Semi-Symbolic Computation of Efficient Controllers in Probabilistic Environments

Christian von Essen, Barbara Jobstmann, David Parker, and Rahul Varshneya

September 11, 2012

Abstract: We present a semi-symbolic algorithm for synthesizing efficient controllers in a stochastic environment, implemented as an add-on to the probabilistic model checker PRISM. The user specifies the environment and the controllable actions using a Markov Decision Process (MDP), modeled in the PRISM language. Controller efficiency is defined with respect to a user-specified assignment of costs and rewards to the controllable actions. An optimally efficient strategy minimizes the ratio between the encountered costs and rewards. At the core of the implementation is the first semi-symbolic algorithm based on a recently developed strategy improvement algorithm for MDPs with ratio objectives. We show the effectiveness of our implementation using a set of benchmarks.

1 Introduction

Efficiency is a central point in controller design because, in many settings, the controller needs to make a trade-off between competing quantities. For instance, consider a production line in which the speed, i.e., the number of produced units, can be adjusted. A controller that produces as many units as possible seems preferable. However, running the line in a faster mode increases the power consumption and the probability to fail, resulting in higher repair costs. Therefore, it is natural to ask for an "efficient" controller, i.e., a controller that minimizes the power and repair costs per produced unit.

In this paper, we present an algorithm that can not only analyze how efficient a controller is but can also automatically construct the "most efficient" controller for a given environment, where efficiency is defined as the ratio between a given *cost* model and a given *reward* model [1]. This choice is inspired by the idea that an efficient system has to balance between the time or effort it uses versus the intended task. For instance, consider an automatic gear-shifting unit (ACTS) that optimizes its behavior for a given driver profile. The goal of the ACTS is to optimize the fuel consumption per kilometer (l/km), a commonly used unit to quantify efficiency. In order to be most efficient, the 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. Objectives of this nature have also been shown to be appropriate in other contexts, such as the ratio between energy consumption and latency and the ratio between detected collisions and failed transmissions as a measure of efficiency in MAC protocols [2].

We also present an implementation of the algorithm, which allows the user to specify a probabilistic model of the system to control, together with an assignment of costs and rewards to each possible action that the controller can choose. The output is a controller that optimizes the expected ratio between the accumulated costs and rewards.

Our implementation is semi-symbolic: it combines both symbolic (Binary Decision Diagram-based) and explicit-state techniques. It takes the form of an add-on to PRISM [3], a probabilistic model checker that supports verification of Markov chains, Markov decision processes (MDPs) and probabilistic timed automata. PRISM provides model checking of a variety of quantitative properties, including some reward-based measures, but has no support for ratio objectives, which require rather different techniques. Ratio-based measures are a useful class of properties that are not expressible with multi-objective MDP model checking [4]. As a side-effect, PRISM can now also handle MDPs with the classical average objective.

Contributions. We provide the first semi-symbolic algorithm and implementation to find optimal strategies for MDPs with ratio objectives (Ratio-MDPs) [1]. Our approach avoids the need for *unichain* MDPs, which was a major bottleneck in the direct symbolic implementation of [1]. Our implementation is fully integrated into the PRISM model checker, providing easy-to-use tool support for the ratio optimization criterion. We also added human-readable output of strategies to PRISM. Our work is related to the work of Wimmer et al. [5], which performs semi-symbolic computation for MDPs with average objectives (a special case of the ratio objectives). We show that our implementation can also outperform theirs.

2 Encoding and Algorithm

In this section, we first recall the symbolic encoding of Markov Decision Processes (MDPs) [6] used in PRISM [3], then we give a formal definition of the ratio objectives. Finally, we present the new semi-symbolic policy iteration algorithm, which is based on our recently-developed explicit version [1].

Encoding MDPs and Strategies. To encode MDPs symbolically, we use Multi-Terminal Binary Decision Diagrams (MTBDDs) [7], which are efficient data structures, generalizing Binary Decision Diagrams (BDDs) [8], to represent functions from finite domains to finite ranges. We encode a Ratio-MDP using a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{T})$ and two MTBDDs $\mathcal{R}, \mathcal{W}$ for the cost/reward function, respectively. The BDD $\mathcal{S} : V \to \mathbb{B}$ represents the set of reachable states using a set of state variables V, the BDD $\mathcal{A} : V \times A \to \mathbb{B}$ encodes for each state the available actions using a set of action variables A, the MTBDD $\mathcal{T} : V \times A \times V \to [0,1]$ encodes the probability of moving from state s to state s' under the condition that a particular action a is chosen. If action a is not available in state s, then the corresponding value in $\mathcal{T}$ is 0. The two MTBDDs $\mathcal{R}, \mathcal{W} : V \times A \to [0, d]$ assign costs (or rewards) to each state-action pair. Strategies of MDPs[1] are also encoded symbolically using a BDD $\Sigma : V \times A \to \mathbb{B}$ that describes for every state, if an action is taken or not.

Ratio Objectives. The ratio objective expresses the ratio between the accumulated costs and the accumulated rewards. Formally, it is defined as

$$\lim_{l\to\infty} \liminf_{n\to\infty} \frac{\sum_{i=l}^{n} \text{costs}(\rho_i)}{1 + \sum_{i=l}^{n} \text{rewards}(\rho_i)}$$

for a given MDP trace ρ (i.e., an infinite sequence of state-action pairs ρ_i). We aim for *efficient* controllers that minimize the costs per obtained reward, i.e., optimal strategies that minimize the expected ratio value.

Overview. The algorithm proceeds as follows: first, it partitions the Ratio-MDP into end-components[2] [9], then it computes an optimal strategy for each end-component, and finally merges these strategies. For the first and the last step, we use algorithms developed for MDPs with average objectives (Average-MDP), a well-studied objective that can be seen as a special case of the ratio objective. We have implemented symbolic versions of these algorithms. The computation of the optimal strategy in an end-component is based on a sequence of reductions of the Ratio-MDP together with a strategy to an Average-MDP (see below). To solve the induced Average-MDPs, we leverage the work of Wimmer et al. [5]. We have reimplemented their semi-symbolic (symblicit) algorithm for Average-MDPs. Once we have computed a strategy for the entire Ratio-MDP, we transform it into a PRISM-like format to make it readable by the user.

Strategies for End-Components. Given an end-component, we first perform two simple checks on its structure to see if the value of this end-component is 0 or ∞. If both checks fail, then we proceed with Algorithm 1, which we will explain in more detail next.

The algorithm first picks any strategy τ that has a finite and strictly positive value (Line 1). We observed that the choice of this strategy has a strong influence on the performance of our algorithm and are currently developing heuristics to optimise this phase. Then, in Line 1 we enter a loop that produces in every iteration a new strategy that has the same or a better ratio value (λ) than the previous strategy. We exit the loop if the same strategy is produced, i.e., there is no strategy with a better ratio value for this MDP.

In the loop, we first compute the ratio value λ that can be obtained by a strategy generated from the strategy τ (Line 1). This computation is done semi-symbolically. First, we compute the Markov chain C induced by strategy τ. For C, we symbolically compute a bisimulation relation [5], which allows us to construct an equivalent smaller Markov chain C'. Then, we compute all recurrence classes (i.e., the strongly connected components) of C'. For each recurrence class, we build an explicit-state representation

[1]We have shown in [1] that we only need to consider pure, memoryless strategies (i.e. those which which pick a single action in each MDP state).

[2]An end-component can be thought of as the analogue of a strongly connected component for MDPs.

```
    Input: MDP $mdp$ consisting of a single end-component
    Output: strategy $\tau$ and optimal ratio value $\lambda$ of $mdp$
 1  $\tau$ = initial($mdp$) ;
 2  $\tau_{old}$ = $\perp$;
 3  while $\tau \neq \tau_{old}$ do
 4      $\lambda$ = lambda($mdp$, $\tau$);
 5      $g$, $b$ = gainAndBias($mdp$, $\tau$, $\lambda$);
 6      while $g \geq 0$ and $\tau_{old} \neq \tau$ do
 7          $\tau_{old}$ = $\tau$;
 8          $\tau$ = next($mdp$, $\tau$, $\lambda$, $g$, $b$);
 9          $g$, $b$ = gainAndBias($mdp$, $\tau$, $\lambda$);
10      end
11  end
12  return unichain($mdp$, $\tau$), $\lambda$
```
Algorithm 1: Optimisation for a single end-component

of the sub-model and calculate the steady-state distribution using the SOR-method [10], which in turn is used to calculate the ratio value of the recurrence class. We set λ to the value of the best recurrence class. This value is not necessarily the value of τ but we can construct a strategy that has value λ. Furthermore, λ is at least as good as the actual value of τ.

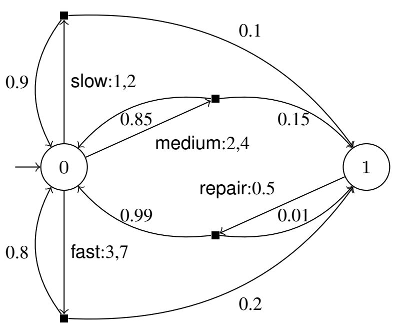
In the rest of the algorithm, we perform computation on an MDP with average objective induced by the reward function $\mathcal{R} - \lambda \times \mathcal{W}$ (which we compute symbolically). For this induced MDP, we compute gain (g) and bias (b) vectors, two measures used by the strategy improvement algorithm for Average-MDPs [6]. The computation of gain and bias is similar to the computation of the ratio value, i.e., we calculate gain and bias explicitly on an equivalent smaller Markov Chain [5]. We know that a state has a gain smaller than zero in the induced MDP if and only if its ratio value is smaller (i.e., better) than the ratio value from which the induced MDP was calculated [1]. In Line 1, the algorithm computes the gain and bias vectors for the current strategy τ. Since the ratio value of strategy τ is at most as good as λ, the gain of all states at this point is greater than or equal to zero.

We now enter the inner loop (Line 1), which runs while the strategy keeps changing and all entries of the gain vector are greater than or equal to zero. Equivalently, the loop runs until there is a recurrence class of the current strategy that has a value smaller than λ or until there is no better strategy anymore. In the inner loop, we try to improve the strategy (Line 1) and calculate the new strategy's gain and bias (Line 1). Note that the choice of the next strategy and the way of computing the value λ differs from our description in [1]. The reason is that, to prove the correctness of the algorithm, we were computing (unichain) strategy. Forcing the algorithm to use a unichain strategy was a major bottleneck in our initial implementation, because it increased the number of bisimulation classes significantly. A new correctness proof is found in the appendix; the key to which is that it is always possible to build a unichain strategy with the calculated ratio value, but it is never necessary to do so.

3 Interaction

In this section, we give a simple example showing how to use our implementation to construct efficient controllers. (For larger and more elaborate examples showing the potential and performance of our add-on, we refer the reader to Section 4.) In the top part of Figure 1, we show the behavior of a simple production line modeled with an MDP. At the bottom, we show the corresponding PRISM encoding, the input to our tool.

The production line can either be in state working (depicted in Figure 1 by a circle labeled with 0) or in state broken (circle labeled with 1). If the line is working, the controller can choose between the three modes of operation: slow, medium, fast (indicated by the three outgoing edges from State 0). The number of items produced per time step and the corresponding maintenance cost, e.g., resulting from the amount of energy consumed depends on the mode of operation: in mode slow, the line produces one item per time step and the maintenance costs amount to two units per time step (indicated by the edge-label slow: 1,2).



```
mdp
module simpleRatio
 state : [0..1] init 0; //0-RUNNING,1-BROKEN
 [slow]   state=0 -> 0.9  : (state' = 0) +
                     0.1  : (state' = 1);
 [medium] state=0 -> 0.85 : (state' = 0) +
                     0.15 : (state' = 1);
 [fast]   state=0 -> 0.8  : (state' = 0) +
                     0.2  : (state' = 1);
 [repair] state=1 -> 0.99 : (state' = 0) +
                     0.01 : (state' = 1);
endmodule
rewards "cost"
 [slow]   true : 2;
 [medium] true : 4;
 [fast]   true : 7;
 [repair] true : 5;
endrewards
rewards "reward"
 [slow]   true : 1;
 [medium] true : 2;
 [fast]   true : 3;
 [repair] true : 0;
endrewards
```

Figure 1: MDP and PRISM encoding of a simple production line

```
*************************************************************
End-component-0 value : 2.378787878787879
*************************************************************
Action slow.a=0, medium.a=1, fast.a=0, repair.a=0, state = 0,
Action slow.a=0, medium.a=0, fast.a=0, repair.a=1, state = 1,
```

Figure 2: Output of PRISM Add-On

In mode medium, the line produces two items and has costs of four and in mode fast, three items are produced and costs amount to seven. The probability of the line breaking down also depends on the mode of operation: the line is more likely to break down if it is operated in a faster mode. E.g., in mode slow, the probability of breaking down is 0.1, while in mode fast it is 0.2. If the line is broken, it can be repaired resulting in maintenance costs of five units.

We aim for a controller that produces the items as efficiently as possible, i.e., it minimizes the average maintenance costs per produced item. To obtain the most efficient controller, we call our implementation with the PRISM model shown in Figure 1. Our tool responds immediately with the result shown in Figure 2. It states that an optimal strategy chooses action medium in state 0, and action repair in state 1. In general, we output for each actions an expression describing the states in which this action is taken. The expression corresponds to the guard of the action in the PRISM model.

4 Experimental results

Table 1 shows the results of our implementation on various benchmarks. The implementation can be downloaded from `http://www-verimag.imag.fr/~vonessen/ratio.html`. The first column shows the name of the example; column `#States` denotes the number of states the model has; `#Blocks` the maximum number of blocks of the partitions we construct while analyzing the model; `Time` the total time needed; `RAM` the amount of memory used (including all memory used by PRISM and its Java

Table 1: Experimental results table

Name	#States	#Blocks	Time in sec	RAM in MB
*pump*3	386	271	0.9	112
*pump*4	1560	945	5.6	150
*pump*5	5904	3089	20.5	236
*pump*6	21394	9448	96.8	326
*rabin*3	27766	722	5.2	199
*rabin*4	668836	12165	104.6	537
zeroconf	89586	29427	2948.7	608
acts	1734	1734	1.6	159
*phil*6	917424	303	1.2	181
*phil*7	9043420	303	1.9	262
*phil*8	89144512	342	2.6	295
*phil*9	878732012	342	3.3	287
*phil*10	8662001936	389	4.3	303
*power*1	8904	72	0.415	89.9
*power*2	8904	n/a	n/a	85

virtual machine). Below, we briefly describe the examples and discuss the results.

Experiments. Examples *pump*3-6 model the water-pump system described in [1]. We optimize the ratio between maintenance costs and amount of water produced by several pumps running in parallel. Example *zeroconf* is based on a model of the ZeroConf protocol [11]. We modify it to measure the best-case efficiency of the protocol, finding the expected time it takes to successfully acquire an IP address. We choose a model with two probes sent, two abstract clients and no reset. This model shows the limit of our technique when bisimulation produces many blocks. In experiments *phil*6-10, we use Lehmann's formulation of the dining philosophers problem [12]. Here we measure the amount of time a philosopher spends. This model is effectively a mean-payoff because we have a cost of one for each step. We use this experiment to compare our implementation to [5]. We are several orders of magnitude faster. We attribute the increase in speed to good initial strategy. In *rabin*3 and *rabin*4, we measure the efficiency of Rabin's mutual exclusion protocol [13]. We minimize the time of a process waiting for its entry into the critical section per entry into the critical section. Note that only the ratio objective allows us to measure exactly this property, because we grant a reward every time a process enters the section and a cost for every time a process has to wait for its entry. We also modeled an automatic clutch and transmission system (*acts*). Each state consists of a driver/traffic state (waiting in front of a traffic light, breaking because of a slower car, free lane), current gear (1-4) and current motor speed (100 - 500 RPM). We modeled the change of driver state probabilistically, and assumed that the driver wants to reach a given speed (50 km/h). Given this driver and traffic profile, the transmission rates and the fuel consumption based on motor speed, we synthesized the best points to shift up or down. In *power*1-2, we used the example from [14, 15], which the authors use to analyze dynamic power management strategies. Our implementation allows solution of optimization problems that are not possible with either [14] or the multi-objective techniques in [15]. For example, in *power*1 we ask the question "What is the best average power consumption per served request". In *power*2, we ask for the best-case power-consumption per battery lifetime, i.e., we ask for how many hours a battery can last.

Observations. The amount of time needed by the algorithms strongly depends on the amount of blocks it constructs. We observed that a higher number of blocks increases the time necessary to construct the partition. Each refinement step takes longer the more blocks we have. Analogously, the more blocks we have, the bigger the matrices we need to analyze are. We observed an almost monotone increase in the number of blocks while policy iteration runs. Accordingly, it is beneficial to select an initial strategy with as few blocks as possible.

In the original policy iteration algorithm of [1], we constructed unichain strategies from multichain strategies several times throughout the algorithm. As it turns out, unichain strategies increase the amount of blocks dramatically. We therefore successfully modified our algorithm to avoid them. These first measures drastically improved performance. When testing our algorithm on various other examples, we sometimes ran out of memory during the decomposition into end-components. Future work will include improving the efficiency of this phase, as well as that of the lumping process.

The symbolic encoding as well as lumping are crucial to handle models of a size that the explicit implementation described in [1] could not handle (storing a model of the size of *phil10* was not feasible on our testing machine).

References

[1] C. von Essen and B. Jobstmann, “Synthesizing efficient controllers,” in *VMCAI*, ser. Lecture Notes in Computer Science, V. Kuncak and A. Rybalchenko, Eds., vol. 7148. Springer, 2012, pp. 428–444. 1, 2, 1, 2, 4, 2

[2] H. Yue, H. C. Bohnenkamp, and J.-P. Katoen, “Analyzing energy consumption in a gossiping mac protocol,” in *MMB&DFT 2010*, 2010, pp. 107–119. 1

[3] M. Z. Kwiatkowska, G. Norman, and D. Parker, “Prism 4.0: Verification of probabilistic real-time systems,” in *CAV*, ser. Lecture Notes in Computer Science, G. Gopalakrishnan and S. Qadeer, Eds., vol. 6806. Springer, 2011, pp. 585–591. 1, 2

[4] V. Forejt, M. Kwiatkowska, G. Norman, D. Parker, and H. Qu, “Quantitative multi-objective verification for probabilistic systems,” in *Proc. TACAS’11*, 2011. 1

[5] 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*, 2010. 1, 2, 2, 4

[6] M. L. Puterman, *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley-Interscience, April 1994. 2, 2, 4

[7] M. Fujita, P. C. Mcgeer, and J. C. Y. Yang, “Multi-Terminal Binary Decision Diagrams: An Efficient Data Structure for Matrix Representation,” *Formal Methods in System Design*, vol. V10, no. 2, pp. 149–169, April 1997. [Online]. Available: http://dx.doi.org/10.1023/A:1008647823331 2

[8] R. E. Bryant, “Symbolic boolean manipulation with ordered binary-decision diagrams,” *ACM Comput. Surv.*, vol. 24, no. 3, pp. 293–318, 1992. 2

[9] L. de Alfaro, “Formal verification of probabilistic systems,” Ph.D. dissertation, Stanford University, 1997. 2

[10] D. M. Young, “Iterative methods for solving partial difference equations of elliptic type,” *Transactions American Mathematical Society*, vol. 76, no. 92-111, 1954. 2

[11] M. Z. Kwiatkowska, G. Norman, D. Parker, and J. Sproston, “Performance analysis of probabilistic timed automata using digital clocks,” *Formal Methods in System Design*, vol. 29, no. 1, pp. 33–78, 2006. 4

[12] D. J. Lehmann and M. O. Rabin, “On the advantages of free choice: A symmetric and fully distributed solution to the dining philosophers problem,” in *POPL*, 1981. 4

[13] M. O. Rabin, “N-process mutual exclusion with bounded waiting by 4 $log_2 n$-valued shared variable,” *J. Comput. Syst. Sci.*, vol. 25, no. 1, pp. 66–75, 1982. 4

[14] G. Norman, D. Parker, M. Z. Kwiatkowska, S. K. Shukla, and R. Gupta, “Using probabilistic model checking for dynamic power management,” *Formal Asp. Comput.*, vol. 17, no. 2, pp. 160–176, 2005. 4

[15] V. Forejt, M. Z. Kwiatkowska, G. Norman, D. Parker, and H. Qu, “Quantitative multi-objective verification for probabilistic systems,” in *TACAS*, ser. Lecture Notes in Computer Science, P. A. Abdulla and K. R. M. Leino, Eds., vol. 6605. Springer, 2011, pp. 112–127. 4

Appendix

This appendix provides a proof of correctness of Algorithm 1, omitted from the main paper due to space limitations.

Lemma 1. *Let λ be some rational or irrational number, let $\mathcal{M}$ be an MDP and let d be a strategy. Further, let $C_0, C_i, \ldots$ be the recurrence classes of $\mathcal{M}$ under* d*, let π_{C_i} be the steady state distribution of recurrence class C_i and let g_{C_i} be the gain of class C_i in $\mathcal{M}_\lambda$ under strategy* d*, i.e. $g_{C_i} = \pi_{C_i} \times (c_d - \lambda \times r_d)$. Then $g_{C_i} \geq 0 \iff \frac{\pi_{C_i} \times c_d}{\pi_{C_i} \times r_d} \geq 0$.*

Proof. $g_{C_i} \geq 0 \iff \pi_{C_i} \times (c_d - \lambda \times r_d) \geq 0 \iff \pi_{C_i} \times c_d - \lambda \times \pi_{C_i} \times r_d \geq 0 \iff \pi_{C_i} \times c_d \geq \lambda \times \pi_{C_i} \times r_d \iff \frac{\pi_{C_i} \times c_d}{\pi_{C_i} \times r_d} \geq 0$ □

Lemma 2. *Let $\mathcal{M}$ be an MDP and d be a strategy. Then d and its value λ are optimal if and only if* `improveUsingGain` *and* `improveUsingBias` *can find no strategy with a recurrence class with gain smaller than 0 [1].*

Theorem 1. *Let $d_0, d_1, \ldots$ be the sequence of strategies of the algorithm. Further, let λ_i be the best ratio value of all recurrence classes of d_i, let g_i be the gain of d_i and let b_i be the bias of d_i. Then $(\lambda_i, g_i, b_i) > (\lambda_{i+1}, g_{i+1}, b_{i+1})$ for all i and the last strategy is optimal.*

Proof. We first prove that $(\lambda_i, g_i, b_i) > (\lambda_{i+1}, g_{i+1}, b_{i+1})$ for all i. From the algorithm, we have that d_{i+1} is a result of `improveUsingGain` or of `improveUsingBias`, and that d_{i+1} is not equal to d_i. If $g_{i+1} \geq 0$, then the best ratio value of all recurrence classes of d_{i+1} is equal to λ_i. According to [6], $g_i > g_{i+1}$ or $b_i > b_{i+1}$. Otherwise there is a recurrence class with gain smaller than zero. According to Lemma 1, the best ratio value a recurrence class is smaller than λ_i.

The last strategy is such that neither `improveUsingGain` nor `improveUsingBias` can improve the strategy. From Lemma 2 shows that it is optimal. □