Synthesizing Efficient Controllers[*]

Christian von Essen[1] and Barbara Jobstmann[2]

[1] UJF/Verimag, Grenoble, France
[2] CNRS/Verimag, Grenoble, France

Abstract. In many situations, we are interested in controllers that implement a good trade-off between conflicting objectives, e.g., the speed of a car versus its fuel consumption, or the transmission rate of a wireless device versus its energy consumption. In both cases, we aim for a system that efficiently uses its resources. In this paper we show how to automatically construct efficient controllers. We provide a specification framework for controllers in probabilistic environments and show how to synthesize implementations from them. We achieve this by reduction to Markov Decision Processes with a novel objective function. We compute optimal strategies for them using three different solutions (linear programming, fractional linear programming, policy iteration). We implemented and compared the three algorithms and integrated the fastest algorithm into the model checker PRISM.

1 Introduction

Synthesis aims to automatically generate a system from a specification. We focus on synthesizing reactive systems [17] from specifications given in temporal logics [12]. In this setting, specifications are usually given in a qualitative sense, i.e., they classify a system either as good (meaning the system satisfies the specification) or as bad (meaning the system violates the specification). Quantitative specifications assign to each system a value that provides additional information about the system. Traditionally, quantitative techniques are used to analyze properties like response time, throughput, or reliability (cf. [7, 9, 1, 10]).

Recently, quantitative reasoning has been used to state preference relations between systems satisfying the same qualitative specification [2]. E.g., we can compare systems with respect to robustness, i.e., how reasonable they behave under unexpected behaviors of their environments [3]. A preference relation between systems is particularly useful in synthesis, because it allows the user to guide the synthesizer and ask for "the best" system. In many settings a better system comes with a higher price. E.g., consider an assembly line that can be operated in several modes that indicate the speed of the line, i.e., the number of units produced per step. We would prefer a controller that produces as many units as possible. However, running the line in a faster mode increases the power consumption and the probability to fail, resulting in higher repair costs. We are

[*] This work was supported by the EU project ASCENS, FP7-257414.

interested in an "efficient" controller, i.e., a system that minimizes the power and repair costs per produced unit. The efficiency of a system is a natural question to ask; it has also been observed by others, e.g, Yue et al. [16] used simulation to analyze energy-efficiency in a MAC (Media Access Control) Protocol.

In this paper we show how to automatically synthesize a system that has an efficient average-case behavior in a given environment. We define efficiency as ratio between a given *cost* model and a given *reward* model. To further motivate this choice, consider the following example: assume we want to implement an automatic gear-shifting unit (ACTS) that optimizes its behavior for a given driver profile. The goal of our implementation is to optimize the fuel consumption per kilometer (l/km), a commonly used unit to advertise efficiency. In order to be most efficient, our system has to maximize the speed (given in km/h) while minimizing the fuel consumption (measured in liters per hour, i.e., l/h) for the given driver profile. If we take the ratio between the fuel consumption (the "costs") and the speed (the "reward"), we obtain l/km, the desired measure.

Given an efficiency measure, we ask for a system with an optimal average-case behavior. The average-case behavior with respect to a quantitative specification is the expected value of the specification over all possible behaviors of the systems in a given probabilistic environment [5]. We describe the probabilistic environment using Markov Decision Processes (MDPs), which is a more general model than the one considered in [5]. It allows us to describe environments that react to the behavior of the system (like the driver profile).

In the following we summarize our contributions[3] and outline the paper.

1. We present a framework to automatically construct a system with an efficient average-case behavior with respect to a reward and a cost model in a probabilistic environment. To the best of our knowledge, this is the first approach that allows synthesizing efficient systems automatically. After giving the necessary preliminaries in Section 2, we introduce our framework using a simple example in Section 3. In our framework, finding an optimal system corresponds to finding an optimal strategy in an MDP with ratio objective.
2. We introduce and study MDPs with ratio objectives (in Section 4). We present several algorithms to compute optimal strategies in MDPs under ratio objectives. All algorithms are based on decomposing the MDP into end-components [7]. The algorithms differ in the way they compute an optimal strategy for a single end-component. One algorithm uses fractional linear programming. The second one, a simple adaption of an algorithm presented in [7], is based on a reduction to linear programming. The third algorithm is based on policy iteration and a sequence of reductions to MDPs with long-run average-reward objective. This novel algorithm based on policy iteration is particularly interesting, since it can readily be applied to symbolically encoded MDPs and to large structures [15]. In Section 5, we compare our

[3] We presented preliminary results for ergodic MDPs in a workshop [14]. We refer to it to provide omitted details. The current paper presents, in addition, (i) the solution for the general case, (ii) a novel algorithm based on policy iteration, and (iii) an implementation.

framework based on MDPs with ratio objectives to related work and discuss the need for separating the cost and reward model.

3. We have implemented all algorithms in a stand-alone tool and compare them on our examples (see Section 6). In order to increase the scope of our approach, we also integrated the best-performing algorithm into the explicit-state version of PRISM [10], a well-known probabilistic model checker.

2 Preliminaries

Words, Quantitative Languages, and Specifications. Given a finite alphabet Σ, a *word* $w = w_0 w_1 \dots$ is a finite or infinite sequence of *letters* in Σ. We use w_i to denote the $(i{+}1)$-th letter. The empty word is denoted by ϵ. We use Σ^* (Σ^ω) to denote the set of finite (infinite) words. Given two words $w \in \Sigma^*$ and $v \in \Sigma^* \cup \Sigma^\omega$, we write wv for their concatenation. A *(quantitative) language* [4] is a function $\psi : \Sigma^\omega \to \mathbb{R}^+ \cup \{\infty\}$ associating to each infinite word a value from the extended non-negative reals. A *qualitative language* is a special case mapping words to 1 or 0. We use qualitative and quantitative languages as *specifications* to describe the desired behavior of a system.

Labeled Transition Systems, Quantitative Automata and the Ratio Objective. A *Labeled transition systems (LTS)* is a tuple $\mathcal{A} = (Q, q_0, \Sigma, \delta)$ where Q is a finite or infinite set of states, $q_0 \in Q$ is the start state, Σ is a finite alphabet and $\delta : Q \times \Sigma \to Q$ is the *transition function*. We call an LTS finite if and only if Q is finite. We define δ^* as the usual extension of δ to finite words. The run of $\mathcal{A}$ on an infinite word $w = w_0 w_1 w_2 \dots$ is the sequence of tuples $(q_0, w_0), (q_1, w_1), (q_2, w_2) \dots$ where $q_{i+1} = \delta(q_i, w_i)$. Given finite LTS $\mathcal{A}$, a *cost or reward function* $c, r : Q \times \Sigma \to \mathbb{N}$ maps every transition of $\mathcal{A}$ to a natural number. We call a finite LTS with one or more cost/reward functions a *quantitative automaton.*

An *objective function* maps runs of a quantitative automaton to elements of $\mathbb{R}^+ \cup \{\infty\}$. Given a quantitative automaton $\mathcal{A} = (Q, q_0, \Sigma, \delta)$ with cost function c and reward function r, we define the *ratio objective function* [3] for each run $\rho = (q_0, w_0), (q_1, w_1), (q_2, w_2) \dots$ of $\mathcal{A}$ as

$$\mathcal{R}^{\mathcal{A}}_{\frac{c}{r}}(\rho) := \lim_{m\to\infty} \liminf_{l\to\infty} \frac{\sum_{i=m}^{l} c(q_i, w_i)}{1 + \sum_{i=m}^{l} r(q_i, w_i)}. \tag{1}$$

We write $\mathcal{R}(\rho)$, if $\mathcal{A}$, c, and r are clear from the context. Intuitively, $\mathcal{R}(\rho)$ is the long-run ratio between the costs and rewards accumulated along a run. The first (left-most) limit allows us to ignore a finite prefix of the run, which ensures that we only consider the long-run behavior. The 1 in the denominator avoids division by 0 if the accumulated rewards are 0 and has no effect otherwise. We need the limit inferior here because the sequence of ratios might not converge. E.g., a run $q^1 p^2 q^4 p^8 \dots$ with $c(q) = 0$ and $r(q) = c(p) = r(p) = 1$. Its value alternates between 1/6 and 1/3 and does not converge. The limit inferior of this run is 1/3. The ratio objective generalizes the long-run average objective (also known as mean-payoff objective, cf. [18]).

(Finite-)State Systems. A *(finite-)state system* is a tuple $\mathcal{S} = (S, s_0, L, A, \delta, \tau)$, where (S, s_0, L, δ) is a (finite) LTS, A is an *output alphabet*, and $\tau : S \to A$ is an *output function* mapping states to letters from A. The alphabet L is called the *input alphabet* of $\mathcal{S}$. We use $\mathcal{O}_\mathcal{S}$ to denote the function mapping input words $w \in L^\omega$ to the joint input/output word by applying τ to the run of $\mathcal{S}$ on w.

Given a quantitative language ψ over $(L \times A)^\omega$, the *value of the system $\mathcal{S}$ for a given input word $w \in L^\omega$ under ψ* is the value of the joint input/output word under ψ. We obtain the value by composing the functions ψ and $\mathcal{O}_\mathcal{S}$ and apply the composed function to w, i.e., $(\psi \cdot \mathcal{O}_\mathcal{S})(w)$.

Given a probability distribution μ over the input words L^ω, the *(average-case) value of a system $\mathcal{S}$ with respect to a specification ψ and the probability distribution μ* is the expected value $\mathbb{E}_\mu[\psi \cdot \mathcal{O}_\mathcal{S}]$. A system $\mathcal{S}$ is *optimal* with respect to ψ and μ if for every system $\mathcal{S}'$ the value of $\mathcal{S}'$ is smaller than or equal to the value of $\mathcal{S}$.

Markov Chains and Markov Decision Processes. Let $\mathcal{D}(S) := \{p : S \to [0,1] \mid \sum_{s \in S} p(s) = 1\}$ be the *set of probability distributions* over a finite set S. A *Markov decision process (MDP)* is a tuple $\mathcal{M} = (S, s_0, A, \tilde{A}, p)$, where S is a finite set of *states*, $s_0 \in S$ is an *initial state*, A is a finite set of *actions*, $\tilde{A} : S \to 2^A$ is the *enabled action function* defining for each state s the set of enabled actions in s, and $p : S \times A \to \mathcal{D}(S)$ is a probabilistic *transition function*. For technical convenience we assume that every state has at least one enabled action. If $|\tilde{A}(s)| = 1$ for all states $s \in S$, then $\mathcal{M}$ is called a *Markov chain (MC)*. In this case, we omit A and $\tilde{A}$ from the definition of $\mathcal{M}$.

An *L-labeled MDP* is a tuple $\mathcal{M} = (S, s_0, A, \tilde{A}, p, \lambda)$, where $(S, s_0, A, \tilde{A}, p)$ is an MDP and $\lambda : S \to L$ is a labeling function such that $\mathcal{M}$ is deterministic with respect to λ, i.e, $\forall s, a, s', s''$ if $p(s,a)(s') > 0$, $p(s,a)(s'') > 0$, and $s' \neq s''$, then $\lambda(s') \neq \lambda(s'')$. In Section 3, we use L-labeled MDPs to represent probabilistic environments that react to the actions chosen by the system. Since the environment does not know which action a system might choose, we require that in every state of an L-labeled MDP all actions are enabled, i.e., $\forall s \in S : \tilde{A}(s) = A$.

Sample Runs, Strategies, and Objective Functions. A *(sample) run* ρ of $\mathcal{M}$ is an infinite sequence of tuples $(s_0, a_0)(s_1, a_1) \cdots \in (S \times A)^\omega$ of states and actions such that for all $i \geq 0$, (i) $a_i \in \tilde{A}(s_i)$ and (ii) $p(s_i, a_i)(s_{i+1}) > 0$. We write Ω for the set of all runs, and Ω_s for the set of runs starting at state s.

A *strategy (or policy)* is a function $d : (S \times A)^* S \to \mathcal{D}(A)$ that assigns a probability distribution to all finite sequences in $(S \times A)^* S$. A strategy must refer only to enabled actions. An MDP together with a state s and a strategy d defines a probability space $\mathcal{P}^d_{\mathcal{M},s}$ that uniquely defines the probability of every measurable set of runs starting in s.

Given a measurable function $f : \Omega \to \mathbb{R}^+ \cup \{\infty\}$ that maps runs of $\mathcal{M}$ to values in $\mathbb{R}^+ \cup \{\infty\}$, we use $\mathbb{E}^d_{\mathcal{M},s}[f]$ to denote the expected value of f under the probability measure of $\mathcal{P}^d_{\mathcal{M},s}$ and call it the *value of s under strategy d wrt f*. Given an MDP $\mathcal{M}$ and a state s, a strategy d is called *optimal for objective f and state s* if $\mathbb{E}^d_{\mathcal{M},s}[f] = \min_{d'} \mathbb{E}^{d'}_{\mathcal{M},s}[f]$, where d' ranges over all possible strategies. Given an optimal strategy d for function f and state s, $\mathbb{E}^d_{\mathcal{M},s}[f]$ is called the *value of state s wrt f*. The *value of $\mathcal{M}$ wrt f* is the value of its initial state. If f is the ratio objective from Eqn. 1, then we call $\mathcal{M}$ with f a Ratio-MDP. If the domain of c is $\{1\}$, then the objective is equal to the classical *average-reward or mean-payoff objective* ([13]), and we call $\mathcal{M}$ with f an Average-MDP.

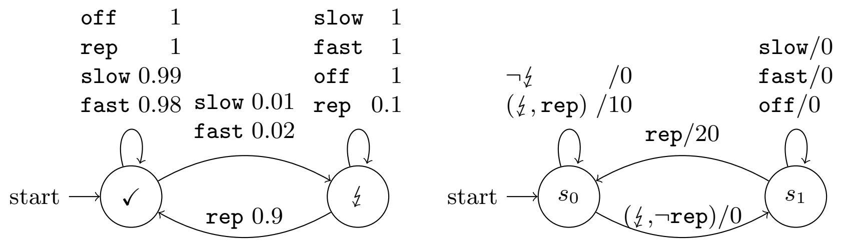
3 Specifying Efficient Controllers

We use a simple example to introduce our quantitative synthesis framework. Each part of the example is used to highlight one part of the framework.

Example. Assume we aim to synthesize an efficient controller for a reactor cooling system that controls the activation and maintenance of three pumps. The task of the pumps is to keep the reactor cool. If no pump is running, then the reactor cannot work. If all pumps are working, then the reactor can work at maximum effectivness. A pump that is working can break down, and a pump that is broken must be repaired. If a broken pump is not repaired immediately, then the cost of repairing increases. If two or more pumps are repaired at the same time, then there is a discount on repairing them. We aim for a system that is most efficient, i.e., it minimizes the maintenance costs per water-flow unit.

Modeling the Environment. We model the environment (i.e., the pumps) and its reaction to actions taken by the system using labeled MDPs. The model of a single pump is shown in Figure 1(a). A pump has two states: broken ($\lightning$) and ok ($\checkmark$). In each of these states, the system can either turn a pump on to a slow mode with action `slow`, turn it on to a fast mode with action `fast`, switch it off with action `off`, or repair it with action `rep`. The failure of a pump is controlled by the environment. We assume a failure probability of 1% when the pump is running slowly and 2% when the pump is running fast. If it is turned off, then a failure cannot happen. Transitions in Figure 1(a) are labeled with actions and probabilities, e.g., the transition from state $\checkmark$ to $\checkmark$ labeled "`slow` 0.99" means that we go from state $\checkmark$ with action `slow` with probability 0.99 to state $\checkmark$. Note that the labels of the states ($\checkmark$ and $\lightning$) of this MDP correspond to decisions the environment can make. The actions of the MDP are the decisions the system can use to control the environment. The specification for n pumps is the synchronous product of n copies of the model in Figure 1(a), i.e., the state space of the resulting MDP is the Cartesian product, and the transition probabilities are the product of the probabilities; e.g., for two pumps, the probability to move from ($\checkmark$, $\checkmark$) to ($\checkmark$, $\checkmark$) on action (`slow`, `slow`) is 0.99^2.

System. Our systems are state machines that read the state of the environment and return actions to perform. The action affects the probability of the next state of the pump. A system $\mathcal{S}$ also represents a strategy $d_{\mathcal{S}}$ mapping finite sequences of environment states to actions. For example, a system that repairs the pump when it is broken would map the sequence $\checkmark\ \checkmark\ \lightning$ to action `rep`. The strategy $d_{\mathcal{S}}$ together with the description of the environment (an MDP) induces a probability space over sequences of pairs of states and actions. E.g., if the system chooses action `slow` as long as the pump is $\checkmark$ and action `rep` as soon as



(a) A pump (b) Quantifying a pump

Fig. 1. Environment model and quantitative specification of the pumps example

the pump breaks down, then the sequence (✓, `slow`) (✓, `slow`) (↯, `rep`), which corresponds to transitions $\checkmark \xrightarrow{\texttt{slow}} \checkmark$ and $\checkmark \xrightarrow{\texttt{slow}} ↯$, has probability $0.99 \cdot 0.01$.

Specification. We use a quantitative specification, given by an automaton with a reward and a cost function, to evaluate a system with respect to a desired property. The automaton reads words over the joint input/output alphabet and assigns a value to them. For example, the specification automaton for the pump controlling system reads pairs consisting of (i) a state of a pump (input of the system) and (ii) an action (output of the system). We obtain this automaton by composing automata with a single cost function in various ways. In our example, we use for each pump two automata with a single cost function to express the repair costs and the water flow due to this pump. The automaton for the repair costs is shown in Figure 1(b). It assigns repair costs of 10 for repairing a broken pump immediately and costs 20 for a delayed repair. If we sum the numbers that the automaton outputs, we obtain the repair costs of a run. For example, sequence (✓, `slow`) (↯, `rep`) (↯, `rep`) has cost $0 + 10 + 10 = 20$. The water flow depends on the speed of the pump. The automaton describing the water flow assigns value 2 if a pump is running on slow speed, 4 if it is running on fast speed, and 0 if the pump is turned off or broken.

We extend the specification to multiple pumps by building the synchronous product of copies of the automata described above and compose the cost and reward functions in the following ways: we sum the rewards for the water flow and we take the maximum of repair costs of different pumps to express a discount for simultaneous repairs of more than one pump. The final specification automaton is the product of the water flow automaton and the repair cost automaton with (i) the repair cost as cost function and (ii) the water flow as reward function. We prefer behaviors with a small ratio between the accumulated repair costs and the accumulated water flow rewards and so we take Eqn. 1 as objective function $\mathcal{R}$.

We lift the specification from a single behavior to a system $\mathcal{S}$ (a set of behaviors) by computing the expected value of $\mathcal{R}$ under the probability distribution given by the environment model (an MDP) and the strategy $d_{\mathcal{S}}$ representing the system. This corresponds to averaging over all behaviors weighted by their probability. Our goal is to find a system that minimizes the maintenance costs and maximizes the water flow, i.e., a system that minimizes this expected value.

In the current specification a system that keeps all the pumps turned off has the (smallest possible) value zero, because pumps that are turned off do not break down and there is no need to repair them. Therefore, we require that at least one of the pumps is working. We can specify this requirement by using a qualitative specification described by a safety[4] automaton. This safety requirement can then be ensured by adapting the cost functions of the ratio objective [5, 14]. For simplicity, we say here that any action in which not at least one pump is working has an additional cost of 10.

Synthesis using MDPs. We build an MDP from the environment descriptions and the quantitative specifications by taking their product. The intuition is to run the environment MDP and the specification automaton in parallel. We get states of the form (s, q), where s is a state in the environment MDP and q is a state of the automaton. The set of actions enabled in this state is equal to the set of actions available in state s. The probability of moving from state (s, q) to state (s', q') when choosing action a is zero if there is no transition in the automaton from q to q' with input (s, a). Otherwise, the probability is equal to the probability of moving from s to s' in the MDP when choosing a.

The cost of an action is derived from the corresponding costs in the automaton. E.g., the probability of moving from $((\checkmark, \checkmark), (s_0, s_0))$ to $((\lightning, \lightning), (s_0, s_0))$ when choosing action $(\texttt{slow}, \texttt{slow})$ is 0.01^2, while the probability of moving to $((\lightning, \lightning), (s_1, s_1))$ is 0 because we cannot move from s_0 to s_1 with this action. The repair costs of choosing action $(\texttt{slow}, \texttt{slow})$ in state $((\checkmark, \checkmark), (s_0, s_0))$ is 0, while the water flow is 4 (2 for each pump).

The product of an MDP and an automaton with two cost functions is a Ratio-MDP. As we will show in Lemma 1, there is always a pure and memoryless optimal strategy for a Ratio-MDP. Such a strategy d corresponds to the following finite-state system with optimal behavior. The states of the system are the states of the MDP. The output function for state s is the action chosen by the strategy d for state s, i.e., $\tau(s) = d(s)$. The transition function of the system refers to the states of the MDP (or more precisely to their labeling function). The system moves from state s to s' with input label l, if s' is (labeled) l and the probability to reach state s' from state s under the action $d(s)$ given by the strategy d is strictly positive. In this way, the environment determines the next state of the system. The expected value of the system is equal to the expected value of the strategy it was constructed from.

Theorem 1. *Given an MDP $\mathcal{M}$ describing an environment and an automaton with ratio objective describing a quantitative specification, we can automatically synthesize an optimal finite-state system.*

[4] Our approach can also handle liveness specifications resulting in a Ratio-MDP with parity objective, which is then reduced to solving a sequence of MDP with mean-payoff parity objectives [5].

4 Solving MDPs with Ratio-Objectives

In this section we first recall well-known notions for MDPs and their strategies, then we prove some useful properties of Ratio-MDPs. Finally, we present three algorithms to compute the optimal value and an optimal strategy of a Ratio-MDP. We assume that an optimal strategy minimizes the ratio objective, the algorithms for maximizing the objective are analogous.

End-components [6, 7]. Given an MDP $\mathcal{M} = (S, s_0, A, \tilde{A}, p)$, a set $C \subseteq S$ of states is an *end-component* if (i) C is p-closed (i.e., for all $s \in C$ exists $a \in A(s)$ such that $\forall s' \in S \setminus C : p(s,a)(s') = 0$) and (ii) the MDP obtained by restricting $\mathcal{M}$ to the states in C is strongly connected. An end-component is maximal, if it is not included in a strictly larger end-component.

Classification of MDPs and Strategies. Given an Markov chain $\mathcal{M} = (S, s_0, p)$, a state $s \in S$ is called *recurrent*[5] if the expected number of visits to s in the random walk starting from s_0 is infinite; otherwise s is called *transient.* A minimal set of recurrent states that is closed under p is called *recurrence class.* $\mathcal{M}$ is *unichain* if it consists of a single recurrence class and a (possibly empty) set of transient states; otherwise, $\mathcal{M}$ is called *multichain.*

A probability distribution π over S is a *stationary distribution* if all its entries satisfy $\pi(s) = \sum_{s' \in S} \pi(s')p(s,s')$. If $\mathcal{M}$ is unichain, then $\mathcal{M}$ has a unique stationary distribution. A strategy d is *pure* if for all sequences $w \in (S \times A)^*$ and for all states $s \in S$, there is an action $a \in A$ such that $d(ws)(a) = 1$. A *memoryless* strategy is independent of the history of the run, i.e., for all $w, w' \in (S \times A)^*$ and for all $s \in S$, $d(ws) = d(w's)$ holds. If a strategy is pure and memoryless, we represent it by a function $d : S \to A$. An MDP $\mathcal{M} = (S, s_0, A, \tilde{A}, p)$ together with a pure and memoryless strategy $d : S \to A$ defines an MC $\mathcal{M}^d = (S, s_0, A, \tilde{A}_d, p)$, in which only the actions prescribed in the strategy d are enabled, i.e., $\tilde{A}_d(s) = \{d(s)\}$. We call a pure and memoryless strategy d *unichain* (*multichain*) if the MC $\mathcal{M}^d$ is unichain (multichain, respectively). If $\mathcal{M}$ is associated with a cost function c, then we denote by c_d the corresponding cost-vector function[6], i.e., $c_d(s) = c(s, d(s))$ for all $s \in S$.

In the following, we discuss properties of Ratio-MDPs that are necessary for the correctness of the algorithms. Lemma 1 tells us that we need to consider only pure and memoryless strategies to compute the optimal value (see [14] for the proof). Lemma 2 strengthens this result to unichain strategies for MDPs that have a single end-component.

Lemma 1. *[14] Ratio-MDPs have optimal pure and memoryless strategies.*

Lemma 2. *Given a Ratio-MDP $\mathcal{M} = (S, s_0, A, \tilde{A}, p)$ such that S is an end-component of $\mathcal{M}$ and let d be a pure and memoryless strategy with value λ, then there exists a unichain strategy d' with value $\lambda' \leq \lambda$.*

[5] We do not distinguish null and positive recurrent states because we only consider finite MCs.

[6] We use vector and function notation interchangeably.It is well-known that every finite function can be represented by a vector and vice versa, given a total order on the domain elements.

Proof. This follows from prefix-independence of the ratio objective and the fact that for every pair of states s and s' in an end-component, there exists a strategy such that s can reach s' with probability 1. This allows us to construct from an arbitrary pure and memoryless strategy a unichain strategy with the same or a better value: d' fixes the recurrent class C with the minimal value induced by d; for states outside of C, d' plays a strategy to reach C with probability 1.

Next, we show how to compute the ratio value of an MDP with respect to a unichain strategy. Since the ratio objective is prefix-independent, the value of a state depends only on the rewards obtained in the (single) recurrence class induced by the unichain strategy. All states in a recurrence class have the same value, because they can reach each other with probability 1. Recall that the value of a state is the expected payoff over all runs starting in this state, and that the expected payoff is the Lebesgue integral over all runs. Every recurrence class has a stationary distribution π, in which the s^{th}-entry $\pi(s)$ corresponds to the expected fraction of time spent in state s. We call a run *well-behaved* if, for each state s, the number of visits to the state s up to position n of the run divided by n converges with $n \to \infty$ to $\pi(s)$. The set of well-behaved runs has probability 1 (see [14] for more details). Every measurable set that is disjoint from the set of well-behaved runs has probability 0 and does not contribute to the value of a state. All *well-behaved* runs have the same value, which corresponds to the expected average reward with respect to the cost function divided by the expected average reward with respect to the reward function (if the corner cases of value zero and infinity are neglected). A classical result for MDPs with a cost function c and a unichain strategy d says that the expected average reward, i.e., $\mathbb{E}_d^{\mathcal{M}}[\lim_{n\to\infty} \frac{1}{n} \sum_{i=0}^{n-1} c(s_n)]$, is equal to $\pi \cdot c_d$, where π is the stationary distribution under strategy d. This gives the following lemma.

Lemma 3. *For a Ratio-MDP $\mathcal{M}$ and a unichain strategy d, let π be the stationary distribution induced by d and let c_d and r_d be the reward vectors under strategy d of the cost and the reward function, respectively, then we have*

$$\mathbb{E}_{\mathcal{M}}^{d}[\mathcal{R}] = \lim_{l\to\infty} \frac{\pi \cdot c_d}{1/l + \pi \cdot r_d}$$

The limit in Lemma 3 takes care of the case in which $\pi \cdot r_d = 0$. Lemma 4 addresses this and another corner case.

Lemma 4. *For every Ratio-MDP $\mathcal{M} = (S, s_0, A, \tilde{A}, p)$ such that S is an end-component of $\mathcal{M}$, we can check efficiently if the value of $\mathcal{M}$ is zero or infinity and construct corresponding strategies.*

Proof. $\mathcal{M}$ has value zero if there exists a strategy such that the expected average reward w.r.t. the cost function c is zero. We check this by removing all actions from states in $\mathcal{M}$ that have $c > 0$ and then recursively removing all actions that lead to a state without an enabled action. If the resulting MDP $\mathcal{M}'$ is non-empty, then there is a strategy with value 0 for the original end-component. It can be computed by building a strategy that moves to and stays in $\mathcal{M}'$.

$\mathcal{M}$ has value infinity iff (i) for every strategy the expected average reward w.r.t. cost function c is not zero, i.e., $\mathcal{M}$ has not value zero, and (ii) for all strategies the expected average reward w.r.t. the reward function r is zero. This can only be the case if for all actions in the end-component the value of cost function r is zero. In this case, any arbitrary strategy will give value infinity.

Since the ratio objective is prefix-independent and every run in a finite MDP reaches and stays in an end-component with probability 1, regardless of the strategy, we can compute the optimal value and a corresponding strategy for an arbitrary Ratio-MDP by decomposing it into end-components.

Lemma 5. *Given a Ratio-MDP $\mathcal{M}$ and an optimal pure and memoryless strategy d_i for every maximal end-component C_i in $\mathcal{M}$, we can compute the optimal value and construct an optimal strategy for $\mathcal{M}$.*

Proof. Let λ_i be the value obtained with d_i in the Ratio-MDP induced by C_i. Wlog we assume that every action is enabled in exactly one state. Let $\overline{\mathcal{M}}$ be the quotient MDP of $\mathcal{M}$ with respect to the equivalence relation induced by the partitioning into maximal end-components (i.e., $s \equiv s'$ iff $(s = s') \vee \exists i, s, s' \in C_i$). Note that in $\overline{\mathcal{M}}$ the probability of moving from some state $\overline{s}$ (representative of an equivalence class) with action a to another state $\overline{t}$ is $\sum_{t \in \overline{t}} p(s,a)(t)$. The actions enabled in $\overline{s}$ are all the actions enabled in any state equivalent to s. We modify $\overline{\mathcal{M}}$ to obtain an Average-MDP $\mathcal{M}'$ by removing all actions for which there is a state s such that $p(s,a)(s) = 1$. Furthermore, for all states s that represent an end-component C_i with value $\lambda_i < \infty$ we add a new action a_i with $p(s,a_i)(s) = 1$ and costs λ_i; all other actions have cost 0. We now recursively remove states without an enabled action and actions leading to removed states. If the initial state s_0 is removed, the Ratio-MDP has value infinity, because we cannot avoid to reach and stay in an end-component with value infinity. Otherwise, let d' be an optimal strategy for $\mathcal{M}'$. We define d by $d(s) = d'(\overline{s})$ for all states $s \notin \bigcup C_i$. For $s \in C_i$, if $d'(\overline{s}) = a_i$, we set $d(s) = d_i(s)$. Otherwise, let a be the action chosen in state $\overline{s}$, and let s' be the state in which a is enabled. Then, we set $d(s') = a$ and forall other states in C_i we choose d such that we reach s' with probability 1. We can choose the strategy arbitrarily in states that were removed from $\mathcal{M}'$, because these states will never be reached by construction of d.

4.1 General Shape

The general shape of the algorithms is shown in Algorithm 1. In Line 1 we decompose the Ratio-MDP into maximal end-components [7]. Then, we analyse each end-component separately: the predicates `isZero` and `isInfty` (Line 4 and 5, resp.) check if an end-component has has value zero or infinity using Lemma 4. If both checks fail, we use the function `solveEC` (Line 6), which implements one of the three algorithms presented in Section 4.2, 4.3, and 4.4 to compute the optimal value and an optimal strategy for this end-component. Finally, function `compose` (Line 9) takes these values and strategies from all the end-components and computes an optimal strategy for $\mathcal{M}$ using Lemma 5.

```
  Input: Ratio-MDP $\mathcal{M}$, start state $s_0$
  Output: Value $\lambda$ and optimal strategy $d$
1 $ecSet \leftarrow$ decompose($\mathcal{M}$);
2 foreach $i \leftarrow [0 \ \ldots \ |ecSet| - 1]$ do
3     switch $ecSet_i$ do
4         case isZero : $\lambda_i \leftarrow 0$; $d_i \leftarrow$ zero-cost strategy;
5         case isInfty : $\lambda_i \leftarrow \infty$; $d_i \leftarrow$ arbitrary; ;
6         otherwise : $d_i \leftarrow$ solveEC($ecSet_i$);
7     endsw
8 end
9 $d \leftarrow$ compose($\mathcal{M}, \lambda_0, \ldots, \lambda_{|ecSet|-1}, d_0, \ldots, d_{|ecSet|-1}$);
```

Algorithm 1: Finding optimal strategies for Ratio-MDPs

4.2 Policy Iteration

In this section we present a policy iteration algorithm for Ratio-MDPs and prove that it terminates and converges. Due to Lemma 5 we consider only Ratio-MDPs that are end-components and Lemma 4 allows us to assume that the value of the Ratio-MDP is neither zero nor infinity. The key idea is to construct a sequence of Average-MDPs $\mathcal{M}_i$ from the Ratio-MDP and a decreasing sequence of values λ_i. In every step, we search for an improved strategy in $\mathcal{M}_i$, which is mapped to an improved strategy in the Ratio-MDP leading to the next value λ_{i+1}.

Induced Average-MDP. Given a Ratio-MDP $\mathcal{M}$ and a value λ, we call an Average-MDP with the same structure as $\mathcal{M}$ and the cost function $c' = c - \lambda r$ the *Average-MDP induced by $\mathcal{M}$ and λ* and denote it by $\mathcal{M}_\lambda$.

Improvement. Given a Ratio-MDP $\mathcal{M}$ and a unichain strategy d with value λ and stationary distribution π, we use the Average-MDP $\mathcal{M}_\lambda$ induced by a value λ to improve strategy d. According to Lemma 3, d has ratio value $\lambda = \pi c_d / \pi r_d$, which is equivalent to $0 = \pi(c_d - \lambda r_d) = \pi c'_d$, showing that the average value g of d in $\mathcal{M}_\lambda$ is zero. Assume there exists a unichain strategy d' with an average value g' smaller than zero in $\mathcal{M}_\lambda$, then d' is also a better strategy in the original Ratio-MDP. To prove this, let π' be the stationary distribution of d'. Then, $0 > g' = \pi' c'_{d'} = \pi'(c_{d'} - \lambda r_{d'})$ by the definition of value in an Average-MDP. This is equivalent to $0 > \pi' c_{d'} - \pi' \lambda r_{d'}$ and $\lambda > \pi' c_{d'} / \pi' r_{d'} = \lambda'$ (the ratio value of d') and leads to the following lemma.

Lemma 6. *Given an MDP $\mathcal{M}$, let d and d' be two unichain strategies with values λ and λ', respectively. Then (1) $\lambda = \lambda'$ if and only if the value of d' in $\mathcal{M}_\lambda$ is 0 and (2) $\lambda' < \lambda$ if and only if the value of d' in $\mathcal{M}_\lambda$ is smaller than 0.*

Lemma 6 shows that we can use policy iteration on Average-MDPs to find an improved strategy for the original Ratio-MDP, provided we get a unichain strategy from the improvement. Since Average-MDPs are a special case of Ratio-MDPs, we can use Lemma 2 to convert a multichain strategy to a unichain strategy with a smaller or equal gain in the Average-MDP.

```
    Input: End-component $\mathcal{M}$, unichain strategy $d_0$ (with $0 < \lambda_0 < \infty$)
    Output: Optimal unichain strategy $d_n$
1   $n \leftarrow 0$;
2   repeat
3       $\lambda_n \leftarrow \mathbb{E}^d_{\mathcal{M}}[\mathcal{R}]$;
4       $d_{n+1} \leftarrow$ improved unichain strategy for $\mathcal{M}_{\lambda_n}$;
5       $n \leftarrow n + 1$;
6   until $d_{n-1} = d_n$;
```

Algorithm 2: Policy iteration using Average-MDPs

Algorithm 2 searches for an improved strategy d of $\mathcal{M}_\lambda$ before updating λ and computing a new induced Average-MDP. Instead of asking for an improved strategy in Line 4, we can also ask for a strategy that is optimal with respect to the induced Average-MDP. In Section 6 we show a comparison of these options.

In order to use Algorithm 2 we need an initial unichain strategy d_0 with finite and non-zero value. Due to the case analysis in Algorithm 1, we know that the value of the analyzed end-component C is neither zero nor infinity. Therefore, (i) there is no strategy in C that has value zero and (ii) there exists at least one strategy with value $\lambda < \infty$. Strategy d_0 can be constructed as follows: (i) pick state s and action a such that $r(s, a) > 0$, (ii) set $d_0(s)(a) = 1$ and for all other states $s' \neq s$ pick a pure and memoryless strategy to reach s with probability 1 (such a strategy exists because we consider only end-components).

Termination and Convergence. In Line 4 we search for an improved strategy. According to Lemma 6 if such a strategy is found, then λ_n will decrease in the next step. There are only finitely many strategies and hence the algorithm terminates. It remains to show that we always find such a strategy if possible. Assume some non-optimal unichain strategy d_n with value λ_n, and assume that d^* is optimal and has value λ^*. We now show that d^* has value smaller zero in the MDP induced by λ_n. Let π be the stationary distribution of d^*. Let $v^* = \pi c_{d^*} - \lambda^* \pi r_{d^*}$ be the value of d^* in Average-MDP $\mathcal{M}_{\lambda^*}$ and let $v = \pi c_{d^*} - \lambda_n \pi r_{d^*}$ be the value of d^* in the Average-MDP $\mathcal{M}_{\lambda_n}$. From $\lambda^* < \lambda_n$ it follows that $v^* > v$ and from $v^* = 0$ it follows that $v < 0$. Hence, for each Average-MDP induced by a non-optimal strategy, there exists a strategy with a value smaller than zero.

4.3 Fractional Linear Program

The following fractional linear program also allows us to find an optimal solution for a Ratio-MDP that is an end-component with a finite ratio value. In [14], we provide a detailed explanation of the program in the case of recurrent Ratio-MDP, which can be extended to end-components with a finite ratio value by adapting the strategy construction. The fractional linear program minimizes

$$\frac{\sum_{s \in S} \sum_{a \in \tilde{A}(s)} x(s, a) c(s, a)}{\sum_{s \in S} \sum_{a \in \tilde{A}(s)} x(s, a) r(s, a)}$$

subject to the constrains $\sum_{a \in \tilde{A}(s)} x(s, a) = \sum_{s' \in s} \sum_{a \in \tilde{A}(s')} x(s', a) p(s', a)(s)$ for all states $s \in S$ and $\sum_{s \in S} \sum_{a \in \tilde{A}(S)} x(s, a) = 1$.

We construct a strategy from a solution for $x(s, a)$ as suggested by e.g. [13] for communicating Average-MDPs: we set $d(s) = a$ iff $x(s, a) > 0$. Using Lemma 2 we can construct unichain strategy from d with the same value.

4.4 Linear Program

We can also use the following linear program proposed in [7] to calculate an optimal strategy. We are presenting it here because we compare it to the other solutions in Section 6. The goal is to maximize λ subject to $h_s \leq c_s - \lambda r_s + \sum_{s' \in S} p(s, a)(s') h_{s'}$ for all states $s \in S$ and all actions $a \in \tilde{A}(s)$. To calculate a strategy from a solution h_s to the LP we choose the actions for the states such that the constraints are fulfilled when we interpret them as equations.

5 Related Work

In this section with discuss related work and give an example showing the difference between average and ratio objective. Our synthesis and experimental results are summerized in the next Section.

The related work can be divided into two categories: (1) work using MDPs for quantitative synthesis and (2) work on MDP reward structures. From the first category we first consider the work of Chatterjee et al. [5]. We generalize this work in two directions: (i) we consider ratio objectives, a generalization of average-reward objectives and (ii) we introduce a more general environment model based on MDPs that allows the environment to change its behavior based on actions the system has taken. In the same category there is the work of Parr and Russell [11], who use MDPs with weights to present partially specified machines in Reinforcement Learning. Our approach differs from this approach, as we allow the user to provide the environment, the specification, and the objective function separately and consider the expected ratio reward, instead of the expected discounted total reward, which allows us to ask for efficient systems.

Semi-MDPs [13] fall into the second category. Unlike work based on Semi-MDPs, we allow a reward of value 0. Furthermore, we provide an efficient policy iteration algorithm that works on our Ratio-MDPs as well as on Semi-MDPs. Approaches using the discounted reward payoff (cf. [13]) are also related but focus on immediate rewards instead of long-run rewards. Similarly related is the work of Cyrus Derman [8], who considered the payoff function obtained by dividing the expected costs by expected rewards. As shown later, we believe that our payoff function is more natural. Note that these two objective functions are in general not the same. Closest to our work is the work of de Alfaro [7]. In this work the author also allows rewards with value 0, and he defines the expected payoff over all runs that visit a reward with value greater than zero infinitely often. In our framework the payoff is defined for all runs. De Alfaro also provides a linear programming solution, which can be used to find the ratio value in an end-component (see Section 4.4). We provide two alternative solutions for end-components including an efficient policy iteration algorithm. Finally, we are the first to implement and compare these algorithms and use them to synthesize efficient controllers.

Average versus Ratio Objective. There are well-known techniques for finding optimal strategies for MDPs with average objective. A natural question to ask is, if the average objective would not suffice to describe our problem.

Recall the ACTS unit from Section 1. We want to optimize the relation of two measures: speed (km/h) and fuel consumption (l). In order to use the average objective, we have to combine these two measures. Two methods seem possible. First, we can subtract speed from consumption and minimize the average. When subtracting kilometers per hour from liters, the value of the optimal controller has no intuitive meaning. Furthermore, it can lead to non-optimal strategies as we will show in the next paragraph. Alternatively, we can divide consumption by speed in each step (leading to a measure $\lim_{n\to\infty}(c_1/r_1 + c_2/r_2 \dots)/n$). By this we obtain the correct unit but in general a different value for which the interpretation is not obvious. The same holds for the pumps example discussed in Section 3. We have two different measures: water flow and repair costs, with two different units: liter and dollar. With the ratio objective we can model the problem and its optimization criterion (efficiency) directly, and we can easily interpret the result (expected maintenance cost by liter).

We give a small example to show that simple reduction to subtraction can lead to strategies that differ from the optimal strategy of the ratio objective. Consider an MDP with 2 states, s_0 and s_1. There is one action enabled in s_1. It has cost 1 and reward 100 and leads with probability 1 to s_0. There are two actions in s_0: Action a_0 has cost 5 and reward 1 and leads with probability 1/9 to s_1 and with 8/9 back to s_0. Action a_1 has cost 10 and reward 1 and leads with probability 1/2 to s_1 and with 1/2 to s_0. The steady state distribution of the strategy choosing a_0 is $(9/10, 1/10)$, and so its ratio value is $(9/10 \cdot 5 + 1/10 \cdot 1)/(9/10 \cdot 1 + 1/10 \cdot 100) \approx 0.42$. For the strategy choosing a_1, the steady state distribution is $(2/3, 1/3)$ and the ratio value is $(2/3 \cdot 10 + 1/3 \cdot 1)/(2/3 \cdot 1 + 1/3 \cdot 100) \approx 0.634$, which is larger than the value for a_1. Hence choosing a_0 is the better strategy for the ratio objective. If we now subtract the reward from the cost and interpret the result as an Average-MDP, then we get rewards 4, 9, and -99 respectively. Choosing strategy a_0 gives us $9/10 \cdot 4 - 1/10 \cdot 99 = -6.3$, while choosing strategy a_1 gives us $2/3 \cdot 9 - 1/3 \cdot 99 = -27$. So, choosing a_1 is the better strategy for the average objective.

6 Synthesis and Experimental Results

Synthesis Results. We synthesized optimal controllers for systems with 2 to 5 pumps. They behave as follows: For a system with two pumps, the controller plays it safe. It turns one pump on in fast mode and leaves the other one turned off. If the pump breaks, then the other pump is turned on in slow mode and the first pump is repaired immediately. For three pumps, all three pumps are turned on in fast mode. As soon as one pump breaks, only one pump is turned on in fast mode, the other one is turned off. Using this strategy, the controller avoids the penalty of having no working pump with high probability. If two pumps are broken, then the last one is turned on in fast mode and the other two pumps are been repaired. In the case of four pumps, all pumps are turned on in fast mode if they are all working. If one pump breaks, then two pumps are turned on and the third working pump is turned off. The controller has one pump in reserve for the case that both used pumps break. If two pumps are broken, then only one pump is turned on, and the other one is kept in reserve. Only if three pumps are broken, the controller starts repairing the pumps. Using this strategy, the controller maximizes the discount for repairing multiple pumps simultaneously.

We also modeled the ACTS described in Section 1 using PRISM. The model has two parts: a motor and a driver profile. The state of the motor consists of revolutions per minute (RPM) and a gear. The RPM range from 1000 to 6000, modeled as a number in the interval (10, 60), and we have three gears. The driver is meant to be a city driver, i.e., she changes between acceleration and deceleration frequently. The fuel consumption is calculated as polynomial function of degree three with the saddle point at 1800 rpm. The final model has 384 states and it takes less than a second to build the MDP. Finding the optimal strategy takes less than a second. The resulting expected fuel consumption is 0.15 l/km. The optimal strategy is as expected: the shifts occur as early as possible.

Experiments. We have implemented the algorithms presented in this paper. Our first implementation is written in `Haskell`[7] and consists of 1500 lines of code. We use the Haskell package `hmatrix`[8] to solve the linear equation system and `glpk-hs`[9] to solve the linear programming problems. In order to make our work publicly available in a widely used tool and to have access to more case studies, we have implemented the best-performing algorithm within the explicit-state version of PRISM. It is an implementation of the strategy improvement algorithm and uses numeric approximations instead of solving the linear equation systems.

First, we will give mean running times of our `Haskell` implementation on the pump example, where we scale the number of pumps. The tests were done on a Quad-Xeon with 2.67GHz and 3GB of heap space. Table 1 shows our results. Column n denotes the number of pumps we use, $|S|$ and $|A|$ denote the number of states and actions the final MDP has. Note that $|S| = 3^n$ and $|A| = 12^n$. The next columns contain the time (in seconds) and the amount of memory (in MB) the different algorithms used. LP denotes the linear program, FLP the fractional linear program. We have two versions of the policy iteration algorithm: one in which we improve the induced MDP to optimality (Column Opt.), and one where we only look for any improved strategy (Column Imp.). The policy iteration algorithms perform best, and Imp. is slightly faster than Opt but uses a little more memory. For $n = 5$, the results start to differ drastically. FLP ran out of memory, LP needed about a minute to solve the problem, and both Imp. and Opt. stay below two seconds.

Using our second implementation, we also tried our algorithm on some of the case studies presented on the PRISM website. For example, we used the IPv4 zeroconf protocol model. We asked for the minimal expected number of

[7] http://www.haskell.org
[8] http://code.haskell.org/hmatrix/
[9] http://hackage.haskell.org/package/glpk-hs

Table 1. Experimental results table

n	$\lvert S\rvert$	$\lvert A\rvert$	LP		FLP		Opt		Imp.	
2	9	144	0.002	13	0.015	14	0.003	13	0.003	14
3	27	1728	0.043	14	0.642	20	0.027	13	0.009	14
4	81	20736	1.836	41	14.73	332	0.122	21	0.122	24
5	243	248832	67.77	505	n/a	n/a	1.647	162	1.377	166

occurrences of action `send` divided by occurrences of action `time`. If we choose $K = 5$ and `reset = true`, then the resulting model has 1097 states and finding the optimal strategy takes 5 seconds. For $K = 2$ and `reset = false`, the model has about 90000 states and finding the best strategy takes 4 minutes on a 2.4GHz Core2Duo P8600 laptop.

7 Conclusion

We have presented a framework for synthesizing efficient controllers. The framework is based finding optimal strategies in Ratio-MDPs. To compute optimal strategies we presented three algorithms based on strategy improvement, fractional linear programming, and linear programming, respectively. We have compared performance characteristics of these algorithms and integrated the best algorithm into the probabilistic model checker PRISM. As future work, we are planing to extend our implementation in PRISM to work with symbolically encoded MDPs. Developing a policy iteration algorithm was an important step in this direction, because it allows us to use semi-symbolic (known as symblicit) techniques, which can handle more than 10^{12} states for the long-run average case [15]. We expect that the ratio-case will scale to systems of similar size.

Acknowledgments. The authors would like to thank David Parker for his help with PRISM, and Hugo Gimbert and Luca de Alfaro for answering questions about their work.

References

1. C. Baier and J.-P. Katoen. *Principles of model checking.* MIT Press, 2008.
2. R. Bloem, K. Chatterjee, T. A. Henzinger, and B. Jobstmann. Better quality in synthesis through quantitative objectives. In A. Bouajjani and O. Maler, editors, *CAV*, volume 5643 of *Lecture Notes in Computer Science*, pages 140–156. Springer, 2009.
3. R. Bloem, K. Greimel, T. A. Henzinger, and B. Jobstmann. Synthesizing robust systems. In *FMCAD*, pages 85–92. IEEE, 2009.
4. K. Chatterjee, L. Doyen, and T. A. Henzinger. Quantitative languages. In M. Kaminski and S. Martini, editors, *CSL*, volume 5213 of *Lecture Notes in Computer Science*, pages 385–400. Springer, 2008.
5. K. Chatterjee, T. A. Henzinger, B. Jobstmann, and R. Singh. Measuring and synthesizing systems in probabilistic environments. In T. Touili, B. Cook, and P. Jackson, editors, *CAV*, volume 6174 of *Lecture Notes in Computer Science*, pages 380–395. Springer, 2010.

6. C. Courcoubetis and M. Yannakakis. Markov decision processes and regular events (extended abstract). In M. Paterson, editor, *ICALP*, volume 443 of *Lecture Notes in Computer Science*, pages 336–349. Springer, 1990.
7. L. de Alfaro. *Formal Verification of Probabilistic Systems*. PhD thesis, Stanford University, 1997.
8. C. Derman. On sequential decisions and markov chains. *Management Science*, 9(1):16–24, 1962.
9. B. R. Haverkort. *Performance of computer communication systems - a model-based approach*. Wiley, 1998.
10. D. P. M. Z. Kwiatkowska, G. Norman. PRISM: probabilistic model checking for performance and reliability analysis. *SIGMETRICS Performance Evaluation Review*, 36(4):40–45, 2009.
11. R. Parr and S. J. Russell. Reinforcement learning with hierarchies of machines. In M. I. Jordan, M. J. Kearns, and S. A. Solla, editors, *NIPS*. The MIT Press, 1997.
12. A. Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57. IEEE Computer Society, 1977.
13. M. L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley-Interscience, April 1994.
14. C. von Essen and B. Jobstmann. Synthesizing systems with optimal average-case behavior for ratio objectives. In J. Reich and B. Finkbeiner, editors, *iWIGP*, volume 50 of *EPTCS*, pages 17–32, 2011.
15. R. Wimmer, B. Braitling, B. Becker, E. M. Hahn, P. Crouzen, H. Hermanns, A. Dhama, and O. Theel. Symblicit calculation of long-run averages for concurrent probabilistic systems. In *QEST*, pages 27–36. IEEE Computer Society, 2010.
16. H. Yue, H. C. Bohnenkamp, and J.-P. Katoen. Analyzing energy consumption in a gossiping mac protocol. In B. Müller-Clostermann, K. Echtle, and E. P. Rathgeb, editors, *MMB/DFT*, volume 5987 of *Lecture Notes in Computer Science*, pages 107–119. Springer, 2010.
17. A. P. Z. Manna. *Temporal verification of reactive systems - safety*. Springer, 1995.
18. U. Zwick and M. Paterson. The complexity of mean payoff games on graphs. *Theor. Comput. Sci.*, 158(1&2):343–359, 1996.