. *The language of an action-restricted tree $\mathcal{T}_{|_a}$ is*

$$\mathcal{L}(\mathcal{T}_{|_a}) = \{(\nu(s)) \mid (\nu(s), a) \in \mathcal{L}(\mathcal{T})\}.$$



(A) $\mathcal{T}_{|_{left}}$ and $\mathcal{T}_{|_{down}}$ (B) $\mathcal{T}_{|_{right}}$ (C) $\mathcal{T}_{|_{up}}$

FIGURE 3.4: Action trees for the decision tree of the route problem.

Proof (sketch). An instance $\bar{x}$ is in the language of the tree if the unique path, given by $\tau = n_0 \dots n_k n_{k+1}$, where n_{i+1} is the left child of n_i if $\bar{x} \models \rho(n_i)$, has a leaf node labelled with "good".
Then the truth of the Lemma is intuitively clear if we consider how the action-restricted tree is constructed: Nodes that branch on the action variable are replaced by either the left or right subtree depending on the action a. Therefore we can construct from the unique path for instance $(\nu(s), a)$ in $\mathcal{T}$ a path for $\nu(s)$ in $\mathcal{T}_{|_a}$ by dropping all corresponding nodes. Conversely, we can lift a path in the restricted tree to a full path by reinserting all dropped nodes. In both cases, the leaf remains the same (labelled with "good"), which entails that the instance is contained in the language. □

3.3.2 State Importance with Linear Programs

The original algorithm by Brázdil et al. [5] uses simulation to approximate the importance function that decides how often training instances are learned. One decided disadvantage of this approach is that the result and the resulting decision tree representation can vary from execution to execution. Depending on the model and sample size, the differences may even be substantial, as we show in Chapter 5. In order to produce the best result, the algorithm potentially has to be run many times. We propose an alternative deterministic means of computing an importance measure through linear programming and show that the resulting trees are roughly of the same size and accuracy (Chapter 5). In a nutshell, the approach amounts to the formulation of a flow problem on the state graph of the MDP as a linear program, a special form of optimization problem.

Definition 3.3.2. ***Linear programming*** *is a method for solving an optimization problem described as a* ***linear program (LP)****, which consists of an objective function and a set of linear (in)equality constraints. A linear program can be expressed in canonical form as*

$$\begin{aligned} \max \quad & \boldsymbol{c}^T \boldsymbol{x} \\ \textit{subject to } & A\boldsymbol{x} \leq \boldsymbol{b} \\ & \boldsymbol{x} \geq 0 \end{aligned}$$

where $\boldsymbol{x}$ *is the vector of n solution variables,* $\boldsymbol{b} \in \mathbb{Q}^m$ *and* $\boldsymbol{c} \in \mathbb{Q}^n$ *are vectors of coefficients and* $\boldsymbol{A} \in \mathbb{Q}^{m \times n}$ *is a matrix of coefficients.*

The idea is inspired by [10], who also formulate a flow problem over probabilistic automata to check multi-objective queries with rewards. We use two different

versions of the LP.
The first is a formulation over an MDP and produces both an optimal scheduler and an importance heuristic in one step, allowing us to skip the initial scheduler generation. The second uses a scheduler generated by a model checker (e.g. via value iteration) and is formulated over the induced MC.
We describe here the formulation of both LPs and compare results of both variants with simulation in Chapter 5.

LP Formulation on MDPs

Let $\mathcal{M} = (S, A, Act, P, \bar{s})$ be an MDP, $T \subseteq S$ the set of all target states and $S_{=0} = \{s \mid Pr^{\mathcal{M}_s}_{\max}(\Diamond T) = 0\}$ the set of states which have 0 probability to reach a target state. $S_{=0}$ can be determined by a model checker.
We want to formulate a max flow problem on the state graph of the MDP, such that the initial state is the source and the states in $T \cup S_{=0}$ are the sinks. In particular we want to maximize the flow over the target states. Therefore we extend the MDP by a bottom state $s_\perp$ and redirect all outgoing edges of states in $T \cup S_{=0}$ to it.
Formally, we construct the MDP $\mathcal{M}_\perp = (S_\perp, A_\perp, Act_\perp, P_\perp, \bar{s})$ with $S_\perp = S \cup \{s_\perp\}$, extended action set $A_\perp = A \cup \{a_\perp\}$ and set the enabled actions for states $s \in T \cup S_{=0}$ to $Act(s) = \{a_\perp\}$. The transition function alters only for states $t \in T \cup S_{=0}$:

$$P_\perp(t, a, s) = 0 \quad \forall a \in A, s \in S$$
$$P_\perp(t, a_\perp, s') = \begin{cases} = 0 & s' \neq s_\perp \\ = 1 & s' = s_\perp \end{cases}.$$

The LP has a solution variable $y_{s,a}$ for every state $s \in S$ and action $a \in Act_\perp(s)$. We can interpret the value of $y_{s,a}$ as the flow over the edge $s \xrightarrow{a} \ldots$, and obtain the flow *over* a state s as

$$\mathit{flow}(s) = \sum_{a \in Act(s)} y_{s,a}.$$

The LP should maximize the sum of $\mathit{flow}(t)$ over target states $t \in T$. Furthermore, for a flow problem it is crucial that the amount of flow that goes into a state also flows out again, and that there is no negative flow over any edge. We can ensure this by adding appropriate constraints to the LP.
Let $Pre(s) := \{s' \in S \mid \exists a \in Act_\perp(s') : P_\perp(s', a, s) > 0\}$ be the set of s-predecessors.
We formulate the LP as follows:

$$\max \quad \sum_{t \in T} y_{t,a_\perp}$$

subject to

$$\sum_{a \in Act_\perp(s)} y_{s,a} - \sum_{\substack{s' \in Pre(s) \\ a' \in Act(s')}} y_{s',a'} \cdot P(s', a, s) = \begin{cases} 0 & \forall s \in S \setminus \{\bar{s}\} \\ 1 & \text{if } s = \bar{s} \end{cases}$$
$$y_{s,a} \geq 0 \quad \forall s \in S, a \in Act_\perp(s)$$

The first constraint expresses the flow conservation equality with one exception: The initial state $\bar{s}$ has an (additional) input of one unit. The last constraint ensures that there is no negative flow over any edge.
The objective function ensures that the flow over target states is maximized, which

is similar to maximizing their reachability. In fact, for a solution of the LP, the value of the optimization function is equal to $Pr^{\mathcal{M}}_{\max}(\Diamond T) = \max_\sigma Pr^{\mathcal{M}_\sigma}(\Diamond T)$ [10]. As a result, the LP also induces a maximizing scheduler σ^{LP} on $\mathcal{M}$.

Definition 3.3.3. *For an MDP $\mathcal{M} = (S, A, Act, P, \bar{s})$ and the solution variables $y_{s,a}$ of the above LP, the induced scheduler $\sigma^{LP} : S \to Dist(Act)$ is defined as*

$$\sigma^{LP}(s)(a) = \frac{y_{s,a}}{\sum_{a \in Act(s)} y_{s,a}}$$

if the denominator is nonzero, and arbitrarily otherwise.

In particular, the denominator is zero when there is no flow over the state s. In this case, the decision in s is irrelevant for the strategy.
We can train a tree to represent σ^{LP} by learning state-action instances (s, a) as good if $y_{s,a} > 0$ and bad if $y_{s,a} = 0$.
To determine how often an instance should be put to the training sequence, we still need the notion of *state importance*, which we define as the flow over $s \in S$

$$Imp^{LP}(s) = flow(s) = \sum_{a \in Act(s)} y_{s,a} \quad \forall s \in S. \tag{3.3}$$

As with the importance function approximated by simulation (Section 3.1), we learn training instances corresponding to states with high importance more often. In practice, the values $Imp^{LP}(s)$ are small (typically ≤ 1), therefore we scale by a factor c such that a set number of training instances are learned in total.

LP Formulation on MCs

If we already have a scheduler σ (potentially different from σ^{LP}) that we want to represent, we cannot use the above LP formulation. However, we can use a very similar LP formulated over the induced MC $\mathcal{M}_\sigma$ to compute an importance function for σ.
As above, we need to transform $\mathcal{M}_\sigma$ by addition of a bottom state $s_\perp$. Formally, we construct MC $\mathcal{M}_{\sigma,\perp} = (S_\perp, P_{\sigma,\perp}, \bar{s})$ with $S_\perp = S \cup \{s_\perp\}$ and altered transition function for states $t \in T \cup S_{=0}$:

$$\begin{aligned} P_\perp(t, s) &= 0 \quad \text{for } s \neq s_\perp \\ P_\perp(t, s_\perp) &= 1. \end{aligned}$$

The corresponding LP has solution variables y_s for every $s \in S$ which represent the flow over the state.
We define $Pre(s) = \{s' \mid P_\perp(s', s) > 0\}$ as the set of s-predecessors in $\mathcal{M}_{\sigma,\perp}$. The LP itself only consists of the flow constraints:

$$\begin{aligned} &\max \quad 0 \\ &\text{subject to} \\ &\qquad y_s - \sum_{s' \in Pre(s)} y_{s'} \cdot P(s', s) = \begin{cases} 0 & \forall s \in S \setminus \{\bar{s}\} \\ 1 & \text{if } s = \bar{s} \end{cases} \\ &\qquad y_s \geq 0 \quad \forall s \in S \end{aligned}$$

As before, we have a constraint for flow conservation and one for non-negative flow. The LP does not have an optimization function because the model is already scheduled by the scheduler σ. We can therefore view the values y_s as the flow induced by σ. If σ is a maximizing scheduler, the value of the flow over target states $\sum_{t \in T} y_t$ is equal to the optimal value obtained by the LP formulation on MDPs.
The importance function used to train a tree for σ is

$$Imp^{LP_\sigma}(s) = y_s,$$

the amount of flow over state s. As with $Imp^{LP}(s)$, an instance corresponding to state s is learned $c \cdot Imp^{LP_\sigma}(s)$ many times, where c is a scaling factor that ensures that a set number of instances is learned in total.

Chapter 4

Scheduler Representation in the PRISM Language

4.1 Introduction

In Chapter 3 we have shown how approximation and visualization in the form of decision trees can help improve the readability of schedulers. Another factor in readability may be the familiarity with the representation language of the input model. PRISM is a language which is widely used to specify probabilistic models for model checking [14, 8]. Therefore we propose an alternative representation of a scheduler as a PRISM module, show how it can be computed from a decision tree representation and prove that it schedules the original program correctly. The resulting program can be viewed as a symbolic description of a "critical subsystem" similar to [9].

4.2 PRISM Language

4.2.1 Syntax

The PRISM language is used for specification of probabilistic models like MCs and MDPs. We describe here a subset of the full language which suffices for our purposes. More information can be found in the online PRISM manual [21].

The model type can be specified by keywords **dtmc**, **mdp** and determines how non-determinism (which module performs which command next) is handled, e.g. for the keyword **dtmc** it is resolved by uniform distributions.
A PRISM program $\mathcal{P}$ is composed of variables, of type bool or bounded integer, and modules. Modules are defined within the keywords **module** name ... **endmodule**. Their behaviour is specified by guarded commands of the form

```
[label] : guard -> p_1 : update_1 + ... + p_n : update_n;
```

where *label* is a synchronization label. The *guard* is a logical expression over predicates over the variables of the model. If it is true, one of the transitions specified by update_i may be taken. Each *update* is an expression $(v_1 = expr_1) \wedge \ldots \wedge (v_m = expr_m)$ that assigns new values to variables. While the guard can contain variables local to other modules, every module can only write to its own local (or global) variables. Each update is associated with a probability p_i. The p_i are required to sum up to 1 for one command.
Commands can be given synchronization labels to force modules to *synchronize* over identically named commands, i.e. transitions from these commands must be performed simultaneously.
Furthermore, modules can be duplicated by renaming them and all local variables

to fresh identifiers. Optionally, commands may also be relabelled. PRISM also supports the definition of *formulas*, expressions which can be (re)used anywhere in the program, and *state-labels*, which are of boolean type and identify sets of states.

4.2.2 Semantics

This section is a summary of the PRISM semantics (for MDPS and MCs), the entirety of which can be found at [20].

The semantics of a PRISM program is a probabilistic model of the specified type. Let $\mathcal{V} = \{v_1, \ldots, v_m\}$ be the set of all local and global variables of the PRISM program. Regardless of model type, the set of states is given as the set of all possible variable valuations $(x_1, \ldots, x_m)$ of $\mathcal{V}$. The initial variable valuation (the initial state $\bar{s}$) can be specified in an init clause for all variables, or, if not specified, is taken to be the lowest value in the range for integers and as false for bools.
The modules of the program are combined into one system module by pairwise parallel composition of modules over the set of commands with common synchronization labels L. The composition of two modules $M = M_1|[L]|M_2$ is constructed as follows.

1. for each command $[]g \rightarrow \lambda_1 : u_1 + \ldots + \lambda_n : u_n$ of M_1 (or M_2), add $[] \rightarrow \lambda_1 : u_1 + \ldots + \lambda_n : u_n$ to the commands of M;
2. for each $l \notin L$ and command $[l]g \rightarrow \lambda_1 : u_1 + \ldots + \lambda_n : u_n$ of M_1 (or M_2), add $[l] \rightarrow \lambda_1 : u_1 + \ldots + \lambda_n : u_n$ to the commands of M;
3. for each $l \in L$, command $[l]g \rightarrow \lambda_1 : u_1 + \ldots \lambda_n : u_n$ of M_1 and $[l]g' \rightarrow \lambda'_1 : u'_1 + \ldots \lambda'_{n'} : u'_{n'}$ of M_2,
   add

   $$\begin{aligned} [l]g\&g' \rightarrow & \lambda_1 \cdot \lambda'_1 : u_1\&u'_1 + \ldots + \lambda_n \cdot \lambda'_1 : u_n\&u'_1 \\ & p_1 \cdot \lambda'_2 : u_1\&u'_2 + \ldots + \lambda_n \cdot \lambda'_2 : u_n\&u'_2 \\ & \vdots \\ & \lambda_1 \cdot \lambda'_{n'} : u_1\&u'_{n'} + \ldots + \lambda_n \cdot \lambda'_{n'} : u_n\&u'_{n'} \end{aligned}$$

   to the commands of M.

Let C be the multiset of commands generated by the parallel composition of all modules according to the above rules. If the specified model type is **`mdp`**, the set of actions A is given as a one-to-one correspondence to C. Therefore we often use a and c interchangeably to refer to an action in the model when it is clear from the context. If we neglect action labels, every command $c \in C$ is of the form

$$[]g \rightarrow \lambda_1 : u_1 + \ldots + \lambda_n : u_n$$

The guard defines a subset of states in which the command is enabled: $S_c = \{s \in S \mid s \models g\}$. An action a is in $Act(s)$ if c is the command corresponding to a and $s \in S_c$. Every update u_i of c corresponds to a transition that the system can make when in a state $s \in S_c$. The transition is defined by assigning every variable a new value, i.e. defining a new variable valuation. If the update is $u_i = (v'_1 = expr_1) \wedge \ldots \wedge (v'_m = expr_m)$, it assigns new values $expri_i$ to variables v_i, where $expr_i$ is a function over

program variables. Every update therefore defines a transition function $u_i : S_c \to S$:

$$u_i(s) = (expr_1(s), \dots, expr_m(s)),$$

which applies the expressions to state s. Every update u_i is associated with a probability λ_i. The PRISM language requires that the sum of probabilities is $\sum_{1 \le i \le n} \lambda_i = 1$. For every $s \in S_c$ we can therefore define a probability distribution $\mu_{c,s} : S \to [0,1]$:

$$\mu_{c,s}(t) := \sum_{\substack{1 \le i \le n \\ \wedge u_i(s) = t}} \lambda_i$$

which gives the transition probability from s to t if command c is executed.

MDP Probabilistic Transition Function

For an MDP, we take the probabilistic transition function $P : S \times A \times S \to [0,1]$ to be

$$P(s,a,t) = \begin{cases} \mu_{c,s}(t) & \text{if } s \in S_c \\ 0 & \text{else.} \end{cases}$$

MC Probabilistic Transition Function

For the transition probability function $P : S \times S \to [0,1]$ of an MC, we first construct a transition probability matrix

$$\bar{P}(s,t) = \sum_{c \in C} \mu_{c,s}(t),$$

and normalise its rows to obtain P:

$$P(s,t) = \frac{\bar{P}(s,t)}{\sum_{s' \in S} \mu_{c,s}(s')}.$$

As explained in [20], the normalisation is required because non-determinism can lead to values in one row to sum up to more than one and can be viewed as a uniform distribution between the non-deterministic choices.

LISTING 4.1: PRISM example program for the mechanic problem.

```
1  mdp
2
3  const double EOT = 24;
4  const int SCREWS = 2;
5
6  formula pBreak = 1/(EOT*2) * (time-lastMaintained1);
7
8  module generator1
9
10         broken1: bool init false;
11         lastMaintained1: [0..EOT-1] init 0;
12
13         [maintain1] !broken1 -> (lastMaintained1'=time);
14         [maintain2] true -> pBreak: (broken1'=true) + 1-pBreak: true;
15         [wait]    true -> pBreak: (broken1'=true) + 1-pBreak: true;
16
17 endmodule
18
19
20 //construct further generators with renaming
21 module generator2 = generator1 [broken1 = broken2, lastMaintained1 =
       lastMaintained2, maintain1=maintain2, maintain2=maintain1] endmodule
22
23
24 module mechanic
25         screws: [0..SCREWS] init SCREWS;
26         [maintain1] screws > 0 -> (screws'=max(screws-1, 0));
27         [maintain2] screws > 0 -> (screws'=max(screws-1, 0));
28         [wait] true -> true;
29 endmodule
30
31
32 module timer
33         time : [0..EOT-1] init 0;
34
35         [maintain1] time<EOT -> (time'=min(EOT-1, time+1));
36         [maintain2] time<EOT -> (time'=min(EOT-1, time+1));
37         [wait] time<EOT -> (time'=min(EOT-1, time+1));
38 endmodule
39
40
41 label "successfulDay" = !broken1 & !broken2 & time=EOT-1;
```

Example 4.2.1. *Listing 4.1 shows a PRISM program for an MDP which models the following scenario. A mechanic's job is to maintain two generators so they don't break down. A broken down generator can no longer be maintained and is out of order for the rest of the day. There are 24 hours in the mechanic's work day, and every hour he has the option to maintain one of the generators or do nothing (wait). Every hour that a generator is not maintained, it has a chance to break down that scales linearly with how many hours have already passed. This behaviour can be reset by maintaining the generator. When the mechanic maintains a generator, he consumes one screw, and he only has a limited supply of them.*
The PRISM program in Listing 4.1 models this as follows. The mechanic is represented by the mechanic module (line 24-29) which has an integer variable to keep track of the number of screws left (line 25). The commands maintain1, maintain2, wait are the actions the mechanic can take. If the mechanic maintains a generator, (s)he requires one screw (guard) and consumes one screw (update). The guard of the wait command is true, since there is no

restrictions on doing nothing.
The timer module (line 32-38) keeps track of the time of day as an integer that ranges from 0 to EOT (end of time) (line 33) and synchronizes with the mechanic module over all the commonly labelled commands. That is, every time the mechanic performs an action, time increases by one unit.
*The two generators are given as the generator1 (line 8-17) and the generator2 module, which is a copy of the first using PRISM's renaming feature (line 21). A generator has a boolean variable which indicates whether it is broken (line 10) and an integer variable that ranges from 0 to EOT (end of time) and stores the time it was last maintained (line 11). A broken down generator can no longer be maintained, indicated by the guard of the command maintain1 (line 13). Whenever the generator is maintained, its maintenance time is set to that hour in the day. When the other generator is maintained or the mechanic waits, there is a chance for it to break down indicated by the probabilistic updates in line 14, 15. The chance for a breakdown is computed by the formula pBreak (line 6) with a base chance of 1/(EOT*2) that scales linearly with time. The generator modules also synchronize with mechanic and timer over all commonly labelled commands, which means that, for example, the mechanic can only maintain generator1 if the conjunction of guards !broken1 & screws > 0 & time < EOT is satisfied.*
Finally, the label "successfulDay" (line 41) identifies a set of states that is of particular interest: The end of the day has been reached and neither generator has broken down.

4.2.3 Simple PRISM Program

To compute the PRISM scheduler representation, we require that the input PRISM program has a special property: Every set of commands that makes up an action in the MDP must have a unique synchronization label which we can use to address the action in the scheduler module. We define programs who satisfy this property as simple.

Definition 4.2.1. *Let $\mathcal{P}$ be a PRISM program and C the set of commands that appear in $\mathcal{P}$. $\mathcal{P}$ is called a* simple *PRISM program if*

- *every command in C has a non-empty label, and*
- *for every command in $c \in C$: If c is a command in module m labelled with l, then there is no other command in m that is also labelled with l.*

Every program $\mathcal{P}$ can be transformed into a simple program $\mathcal{P}'$ without changing its semantics by renaming and duplication of commands:

1. Every command with an empty synchronization label is given a fresh label.
2. If there is a module which contains a set of commands C_l which are labelled with a common label l, but there is no other module which contains a command labelled with l, then rename all commands in C_l to fresh labels.
3. For every set of commands $C_l = \{c_0, \dots c_n\}$ such that no two commands are in the same module, and they all share the same synchronization label l: If there is a command c_i, $1 \leq n$, which occurs in module m, and there is a command $c' \notin C_l$ which also appears in module m and has the same label l: Duplicate all the commands $c \in C_l$ and label them with a fresh common label.

In the worst case, the resulting program has $\prod_{i=0}^{n} m_i$ many commands, where n is the number of modules, and m_i is the number of commands in module i.

LISTING 4.2: Example of a non-simple PRISM program.

```
1  mdp
2
3  module m1
4          x: [0..10] init 0;
5
6          []       true -> (x'=0);
7
8          [subtract] x > 2 -> 0.5: (x'=0) +0.5: (x'=1);
9
10         [add] x < 10 -> 0.8: (x'=x+1) + 0.2: (x'=x);
11         [add] x < 9  -> 0.6: (x'=x+2) + 0.4: (x'=x+1);
12
13 endmodule
14
15
16 module m2
17         y: [0..10] init 0;
18
19         []           true -> (y'=0);
20
21         [subtract] y > 3 -> 1/3: (y'=0) + 1/3: (y'=1) + 1/3: (y'=2);
22
23         [add] y < 10 -> 0.8: (y'=y+1) + 0.2: (y'=y);
24         [add] y < 9  -> 0.6: (y'=y+2) + 0.4: (y'=y+1);
25
26 endmodule
```

Example 4.2.2. *The mechanic program from Listing 4.1 is a simple PRISM program, since it contains no unlabelled commands and every label occurs at maximum once per module. However, the program from Listing 4.2 is not simple, because a) it contains two unlabelled commands and b) a command labelled with "add" appears twice in every module. As a result, the MDP built from the program contains seven actions, two which correspond to an unlabelled command, one which corresponds to a command with label "subtract" and four which all correspond to (different) commands that are labelled with "add".*
The program is transformed into a simple program by labelling the unlabelled commands with "tau1" and "tau2" respectively, and duplicating (and renaming) every "add" command twice (once per add command in the other module). The resulting simple PRISM program is shown in Listing 4.3. The MDP constructed from this program still has the same seven actions, but now each one corresponds to a command with a unique label.

LISTING 4.3: Simple version of the example PRISM program.

```
1  mdp
2
3  module m1
4          x: [0..10] init 0;
5
6          [tau1]     true -> (x'=0);
7
8          [subtract] x > 2 -> 0.5: (x'=0) +0.5: (x'=1);
9
10         [add1] x < 10 -> 0.8: (x'=x+1) + 0.2: (x'=x);
11         [add2] x < 10 -> 0.8: (x'=x+1) + 0.2: (x'=x);
12
13         [add3] x < 9  -> 0.6: (x'=x+2) + 0.4: (x'=x+1);
14         [add4] x < 9  -> 0.6: (x'=x+2) + 0.4: (x'=x+1);
15
16 endmodule
17
18
19 module m2
20         y: [0..10] init 0;
21
22         [tau2]         true -> (y'=0);
23
24         [subtract] y > 3 -> 1/3: (y'=0) + 1/3: (y'=1) + 1/3: (y'=2);
25
26         [add1] y < 10 -> 0.8: (y'=y+1) + 0.2: (y'=y);
27         [add3] y < 10 -> 0.8: (y'=y+1) + 0.2: (y'=y);
28
29         [add2] y < 9  -> 0.6: (y'=y+2) + 0.4: (y'=y+1);
30         [add4] y < 9  -> 0.6: (y'=y+2) + 0.4: (y'=y+1);
31
32 endmodule
```

4.3 PRISM Scheduler Representation

We represent a scheduler in PRISM as a module that schedules the other program modules. The resulting program is interpreted as a **dtmc** (uniform distribution over non-deterministic choices).
For construction of the scheduler module, we use a decision tree representation of the scheduler as starting point, because a translation between the two is faster and yields more compact PRISM modules than with the explicit representation. This means that the scheduled program MC also inherits the same degree of approximation from the tree representation. In fact, the MCs obtained are equivalent, which is a central result of this chapter.

Theorem 4.3.1 (Correctness)**.** *Let $\mathcal{P}$ be a PRISM program describing an MDP $\mathcal{M}$ and $\mathcal{T}$ be a decision tree representation of a scheduler σ for $\mathcal{M}$. Then the semantics of the scheduled program $\mathcal{P}_{\mathcal{T}}$ is an MC which is equivalent to the induced MC $\mathcal{M}_{\sigma^{\mathcal{T}}}$.*

We first describe the construction of $\mathcal{P}_{\mathcal{T}}$ and then prove Theorem 4.3.1 in Section 4.4.

4.3.1 Scheduler Module

For our construction we require that every action in the MDP is associated with a unique synchronization label. This is guaranteed if the input program is simple (Section 4.2.3). Since any non-simple program can be transformed into a simple program without changing its semantics, in the following we assume without loss of generality that $\mathcal{P}$ is simple.

The program $\mathcal{P}$ is transformed to a new program $\mathcal{P}_{\mathcal{T}}$ by extending it with a *scheduler* module which represents the same scheduler as the decision tree $\mathcal{T}$. Since we want the module to have the same effect as the scheduler on an MDP, we also change the program's type to be **dtmc**.
The idea is to use PRISM's synchronization over common synchronization labels to enforce the strategy given by the decision tree representation $\mathcal{T}$. We want to ensure that a (synchronized) command is only enabled if the scheduler module allows it. This is achieved by adding, for every action $a \in A$, a command

```
[a] : predicate -> true;
```

to the scheduler module, where `a` is the unique synchronization label for a. The right-hand side **true** signifies that the command specifies no updates. The `predicate` is the condition on states for which the strategy deems the choice (state, action) good. These conditions can be extracted from $\mathcal{T}$ already as a predicate. This is why we build upon the decision tree representation instead of the explicit representation.
We compute the guard predicates for a tree $\mathcal{T} = (T, \rho, \theta)$ over the variables $\mathcal{V} \cup \{action\}$ as follows. For every action $a \in A$ in the model, construct the action-restricted decision tree $\mathcal{T}_{|_a}$ (Section 3.3.1). As a reminder, $\mathcal{T}_{|_a}$ is obtained from $\mathcal{T}$ by replacing every inner node with a predicate of the form $[action = act]$ by its left subtree if it is satisfied by the substitution $act = a$, and by its right subtree otherwise. Its language contains the instances $(\nu(s))$ such that taking action a in state s is considered good by the strategy. We now want to describe the same idea as a predicate.

For a path $\tau = n_0 \dots n_k n_{k+1}$, $n_i \in N$ for $0 \leq i \leq k$, from root n_0 to leaf $n_{k+1} \in L$, define the predicate path

$$pred(\tau) := \bigwedge_{\substack{0 \leq i \leq k \\ left(n_i) = n_{i+1}}} \rho(n_i) \quad \wedge \bigwedge_{\substack{0 \leq i \leq k \\ right(n_i) = n_{i+1}}} \neg \rho(n_i).$$

The formula $pred(\tau)$ is a logical description of path τ which we can use to classify instances. For example, if we take an instance $\bar{x}$ and its unique path τ in the tree, it follows by definition that $\bar{x} \models pred(\tau)$ and there is no other path τ' such that $\bar{x} \models pred(\tau')$. If the leaf of τ has label "good", $\bar{x}$ is in the language of the tree. Therefore, we can equate predicate satisfaction with language containment.
The set of predicates under which an action a is considered good according to the tree's strategy is

$$goodPred(a) = \{pred(\tau) \mid \tau = n_0 \dots n_k n_{k+1} \text{ is a path in } \mathcal{T}_{|_a} \text{ and } \theta(n_{k+1}) = good\}.$$

If the strategy takes an action a in state s, the instance $(\nu(s), a)$ is in the language of the tree $\mathcal{T}$. The set of good predicates gives us an alternative description for when an action is good according to the strategy, which is expressed in the following lemma.

Lemma 4.3.1. *For every instance* $\bar{x} = (\bar{x}_1, \dots \bar{x}_m, a)$ *of* $\mathcal{T}$ *:*

$$\bar{x} \models \bigvee_{p \in goodPred(a)} p \quad \Leftrightarrow \quad \bar{x} \in \mathcal{L}(\mathcal{T}).$$

The proof is based on Lemma 3.3.1, which states that the language of an action restricted tree is $\mathcal{L}(\mathcal{T}_{|_a}) = \{(\nu(s)) \mid (\nu(s), a) \in \mathcal{L}(\mathcal{T})\}$. In particular, we can lift any path in the action-restricted tree $\mathcal{T}_{|_a}$ to a path in $\mathcal{T}$ and the other way around. We can use the fact that if an instance is in the language of a tree, by definition there is a unique path for that instance in the tree whose leaf is labelled with "good".

Proof. $\Rightarrow$ There exists $p \in goodPred(a)$ such that there is a path $\tau = n_0 \dots n_k n_{k+1}$ in $\mathcal{T}_{|_a}$ with $\bar{x} \models pred(\tau) = p$ and $\theta(n_{k+1}) = good$.
Then $(\bar{x}_1, \dots \bar{x}_m)$ is in the language $\mathcal{L}(\mathcal{T}_{|_a})$. According to Lemma 3.3.1 it follows that $\bar{x} \in \mathcal{L}(\mathcal{T})$.

$\Leftarrow$ By Lemma 3.3.1 we obtain $(\bar{x}_1, \dots \bar{x}_m) \in \mathcal{L}(\mathcal{T}_{|_a})$. By definition, there exists a path $\tau = n_0 \dots n_k n_{k+1}$ in $\mathcal{T}_{|_a}$ with leaf label $\theta(m_{k+1}) = good$, such that n_{i+1} is the left child of n_i iff $(\bar{x}_1, \dots \bar{x}_m) \models \rho(n_i)$. Then $(\bar{x}_1, \dots \bar{x}_m) \models pred(\tau) \in goodPred(a)$. Since $pred(\tau)$ does not contain any predicates over the action variable, it follows that $\bar{x} \models \bigvee_{p \in goodPred(a)} p$.

□

Since $p \in goodPred(a)$ does not contain predicates over the *action* variable, we can also say $\nu(s) \models p$ and write $s \models p$ for simplicity.

Let $p_1, p_2, \dots p_n$ be the set of predicates in $goodPred(a)$ for some action a. For that action, the entry in the scheduler module is

```
[action] : p_1 | p_2 ... | p_n | false -> true;
```

The disjunction with false is necessary in the case that $goodPred(a)$ is empty, in which case the action is never good according to the strategy and should be forbidden.

4.3.2 Deadlocks

Recall that not all states necessarily have a good action according to the strategy (Section 3.2.2). Until now, this introduces a deadlock into the scheduled PRISM program, since all actions are only enabled when they are good for the strategy. That means a deadlock occurs exactly in a state when according to the strategy, no action is classified as good. In such a case, we should choose uniformly from the pool of "bad actions", which is to say, all actions. This means the scheduler module needs to lift its restrictions whenever a deadlock occurs. We can achieve this by introducing a new deadlock formula to the PRISM program which expresses such a situation;

$$deadlock = \bigwedge_{a \in A} \neg \eta_a,$$

with formula η_a expressing exactly when action a is enabled. An action a is enabled when all the guards of the commands labelled with a are satisfied. For an action a, let $origin(a) := \{cmd_1, \dots, cmd_n\}$ be the set of commands in $\mathcal{P}$ labelled with a, with

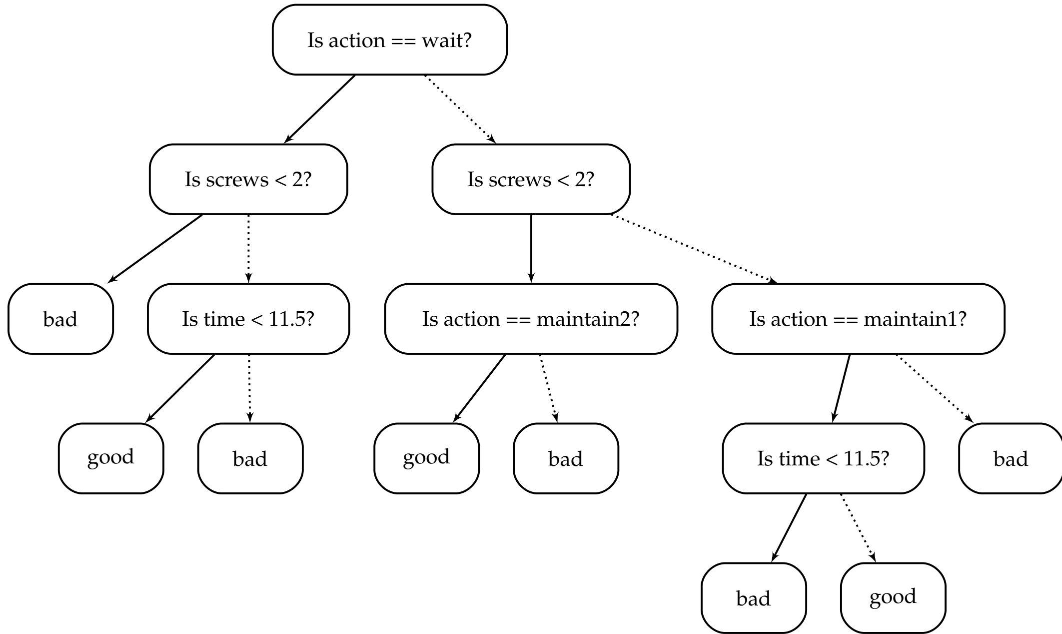


FIGURE 4.1: Decision tree representation of a strategy for the mechanic MDP and the reachability problem: Reach the end of the day (time=EOT) with no generator broken.

$cmd_i : [a]g_i \rightarrow \lambda_{i_1} : u_{i_1} + \ldots + \lambda_{i_n} : u_{i_n}$. The corresponding formula is

$$\eta_a = \bigwedge_{i=0}^{n} g_i \quad \wedge \bigvee_{p \in goodPred(a)} p.$$

Therefore the formula *deadlock* is true exactly when no action is enabled.

All that remains is to modify the guards in the scheduler module to take deadlocks into account:

```
[action] : p_1 | ... | p_n | deadlock -> true;
```

We can also drop the disjunction with $false$, since the guard *deadlock* will ensure that the command is not enabled (except in case of a deadlock) even if $goodPred(a)$ is empty.

Example 4.3.1. *Given the generator maintenance model from example 4.2.1, it might be interesting to know a strategy for the maximum reachability problem* $\Diamond$*"*$successfulDay$*", i.e. the mechanic finishes his work day without a generator breaking down. Figure 4.1 shows a tree* $\mathcal{T}$ *that represents such a strategy. Roughly, the strategy is to wait until time=12 (half time), and then maintain both generators. Considering that with advancing time the chance of a generator breakdown increases (linearly), it seems sensible to reset the probability halfway through to minimize the possibility.*
We can now compute the PRISM representation of the strategy. The sets goodPred(a) for every action $a \in A$ *are:*

$$goodPred(wait) = \{(!(screws < 2) \wedge (time < 11.5))\}$$
$$goodPred(maintain1) = \{(!(screws < 2) \wedge !(time < 11.5))\}$$
$$goodPred(maintain2) = \{(screws < 2)\}$$

From these, we can compute the deadlock formula, as well as the scheduler module. Listing 4.4 shows the additions to the program.

LISTING 4.4: Scheduled PRISM program for the generator maintenance MDP.

```
1  //Scheduled Prism Program
2  dtmc
3
4
5  //Enabled Formulas
6  formula maintain1Enabled =  !broken1 & screws > 0 & time < 24 & ( !(
       screws < 2) & !(time < 11.5) );
7  formula maintain2Enabled =  !broken2 & screws > 0 & time < 24 & screws
       < 2 ;
8  formula waitEnabled =  !(screws < 2) & time < 11.5 ;
9
10 formula deadlock = !maintain1Enabled & !maintain2Enabled & !waitEnabled
        ;
11
12 //other modules as before
13
14 module scheduler
15          [maintain1] (!(screws < 2) & !(time < 11.5) ) | deadlock ->
                true;
16          [maintain2] screws < 2  | deadlock -> true;
17          [wait] (!(screws < 2) & time < 11.5 ) | deadlock -> true;
18 endmodule
```

4.4 Correctness

To prove Theorem 4.3.1 we (symbolically) construct the command set for $\mathcal{P}_\mathcal{T}$ by parallel composition and show that for every command in the original program's semantics, it contains a command which differs only in the guard. The set of states who fulfil the guard of the scheduled command in $\mathcal{P}_\mathcal{T}$ is a subset of the states who fulfil the guard of the original command *program*. States in the difference of both sets are precisely those filtered by the scheduler $\sigma^\mathcal{T}$. Furthermore, we show that the normalisation of the rows in the probability transition matrix achieves the same result as the uniform distribution by $\sigma^\mathcal{T}$ over all good actions (or bad actions).

Proof of Theorem 4.3.1. Let $\mathcal{P}$ be a PRISM program with MDP $\mathcal{M} = (S, A, Act, P, \bar{s})$ as its semantics and let $\mathcal{T}$ be the decision tree representation of a scheduler σ for $\mathcal{M}$. Furthermore, let $\mathcal{P}_\mathcal{T}$ be the scheduled PRISM program constructed from $\mathcal{P}$ and $\mathcal{T}$. Remember that the scheduler represented by $\mathcal{T}$ was defined as $\sigma^\mathcal{T} : S \to Dist(Act)$:

$$\sigma^\mathcal{T}(s) = \begin{cases} \mu_{uni}(goodActions(s)) & \text{if } goodActions(s) \neq \emptyset \\ \mu_{uni}(Act(s)) & \text{else} \end{cases}$$

Consider an arbitrary action $a \in A$ and let $goodPred(a) = \{p_1, p_2, \ldots, p_n\}$ be the set of good predicates for a computed as described in Section 4.3.1.
Then the scheduler module has an entry of the form

$$[a]\ (p_1|p_2|\ldots|p_n) \mid deadlock\ \to true;.$$

which synchronizes with all other commands labelled with a.
Let $C_\mathcal{P}$ and $C_{\mathcal{P}_\mathcal{T}}$ be the command sets generated for the programs $\mathcal{P}$ and $\mathcal{P}_\mathcal{T}$ by

parallel composition. (Note that because we are working with the convention of simple programs, there is a one-to-one correspondence between an action a and the command labelled with a.) Since commands in the scheduler module contain no updates, there exist commands $c_{\mathcal{P}} \in C_{\mathcal{P}}$ and $c_{\mathcal{P}_{\mathcal{T}}} \in C_{\mathcal{P}_{\mathcal{T}}}$ such that the two commands differ only in their guards. More specifically, they are of the form

$$\begin{aligned} c_{\mathcal{P}}: &\quad [a] && g \rightarrow \lambda_1 : u_1 + \ldots + \lambda_n : u_n \\ c_{\mathcal{P}_{\mathcal{T}}}: &\quad [a] && g \& (p_1 | p_2 | \ldots | p_n | deadlock) \rightarrow \lambda_1 : u_1 + \ldots + \lambda_n : u_n \end{aligned}$$

As before, an update u_i defines a function from the subset of states that satisfy the command's guard to S. For $c_{\mathcal{P}_{\mathcal{T}}}$ this subset is $S_{g \wedge (p_i \vee dl)} = \{s \in S \mid s \models g \wedge (p_1 \vee p_2 \vee \ldots \vee p_n \vee deadlock)\}$. We have $S_{g \wedge (p_i \vee dl)} \subseteq S_g$, since we are adding a conjunct to the condition.
For every state that satisfies the guard of $c_{\mathcal{P}_{\mathcal{T}}}$ we obtain a probability distribution $\mu_{c,s}^{\mathcal{P}_{\mathcal{T}}} : S \rightarrow \mathbb{R}_{\geq 0}$ which is identical to $\mu_{c,s}^{\mathcal{P}}$:

$$\mu_{c,s}^{\mathcal{P}_{\mathcal{T}}}(t) = \mu_{c,s}^{\mathcal{P}}(t) = \sum_{\substack{1 \leq i \leq n \\ \wedge u_i(s) = t}} \lambda_i \quad \forall s \in S_{g \wedge (p_i \vee dl)}.$$

It expresses the transition probability from s to t if command c is executed. For all states $s \in S_g \setminus S_{g \wedge (p_i \vee dl)}$ there is no $\mu_{c,s}^{\mathcal{P}_{\mathcal{T}}}$, but there is a $\mu_{c,s}^{\mathcal{P}}$. We need to show that these states are precisely the ones which are also filtered by the scheduler $\sigma^{\mathcal{T}}$, i.e. for which the action a should not be chosen according to the strategy.

Case 1 Let $s \in S_g \setminus S_{g \wedge (p_i \vee dl)}$. Then $s \models g$ but $s \not\models \bigvee_{p \in goodPred(a)} p \vee deadlock$. From $s \not\models \bigvee_{p \in goodPred(a)} p$ and Lemma 4.3.1, we obtain $(s, a) \notin \mathcal{L}(\mathcal{T})$ and consequently $a \notin goodActions(s)$.
From $s \not\models deadlock = \bigwedge_{a' \in A} \neg \eta_{a'}$ with enabled formulas

$$\eta_{a'} = \bigwedge_{j=0}^{n} g_j \wedge \bigvee_{p \in goodPred(a')} p,$$

where g_j are the guards of all the commands with label a, we get that there is at least one other action a' such that s satisfies its guard ($a' \in Act(s)$) and one of its good predicates. Therefore $goodActions(s) \neq \emptyset$.
Together this implies $a \notin supp(\sigma^{\mathcal{T}}(s)) = goodActions(s)$, or in other words $\sigma^{\mathcal{T}}(s)(a) = 0$.

Case 2 Now let $s \in S_{g \wedge (p_i \vee dl)}$, therefore $s \models g \wedge (p_1 \vee p_2 \vee \ldots \vee p_n \vee deadlock)$. Because $s \models S_g$ we know that $a \in Act(s)$. We split into two cases since $s \models (p_1 \vee \ldots \vee p_n)$ implies that $s \models \eta_a$ for some $a \in A$ and therefore entails $s \not\models deadlock$.
Case 2.a: $s \models p_1 \vee p_2 \vee \ldots \vee p_n$. Then $(s, a) \in \mathcal{L}(\mathcal{T})$ (Lemma 4.3.1). With $a \in Act(s)$ it follows that $a \in goodActions(s)$ and $\sigma^{\mathcal{T}}(s)(a) = \frac{1}{|goodActions(s)|}$.
Case 2.b: $s \models deadlock = \bigwedge_{a \in A} \eta_a$. Then for all actions $a' \in A$ there is no predicate $p \in goodPred(a')$ such that $s \models p$. By Lemma 4.3.1 $(s, a') \notin \mathcal{L}(\mathcal{T})$ for all $a' \in A$ and therefore $goodActions(s) = \emptyset$.
Since $a \in Act(s)$ it follows that $\sigma^{\mathcal{T}}(s)(a) = \frac{1}{|Act(s)|}$.

Therefore the scheduler $\sigma^{\mathcal{T}}$ achieves the same as the scheduler module by filtering all states $s \in S_g \setminus S_{g \wedge (p_i \vee dl)}$ (Case 1). For all states $s \in S_{g \wedge (p_i \vee dl)}$ the effect

is equal to the row normalisation of the transition probability matrix $\bar{P}$ (uniform distribution) (Case 2). The normalisation factor for the row associated with s is $N_s = |goodActions(s)|$ if Case 2.a applies, and $N_s = |Act(s)|$ if Case 2.b applies.
We obtain this as follows: $\mu_{c,s}^{\mathcal{P}_\mathcal{T}}(t)$ is a probability distribution over S, which means that $\sum_{t \in S} \mu_{c,s}^{\mathcal{P}_\mathcal{T}}(t) = 1$. That means the row of transition matrix $\bar{P}$ associated with state s sums up to the number of probability distributions defined for s. For an action a, s has a probability distribution $\mu_{c,s}$ exactly if $s \in S_{g \wedge (p_i \vee dl)}$ for the corresponding command c, i.e. one of the Cases 2.a/b applies to s. Therefore there are either $|goodActions(s)|$ many distributions or Case 2.b holds and $s \models deadlock$, which means that s satisfies the guard of all actions $a \in Act(s)$ and there are $|Act(s)|$ many distributions.

Finally we can show equality of the transition probability functions:

$$\begin{aligned} P^{\mathcal{P}_\mathcal{T}}(s,t) &= \frac{1}{N_s} \sum_{c \in C_{\sigma^\mathcal{T}}} \mu_{c,s}^{\mathcal{P}_\mathcal{T}}(t) \\ &= \sum_{c \in C} \mu_{c,s}^{\mathcal{P}}(t) \cdot \sigma^\mathcal{T}(s)(c) \\ &= \sum_{a \in Act(s)} P(s,a,t) \cdot \sigma^\mathcal{T}(s)(a) = P_{\sigma^\mathcal{T}}(s,t). \end{aligned}$$

□

4.5 Alternative Error Computation

Since we have proven that the DTMC obtained from the scheduled PRISM program is equivalent to the induced DTMC $\mathcal{M}_{\sigma^\mathcal{T}}$, it can alternatively be used to compute the relative error in the pruning step of the algorithm. In particular, we need to determine $\mathrm{Pr}^{\mathcal{M}_{\sigma^\mathcal{T}}}(\Diamond T)$ to compute the error.
The original error computation relied on a transformation of $\mathcal{T}$ to an explicit representation of $\sigma^\mathcal{T}$ which requires "prediction" of the label for every state-action pair in the model by traversing the tree (Section 3.2.3). The speed of this relies on the used decision tree implementation.
Alternatively, we can build $\mathcal{M}_{\sigma^\mathcal{T}}$ from the scheduled PRISM program. The speed of this relies on the optimization of model construction in the used model checker.
For the particular combination we used in our implementation we achieved speed-ups for large models by using the scheduled program (Chapter 5).

Chapter 5

Experiments

5.1 Overview

In this section we give details on our implementation of the algorithms presented in the previous chapters and discuss the choice of parameters we used for testing in Section 5.2. We evaluated on a diverse set of benchmarks briefly introduced in Section 5.3.1 and finally compare results in Section 5.3.2.

5.2 Implementation

All versions of the algorithm were implemented in the probabilistic model checker Storm [8]. For the decision tree implementation we used the open source C++ machine learning library waffles [11]. For construction and optimization of all linear programs we use the gurobi optimizer library [12].

Simulation

We implemented the simulation directly in Storm and used the C++ standard library for random number generation.

Pruning Paramters

The library waffles provides two pruning parameters for decision trees:

- `leafthreshold` : Whenever a node has less training instances than this number, the set is not divided any further and a leaf node is created. Default is 1.

- `maxLevels` : Controls the height of the tree. When the path from root to a node is `maxLevels` long, the data is not further divided, and a leaf is created. The default i 0, which indicates that there is no restriction on tree size.

The effect of both pruning parameters on tree size is illustrated by Figure 5.1 for two benchmarks. The plots were created by sampling evenly over the full range of sensible values. For `leafthreshold` (Figure 5.1a and 5.1d), we sampled between 1 and the maximum between the total of either good or bad training instances. For `maxLevels` (Figure 5.1b and 5.1e), we sample between 1 and the maximum tree size (obtained by using default values for both parameters). Since `maxLevels` controls the depth of the tree, which is usually not equal to the size, this creates a plateau at the beginning of the graph before the value is sufficiently high that nodes are actually pruned.
We can see that `maxLevels` has a much more drastic effect; it reduces the tree size

effectively in much smaller intervals than `leafthreshold`. Why this is undesirable is illustrated by Figures 5.1c and 5.1f which compare tree size to the relative representation error. For both benchmarks, a threshold exists where the error jumps exponentially for smaller tree sizes. This behaviour is also observed in all other benchmarks to be around 1% error.
Consequently we choose to perform binary search on `leafthreshold` rather than `maxLevels`, as it allows for much finer tuning of the tree size (and consequently the error) and require that the error is $< 1\%$. This is similar to the findings in [5] whose implementation uses a pruning parameter that is comparable to `leafthreshold`.
Since we perform binary search on `leafthreshold`, with an upper boundary of the maximum of good and bad training instances, the time for binary search depends largely on the number of instances learned.

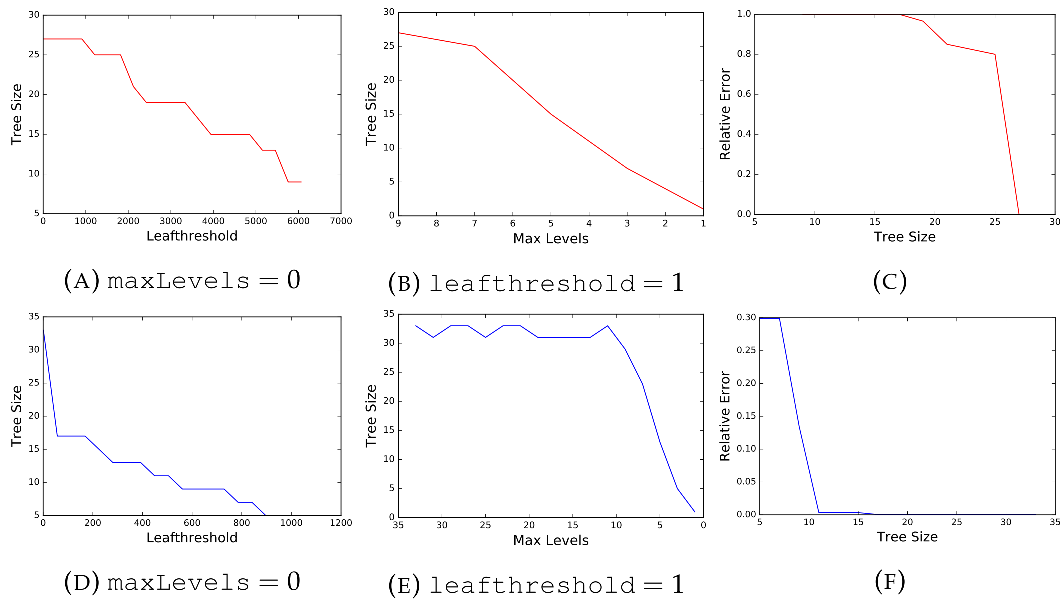
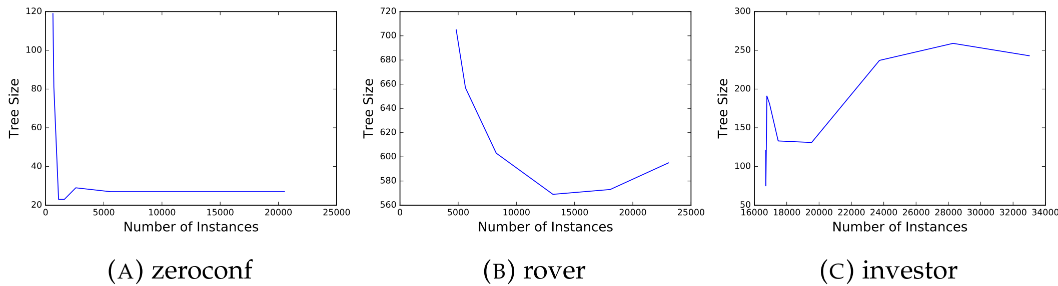


FIGURE 5.1: Effect of pruning parameters on tree size for benchmarks resource gathering (top row) and zeroconf (bottom row).

Linear Programs

The importance values computed by the linear programs are small (typically < 1). Learning state-action instances $Imp^{LP}(s)$ many times therefore would result in (almost) no instances being learned at all. Therefore we scale by a factor c such that about a number n many instances are learned.
The graph in Figure 5.2 shows the effect of the number of instances n that are learned on the tree size if we use the DTMC LP variant. In particular, it shows how the number of instances influences how large a tree is necessary to represent the strategy with a sufficiently small error $< 1\%$. The graphs for the benchmarks zeroconf (Figure 5.2a) and rover (Figure 5.2b) suggest that it suffices to learn a sufficiently large amount of instances to get good results. However, the graph for the investor benchmark (Figure 5.2c) shows that this does not seem to be strictly true. It suggests that sometimes learning more instances is worse than learning less. One possible explanation for this is that when too many instances are learned, enough of the less important decisions of the strategy are emphasized in the training algorithm to avoid being pruned.

For our experiments, we use the simple heuristic of training a large amount ($n \geq 20000$) of instances. In most cases, this is still much less than for simulation.



(A) zeroconf (B) rover (C) investor

FIGURE 5.2: Effect of number of instances learned on tree size for various benchmarks using DTMC LP variant.

5.3 Results

All experiments were run on a single Intel(R) Core(TM) i5-3337U CPU (1.8GHz) with 4 GB memory.

5.3.1 Benchmarks

This section gives a short description of the models and reachability objectives used for evaluation of the implementation. The benchmarks zeroconf and firewire are from the PRISM benchmark suite [15].

zeroconf. Implementation of the IPv4 network configuration protocol which prevents address conflicts between hosts. A host chooses randomly an address from a given set, and then sends four probes to all other hosts of the network. If a probe reaches a host which is already using the address, it responds, forcing the original host to start over. If four probes have been sent without the host receiving any replies, it commences to use the chosen address. The property of interest is the maximum probability that a host configures correctly.

firewire. The Tree Identify Protocol of the IEEE 1394 High Performance Serial Bus "Firewire". Whenever the a node is added to or removed from the bus, the (leader election) protocol is executed to choose a manager (root) node for the bus. We are interested in the maximum probability that a leader is chosen within a certain time frame.

investor describes a model for investment in future markets. The investor can make investments in shares of companies, whose value changes probabilistically. The case study is described in detail in [7]. The reachability objective is that the investor finishes a trade when his shares have a value greater than some threshold.

rover A model for the Mars rover task scheduling problem [13]. A Mars rover can perform four different types of tasks, which vary in success probability, and yield a certain amount of scientific value upon completion. The value, time and energy consumption of a task are determined by probability distributions. The property of interest is the maximal probability that a given amount of scientific value is achieved

within a certain time and energy limit.

resource gathering is one of the case studies found in [22]. It models a variant of grid world which is illustrated by Figure 5.3. An agent starts on a home position, moves on a finite size grid and has to collect resources (gold and gems) and bring them to the home location. On certain grid coordinates there is a chance that the agent gets attacked. Of interest is the maximum probability that a certain amount of gold and gems is collected within a certain number of steps.

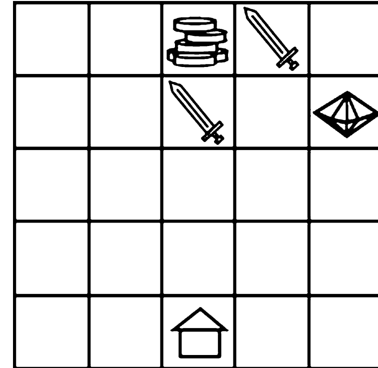


FIGURE 5.3: Grid world for the resource gathering problem. Picture from [22].

mechanic is the problem from Example 4.2.1. A mechanic has to maintain generators so they don't break down within a certain time frame (here: 12). He can maintain a generator only a limited amount of times (here: 4), and every time unit a generator is not maintained it has a chance to break down that scales linearly with passed time. We are interested in the maximum probability that the end of the time frame is reached without a generator breaking down.

5.3.2 Error And Tree Size

In Table 5.1 we compare the three alternatives for computing an importance heuristic on different benchmarks. In particular we are interested in small trees, since all variants achieve an error $< 1\%$. For simulation we list two values per benchmark, the best and the worst (with respect to tree size) of ten executions. In all cases, the tree representation is a marked improvement over the explicit representation which requires entries linear in the number of states.
Between simulation and the LP variants, there is no solution which is the best across all benchmarks, since every method has a benchmark for which it performs best.

			Simulation				LP		DTMC LP	
Benchmark	$\lvert S \rvert$	$Pr_{\max}$	$\lvert\mathcal{T}\rvert$	%Error	$\lvert\mathcal{T}\rvert$	%Error	$\lvert\mathcal{T}\rvert$	%Error	$\lvert\mathcal{T}\rvert$	%Error
zeroconf	88858	0.009	9	0.331	17	0.042	9	0.042	27	0.034
firewire	481136	1.000	11	0.000	21	0.000	5	0.000	13	0.000
investor	35893	0.958	291	0.820	521	0.775	339	0.925	217	0.423
resource	1004	0.810	27	0.000	27	0.000	39	0.000	27	0.000
rover	345488	0.672	533	0.960	613	0.778	665	0.973	595	0.774
mechanic	4941	0.250	27	0.000	27	0.000	27	0.000	27	0.000

TABLE 5.1: Comparison of tree size and relative error for different importance functions. In all cases, $\lvert\mathcal{T}\rvert$ refers to the number of nodes in the decision tree after binary search on leaftreshold, and *%Error* is the relative representation error of the represented scheduler compared to the optimal value $Pr_{\max}$. For simulation, we list the best and worst (in tree size) of ten executions of the algorithm, with the exception of firewire, which is a sample of three executions due to the high simulation time.

However, the tree sizes for a given benchmark all fall into the same order of magnitude. We can therefore say the methods are comparable in performance.

Aside from the importance function used to train the decision tree, another factor that influences tree size is the training algorithm itself, in particular the choice of what predicate to split on in a given node. Consider for example the tree in Figure 5.4a for the mechanic problem with global parameters $SCREWS = 1$ and $EOT = 30$, i.e. the mechanic has to keep the generators running for 30 time units, but he can only maintain once. The tree was built with the DTMC LP method (the other methods produce similar trees). By removing redundant splits on the time variable we obtain the tree in Figure 5.4b, which represents the same strategy in a much more compact format. This type of "cascading" branching can result in unnecessarily large tree sizes because it adds a leaf node on every level of the tree. It is therefore worthwhile to investigate other training algorithms and implementations.

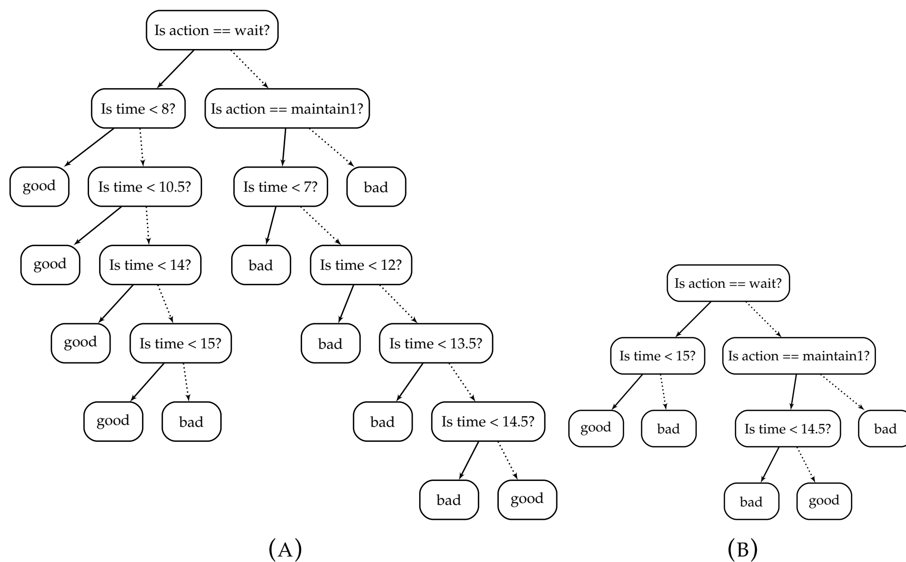


FIGURE 5.4: Two trees representing the same strategy for the mechanic problem with parameters $SCREWS = 1$ and $EOT = 30$.

5.3.3 Runtime

In Table 5.2 we compare the run times of the simulation and LP methods. We list the pure time for simulation and LP construction and optimization, as well as the total time for the algorithm. Simulation tends to take a very long time in the largest models firewire and rover since the average length of a run is higher. Also, since we have to run simulation potentially many times, the total required time to get the best result quickly adds up.
The LP formulation of MDPs also scales with the size of the model, since it requires one solution variable per state-action pair. Across the board the fastest method is the LP formulation on Markov chains; construction and optimization time are less sensitive to model size, since it formulated over the scheduled sub-model.

The bulk of the time after computing the importance function is occupied by binary search on the `leafthreshold` pruning parameter (Section 5.2). The number of iterations depends on the upper boundary for `leafthreshold` , which in turn

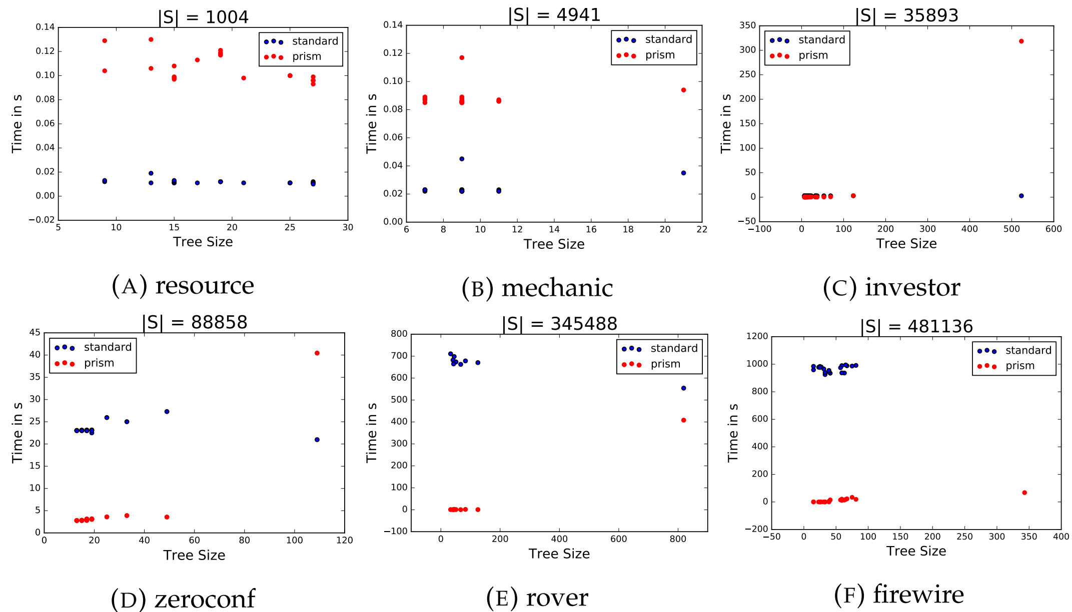
	Simulation		LP		DTMC LP	
Benchmark	Sim	Total	LP	Total	LP	Total
zeroconf	35.953s	72.921s	470.542s	551.046s	1.314s	107.974s
investor	278.759ss	1101.698ss	500.655ss	1119.033ss	3.408ss	439.054ss
firewire	40639.157s	57145.473s	2268.936s	3569.279s	7.468s	1784.663s
resource	1.353s	93.783s	0.013s	7.42s	0.01s	6.083s
rover	1815.201s	2682.929s	991.516s	1778.961s	4.617s	1667.501s
mechanic	6.388s	23.261s	0.116s	4.927s	0.116s	4.619s

TABLE 5.2: Comparison of run times for different benchmarks. *Sim* is the time spent on simulation of the scheduler on the MDP in one execution and *LP* is the time for construction and optimization of the corresponding linear program. *Total* refers to the total runtime of one execution of the algorithm.

depends on the number of instances learned. For simulation, this number tends to be much larger.

Runtime for Error Computation

In every step of the binary search, the tree is built ($< 1s$) and the error has to be computed. We have two alternative ways of computing the error. One is the standard method of converting the decision tree to an explicit representation and building the induced Markov chain. The other is to use a model checker to build the same Markov chain from a scheduled PRISM program. (We refer to the algorithms as "standard" and "prism" for brevity.) Run times for both algorithms are affected by model size and tree size to differing degrees. Therefore we compare the run time for different models separately over variable tree sizes in Figure 5.5.



(A) resource (B) mechanic (C) investor

(D) zeroconf (E) rover (F) firewire

FIGURE 5.5: Comparison of run times for the two alternative error computation algorithms by benchmark (model size). The graphs were generated by sampling uniformly over the range of applicable values for the leafthreshold parameter. The outliers in 5.5c, 5.5d and 5.5e correspond to unpruned trees.

For small models (mechanic, resource gathering) the standard algorithm wins by a small factor of $10^{-1}s$. However, for the larger models the prism algorithm pulls ahead by a much more substantial amount that grows with increasing model size. The investor model appears to be just at the midpoint, with prism already pulling slightly ahead by $1s$.
We can explain this as follows. For the standard computation, the tree is traversed for every state-action pair in the model. By contrast, the prism variant only traverses the tree once per action. The action set of most models is relatively small compared to the state set, except for very small models. Instead, the deciding factor is the construction of the scheduled model using a model checker (Storm). Since the scheduled model is a fragment of the original model, the size plays a lesser role than for the standard computation. Of course, we also benefit from any optimization on the part of the model checker.
The outliers in Figure 5.5c, 5.5d and 5.5c hint at a tendency for model construction time to go up with very large trees. This is likely due to the size of the scheduled PRISM program also increasing with tree size. However, these single sample points correspond to unpruned trees (i.e. the biggest possible trees) and the graphs were created by uniform sampling over the pruning parameter which is also calibrated by binary search. Therefore the search is concentrated on the area with a dense amount of samples, and unpruned trees are extremely unlikely to be considered.

Chapter 6

Conclusion

6.1 Discussion

In this thesis, we have examined scheduler representations with a focus on readability for human users. This topic is largely motivated by the idea that automatic generation of optimal strategies is only the first step; transferring the generated knowledge back to the user is just as important. Having a strategy is not enough if it cannot be understood properly by the people who should make use of it: A mountain climber has no use for a route description he or she cannot decipher in a timely manner and a mechanic is more likely to be confused by a service plan that lists all the times that he or she should *not* perform maintenance work.
This is what makes decision trees into a great medium for strategy representation: They follow a natural "check list" approach to decision making and present a manageable amount of information by focussing only on the most important parts of a strategy.
However, there are also limitations to the explanatory capabilities of decision trees. First, they are only really succinct if they are sufficiently small, which is why there is a focus on reducing their size through importance heuristics and pruning. Finding a good heuristic that emphasizes the right decisions is not entirely easy, as shown by our approach of using linear programs, which produce trees of similar size to simulation.
Second, the decision tree representations as they were presented here do not exactly capture a natural decision making process: It would be more natural to have a tree which poses questions only on the current situation (state) and gives as answer the action to be performed. The idea of action trees, "good predicates" and the PRISM scheduler representation try to capture this idea by expressing exactly in what situations a specific action is good.

On the technical side, it seems preferable to have a consistent method of generating results in contrast to a probabilistic method, especially since simulation time also scales with model size.
The scheduler module is not only useful for representative purposes, but also presents a method of generating an optimally scheduled PRISM program, which can be viewed as a symbolic description of a "critical subsystem" as in [9]. It can be plugged as is into model checkers and other algorithms for further processing, such as our alternative method of computing the relative representation error.

6.2 Outlook

Within the scope of this thesis, we have limited us to memoryless schedulers for single reachability objectives. For the classes of cost-bounded reachability and multi-objective properties, history dependent schedulers are also of interest [10]. It would be interesting to adapt existing approaches to represent these schedulers as well.

The suitability of the decision tree approach is in part due to the ability to learn and represent only the most important decisions of a strategy. Other machine learning tools have the same purpose of extracting prominent features from large amounts of data. As a preprocessing step that reduces the amount of information which needs to be represented, it is possible that they can enable other forms of readable scheduler representations.

The subject of scheduler representation is still under-explored, in part because the explanation of computer-generated strategies in a human readable format is not a trivial task. From the results of this thesis, we can conclude that it is a worthwhile problem to take on.

Bibliography

[1] Erika Ábrahám, Bernd Becker, Christian Dehnert, Nils Jansen, Joost-Pieter Katoen, and Ralf Wimmer. "Counterexample Generation for Discrete-Time Markov Models: An Introductory Survey". In: *SFM*. Vol. 8483. Lecture Notes in Computer Science. Springer, 2014, pp. 65–121.

[2] Husain Aljazzar and Stefan Leue. "Generation of Counterexamples for Model Checking of Markov Decision Processes". In: *QEST 2009, Sixth International Conference on the Quantitative Evaluation of Systems, Budapest, Hungary, 13-16 September 2009*. 2009, pp. 197–206.

[3] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008.

[4] Dimitri P. Bertsekas and John N. Tsitsiklis. "An Analysis of Stochastic Shortest Path Problems". In: *Math. Oper. Res.* 16.3 (1991), pp. 580–595.

[5] Tomás Brázdil, Krishnendu Chatterjee, Martin Chmelik, Andreas Fellner, and Jan Kretínský. "Counterexample Explanation by Learning Small Strategies in Markov Decision Processes". In: Lecture Notes in Computer Science 9206 (2015), pp. 158–177.

[6] Tomás Brázdil, Krishnendu Chatterjee, Jan Kretínský, and Viktor Toman. "Strategy Representation by Decision Trees in Reactive Synthesis". In: Lecture Notes in Computer Science 10805 (2018), pp. 385–407.

[7] *Case Study: Future Markets Investor*. 2018. URL: http://www.prismmodelchecker.org/casestudies/investor.php.

[8] Christian Dehnert, Sebastian Junges, Joost-Pieter Katoen, and Matthias Volk. "A Storm is Coming: A Modern Probabilistic Model Checker". In: Lecture Notes in Computer Science 10427 (2017), pp. 592–600.

[9] Christian Dehnert, Nils Jansen, Ralf Wimmer, Erika Ábrahám, and Joost-Pieter Katoen. "Fast Debugging of PRISM Models". In: *Automated Technology for Verification and Analysis - 12th International Symposium, ATVA 2014, Sydney, NSW, Australia, November 3-7, 2014, Proceedings*. 2014, pp. 146–162.

[10] Vojtech Forejt, Marta Z. Kwiatkowska, Gethin Norman, David Parker, and Hongyang Qu. "Quantitative Multi-objective Verification for Probabilistic Systems". In: *TACAS*. Vol. 6605. Lecture Notes in Computer Science. Springer, 2011, pp. 112–127.

[11] Michael Gashler. "*Waffles*: A Machine Learning Toolkit". In: *Journal of Machine Learning Research* 12 (2011), pp. 2383–2387.

[12] Inc. Gurobi Optimization. *Gurobi Optimizer Reference Manual*. 2016. URL: http://www.gurobi.com.

[13] Arnd Hartmanns, Sebastian Junges, Joost-Pieter Katoen, and Tim Quatmann. "Multi-cost Bounded Reachability in MDP". In: *TACAS (2)*. Vol. 10806. Lecture Notes in Computer Science. Springer, 2018, pp. 320–339.

[14] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. "PRISM 4.0: Verification of Probabilistic Real-Time Systems". In: Lecture Notes in Computer Science 6806 (2011), pp. 585–591.

[15] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. "The PRISM Benchmark Suite". In: *QEST*. IEEE Computer Society, 2012, pp. 203–204.

[16] Florian Leitner-Fischer and Stefan Leue. "Probabilistic fault tree synthesis using causality computation". In: *IJCCBS* 4.2 (2013), pp. 119–143.

[17] Mausam and Andrey Kolobov. *Planning with Markov Decision Processes: An AI Perspective*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2012.

[18] Andrew S. Miner and David Parker. "Symbolic Representations and Analysis of Large Probabilistic Systems". In: *Validation of Stochastic Systems - A Guide to Current Research*. 2004, pp. 296–338.

[19] J. R. Quinlan. "Induction of Decision Trees". In: *Machine Learning* 1.1 (1986), pp. 81–106.

[20] *The PRISM Language - Semantics*. 2018. URL: http://www.prismmodelchecker.org/doc/semantics.pdf.

[21] *The PRISM Language Manual*. 2018. URL: http://www.prismmodelchecker.org/manual/ThePRISMLanguage/Introduction.

[22] Peter Vamplew, Richard Dazeley, Adam Berry, Rustam Issabekov, and Evan Dekker. "Empirical evaluation methods for multiobjective reinforcement learning algorithms". In: *Machine Learning* 84.1-2 (2011), pp. 51–80.

[23] Ralf Wimmer, Nils Jansen, Erika Ábrahám, Bernd Becker, and Joost-Pieter Katoen. "Minimal Critical Subsystems for Discrete-Time Markov Models". In: *TACAS*. Vol. 7214. Lecture Notes in Computer Science. Springer, 2012, pp. 299–314.

[24] Ralf Wimmer, Bettina Braitling, Bernd Becker, Ernst Moritz Hahn, Pepijn Crouzen, Holger Hermanns, Abhishek Dhama, and Oliver E. Theel. "Symblicit Calculation of Long-Run Averages for Concurrent Probabilistic Systems". In: *QEST 2010, Seventh International Conference on the Quantitative Evaluation of Systems, Williamsburg, Virginia, USA, 15-18 September 2010*. 2010, pp. 27–36.