Mathematics Interdisciplinary Research **9** (2) (2024) 151 − 169

Original Scientific Paper

Improving Probabilistic Bisimulation for MDPs Using Machine Learning

Mohammadsadegh Mohagheghi and Khayyam Salehi⋆

Abstract

The utilization of model checking has been suggested as a formal verification technique for analyzing critical systems. However, the primary challenge in applying to complex systems is the state space explosion problem. To address this issue, bisimulation minimization has emerged as a prominent method for reducing the number of states in a system, aiming to overcome the difficulties associated with the state space explosion problem. For systems with stochastic behaviors, probabilistic bisimulation is employed to minimize a given model, obtaining its equivalent form with fewer states. In this paper, we propose a novel technique to partition the state space of a given probabilistic model to its bisimulation classes. This technique uses the PRISM program of a given model and constructs some small versions of the model to train a classifier. It then applies supervised machine learning techniques to approximately classify the related partition. The resulting partition is then used to accelerate the standard bisimulation technique, significantly reducing the running time of the method. The experimental results show that the approach can decrease significantly the running time compared to state-of-the-art tools.

Keywords: Probabilistic bisimulation, Markov decision process, Model checking, Machine learning, Support Vector Machine.

2020 Mathematics Subject Classification: 68N30, 68Q60, 68T01.

How to cite this article
M. Mohagheghi and K. Salehi, Improving probabilistic bisimulation for MDPs using machine learning, *Math. Interdisc. Res.* **9** (2) (2024) 151-169.

⋆Corresponding author (E-mail: kh.salehi@sku.ac.ir)
Academic Editor: Mahdi Dehghani
Received 6 August 2023, Accepted 13 October 2023
DOI: 10.22052/MIR.2023.253367.1431

1. Introduction

In today's world, computers are everywhere, and when they malfunction, the effects can be profound. Furthermore, proving the accuracy of computer systems is important since some safety features that fail could endanger human life. One mistake in the launch of a rocket, for instance, might have a negative impact on the entire operation [1].

Testing is a promising approach to ensure the correctness of a system. However, it is unable to cover all possible scenarios and verify the system's correctness entirely [2]. In contrast, formal methods utilize mathematical techniques to determine if a system would function correctly under all potential circumstances. There are two widely used formal methods: theorem proving and model checking. The former employs mathematical proofs to establish the program properties of the system, often requiring expert involvement. Conversely, model checking automatically verifies that the entire system behavior satisfies the desired properties [2]. This paper will focus on adopting the model checking approach.

Model checking is an approach for formally verifying qualitative or quantitative properties of computer systems. It involves using a Kripke structure [2] or labeled transition system to represent the underlying system and employing temporal logic or automata to specify the desired properties. By utilizing software tools, the proposed model is automatically checked to determine if the specified properties can be guaranteed. Given the stochastic nature of many computer systems, probabilistic model checking is available to verify the properties of such systems. Markov decision processes (MDPs) and Discrete-time Markov chains (DTMCs) are two extensions of transition systems used for modeling stochastic computer systems [3]. DTMCs are suitable for modeling fully probabilistic systems, while MDPs can capture both stochastic and non-deterministic behaviors of computer systems [3]. MDPs often involve uncertainty and randomness in their decision-making processes. MDPs provide a powerful framework for studying such scenarios and finding optimal strategies or policies to achieve desired objectives under uncertain conditions. This makes them valuable tools in various fields, including artificial intelligence, control theory, operations research, and robotics, among others.

The primary obstacle in model checking is the state space explosion problem, wherein the size of models grows exponentially as the number of components increases. This limitation restricts the explicit representation of large models [1–3]. To address this challenge, various techniques have been developed over the past few decades. Symbolic model representation [4, 5], compositional verification [6, 7], statistical model checking [8–10], and reduction techniques [11–13] are among the key approaches proposed to tackle this problem. These techniques are widely utilized in model checking tools to alleviate the impact of the state space explosion problem.

One of the techniques used for model reduction is bisimulation minimization, which establishes an equivalence class on the state space of the model [14, 15].

States within each equivalence class, known as bisimilar states, share the same set of properties. By applying a bisimulation relation, states within a class can be collapsed into a single state, resulting in a reduced model that is equivalent to the original one. Importantly, the reduced model preserves the same set of properties, allowing a model checker to utilize it as a substitute for the original model [2].

The literature defines various types of bisimulation depending on the class of transition systems and the properties being considered. One commonly discussed type is strong bisimulation, where two states, s and t, are considered bisimilar if, for every successor state of s, there exists at least one bisimilar successor state of t, and vice versa [2, 16]. Another type is weak bisimulation, which disregards silent transitions and defines bisimilar states based on a path that includes some silent moves along with a move having the same action [14]. It is worth noting that silent moves opposed to observable moves refer to unimportant or internal transitions [14]. In this paper, our focus is on strong bisimulation, and we propose a machine learning technique to reduce the computational time of iterative algorithms used to compute this particular version of the bisimulation relation for probabilistic systems. Further information about other classes of bisimulation and their associated algorithms can be found in [17].

In previous works, numerous techniques have been proposed for computing probabilistic bisimulation. The initial works on defining bisimulation for probabilistic automata and Markov Decision Processes (MDPs) can be traced back to [18, 19]. Definition of both strong and weak bisimulation for probabilistic systems incorporating non-determinism, along with their associated algorithms, was first introduced in [20]. These works have contributed significantly to the development of techniques for analyzing and verifying probabilistic systems using bisimulation.

To improve the performance of the standard algorithms for computing probabilistic bisimulation in MDPs, we propose a novel approach. The proposed approach uses supervised machine learning techniques to directly compute bisimilar equivalence classes. The computed partition can be used as the initial one for an iterative partition refinement algorithm. One of the benefits of such an approach is its capability to extend to other types of bisimulation or transition systems. To the best of our knowledge, no previous work has used machine learning for classifying bisimilar blocks of states. In summary, the main contributions of our work are as follows:

- We use supervised machine learning to classify the state space of a model to its bisimilar blocks. Our technique uses several small versions of a given model for the training step.

- We define the superblock concept. A superblock contains several bisimilar blocks. In computing probabilistic bisimulation, the number of bisimilar blocks of different versions of a model is different; but, the number of superblocks is fixed.

The structure of the paper is as follows. In Section 2, we review some preliminary definitions of MDPs and probabilistic bisimulation and the standard algorithm for computing a probabilistic bisimulation partition. In Section 3, we describe the proposed approach for using machine learning to compute bisimilar classes of a given MDP model. Section 4 provides the experimental results running on several classes of the standard benchmark models. Finally, Section 5 concludes the paper and introduces some future work.

2. Preliminaries

In the context of a finite set S, a distribution μ over S is a function $\mu : S \to [0, 1]$ that assigns non-negative values to each element of S such that $\sum_{s\in S} \mu(s) = 1$. In other words, for every $s \in S$, the value $\mu(s)$ represents the probability associated with s. The set S is considered as the state space, and each member $s \in S$ is referred to as a state.

The set of all distributions over S is denoted by $D(S)$. It encompasses all possible functions that satisfy the conditions of being a distribution over the set S. Furthermore, given a subset $T \subseteq S$ and a distribution μ, the accumulated distribution over T, denoted as $\mu[T]$, is defined as the sum of the probabilities or weights assigned to the states within T. Mathematically, it is defined as $\mu[T] = \sum_{s\in T} \mu(s)$ [14].

A partition $\mathcal{B}$ of a set S consists of non-empty and disjoint subsets, forming equivalence blocks, which cover the entire set S. An equivalence relation R is defined based on the blocks, where two states are considered equivalent if they belong to the same block. Formally: $s\ R\ t$ if and only if $\exists B_i \in \mathcal{B}$ s.t. $s, t \in B_i$.

The set of equivalence classes of R on S is denoted as S/R. For a subset T of S, T/R represents the set of states related to at least one state in T, that is, $T \subseteq S$, $T/R = \{s \in S|\ \exists t \in T : sRt\}$. A partition $\mathcal{B}_1$ is considered finer than $\mathcal{B}_2$ if every block in $\mathcal{B}_1$ is a subset of a block in $\mathcal{B}_2$. Additionally, the equivalence relation R on distributions over S can be extended by comparing accumulated distributions within the equivalence blocks defined by the partition, formally: $\mu R\nu$ if and only if $\mu[C] = \nu[C]$ for every block $C \in S/R$. That is, the accumulated distribution μ of C is the same as ν.

Definition 2.1 (Markov Decision Process (MDP))**.** An MDP M represents as a tuple (S, s_0, Act, δ, G) where S is a finite set of states, $s_0 \in S$ is an initial state, Act is a set of finite actions, $\delta : S \times Act \to \mathcal{D}(S)$ is a (partial) probabilistic transition function, which maps a state and an action to a distribution of states, and G is the subset of states representing the set of goal states.

MDPs are widely used as mathematical models to represent and analyze systems that exhibit both non-deterministic and probabilistic behavior. The number of states in the MDP is denoted by $|S|$, and the number of actions available is represented by$|Act|$. The set of actions enabled in each state s is denoted by $Act(s)$.

In other words, in state s, we can select an action $\alpha \in Act\,(s)$.

MDP M works as follows. It starts by selecting an initial state s_0. Once MDP M is in a particular state s, a nondeterministic choice between the enabled actions needs to be resolved. Suppose action $\alpha \in Act(s)$ is non-deterministically chosen. Then, according to the induced distribution $\mu = \delta(s, \alpha)$, the next state s' is probabilistically specified. To resolve non-deterministic choices of an MDP, the notion of policies (also known as adversaries) is utilized. A policy is a (deterministic) mapping that associates each state $s \in S$ with a specific enabled action $\alpha \in Act(s)$.

Reachability properties of probabilistic systems are determined as the probability of achieving a set of states of the model. For MDPs, the properties can be determined as the extremal (maximal or minimal) probability of reaching a goal state G over all possible policies. In bounded reachabilities, the number of steps that can be taken is restricted to a predetermined bound. More comprehensive details on probabilistic model checking and the specific iterative methods used for computing reachability properties can be found in [2].

Definition 2.2 (Probabilistic Bisimulation)**.** A probabilistic (strong) bisimulation $R \subseteq S \times S$ is an equivalence for M if and only if for each pair of states $s, t \in S$, the property sRt implies that for every action $\alpha \in Act(s)$ there exists an action $\beta \in Act(t)$ such that $\delta\,(s, \alpha)\ R\ \delta\,(t, \beta)$. In this case, the probability of leaving each block is the same for both actions. The name of actions is irrelevant to characterized the bisimilarity of two states; whereas in probabilistic automata, actions names should be taken into account.

Two states $s, t \in S$ are probabilistically bisimilar if and only if there exists a probabilistic bisimulation R such that sRt. In the literature, probabilistic bisimulation is characterized in terms of a goal set of states G; that is, we have either $s, t \in G$ or $s, t \in S \setminus G$ for any pair of bisimilar states s and t.

The key feature of probabilistic bisimulation is that for any pair of bisimilar states $s, t \in S$, the same set of bounded and unbounded reachability properties are satisfied in both states [2]. Consequently, a reduced bisimilar (especially smaller) MDP can be created by exchanging all bisimilar states of any block $B_i \in \mathcal{B}$ of original MDP M by one state.

Example 2.3. Consider an MDP model depicted in Figure 1. It has 8 states including s_1 and G that are initial and goal state, respectively. State s_7 is a dead state that can never reach the goal state G. Assume that MDP is in state s_5, it nondeterministically select one of the actions a or b. If a is selected, the next state of MDP is chosen probabilistically, that is, with probability of 0.8, the next state is s_7 and with probability of 0.2, the next one is goal state G. Based on Definition 2.2, s_5 and s_6 are bisimilar because they have the same probability distributions in each action over the state models. However, s_4 is not bisimilar with any other state. Because of bisimilarity of s_5 and s_6, there is also a bisimilarity between s_2 and s_3. These two states have only one action with

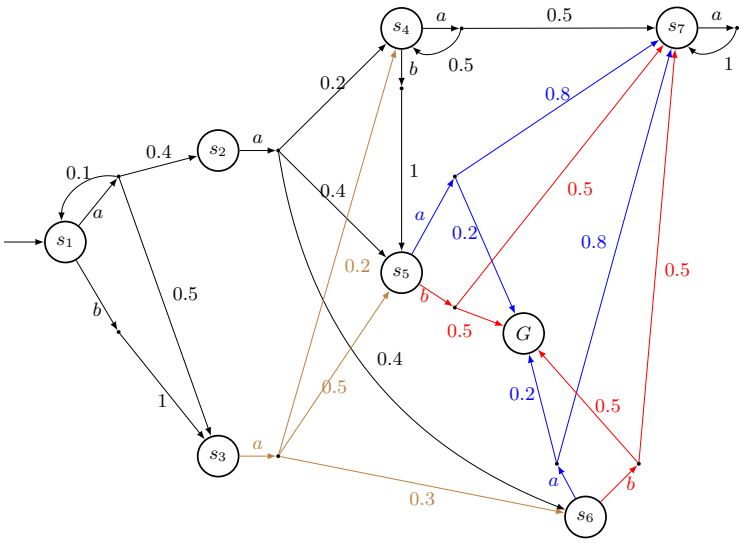

Figure 1: A sample MDP model.

the same accumulated probability of reaching bisimilar blocks $\{s_4\}$ and $\{s_5, s_6\}$. Finally, initial state s_1 is not bisimilar with other states. Thus, we have partition $\mathcal{B}_1 = \{\{s_1\}, \{s_2, s_3\}, \{s_4\}, \{s_5, s_6\}, \{s_7\}, \{G\}\}$. Also, a trivial partition is $\mathcal{B}_0 = \{\{s_1, s_2, s_3, s_4, s_5, s_6, s_7, G\}\}$. It is clear that $\mathcal{B}_1$ is finer than $\mathcal{B}_0$, beacuse each block in $\mathcal{B}_1$ is a subset of a block in $\mathcal{B}_0$.

2.1 The standard algorithm for computing a probabilistic bisimulation

Partition refinement is a widely applicable algorithm for computing a bisimulation relation in various types of transition systems. The algorithm begins with an initial partition and proceeds iteratively by refining the partitions through the splitting of certain blocks into smaller, more refined blocks. The iterations continue until a fixed point is reached, meaning that no further splitting of blocks is possible (Figure 2).

In each iteration, a block is chosen (randomly or based on some orderings [21]) as a splitter to divide some predecessor blocks into smaller and finer ones. The specific method of splitting a block depends on the definition of bisimulation tailored to the underlying transition system being analyzed. Algorithm 1 outlines the steps involved in this approach [2].

In probabilistic bisimulation, the refinement procedure for partitioning blocks takes into account the probabilities associated with reaching a splitter block C. This procedure involves splitting a block B_i from the current partition $\mathcal{B}$ into

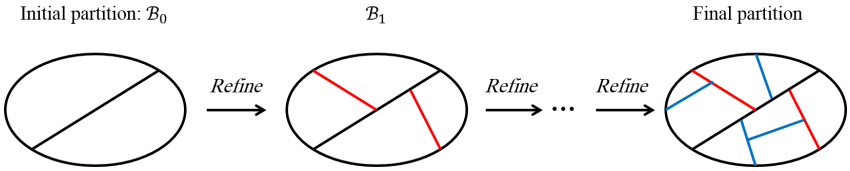

Figure 2: Successive partition refinement procedure.

Algorithm 1: Partition refinement algorithm [2]

Input: An MDP model M
Output: bisimulation partition $\mathcal{B}$
1 Initialize $\mathcal{B}$ to a first partition;
2 **while** *there is a splitter for* $\mathcal{B}$ **do**
3 Choose a splitter C for $\mathcal{B}$;
4 $\mathcal{B} := \mathit{Refine}(\mathcal{B}, C)$;
5 **return** $\mathcal{B}$;

multiple subblocks $B_{i,1}, B_{i,2}, \cdots, B_{i,k}$ based on the following conditions:

1. $\cup_{1 \le j \le k} B_{i,j} = B_i$,
2. $B_{i,j} \cap B_{i,l} = \emptyset$ for $1 \le j < l \le k$,
3. for each $1 \le j \le k$ and every two states $s, t \in B_{i,j}$, it holds that for each action $\alpha \in Act(s)$ there is an action $\beta \in Act(t)$ where $\delta(s, \alpha)[C] = \delta(t, \beta)[C]$.

By utilizing an efficient data structure, the time complexity of the *Refine* method in Algorithm 1 (Line 4) is in $O(|M| + |S| \cdot |Act| \cdot \log |Act|)$ [16]. In the algorithm, a queue of blocks is used, where after refining each block, all computed subblocks, except the largest one, are added to the queue as potential splitters. This strategy ensures that each state is considered in some splitters for at most $\log(|S|)$ times. Based on this approach, the overall time complexity of Algorithm 1 for computing probabilistic bisimulation is in $O(|M| \cdot \log |S| + |S| \cdot \log |S| \cdot |Act| \cdot \log |Act|)$.

There are various approaches to compute the initial partition $\mathcal{B} \subseteq S \times S$. One possible method is to consider two blocks, G and $S \setminus G$, as the initial partition and use G as the first splitter. Using a finer initial partition, Algorithm 1 requires fewer iterations to reach the fixed point. In this paper, a novel heuristic is proposed for computing the initial partition. This heuristic incorporates a machine learning technique to approximate the relevant partition of the probabilistic bisimulation relation. Ideally, the approximated partition aligns with the final partition of the probabilistic bisimulation relation, yielding the best-case scenario.

2.2 The PRISM modeling language

The standard approach in model checking is to use a high-level modeling language to propose a description of the underlying system. A model checker translates the proposed program to a transition system as the semantics of the model. PRISM programs [22] can be used for the case of probabilistic model checking. In this modeling language, each program contains one or more modules, while each module has several variables with a defined domain of values. Several guarded commands describe possible transitions of the model. A probabilistic model checker (such as PRISM [22] or STORM [23]) parses a program to a related MDP or DTMC. An example of a PRISM program is proposed in Figure 3. It defines an MDP model

```
mdp

const int N=2;
const int K;
const int range = 2*(K+1)*N;
const int counter_init = (K+1)*N;
const int left = N;
const int right = 2*(K+1)*N - N;

global counter : [0..range] init counter_init;

module process1

        pc1 : [0..3];
        coin1 : [0..1];

        [] (pc1=0)  -> 0.5 : (coin1'=0) & (pc1'=1) + 0.5 : (coin1'=1) & (pc1'=1);
        [] (pc1=1) & (coin1=0) & (counter>0) -> (counter'=counter-1) & (pc1'=2) & (coin1'=0);
        [] (pc1=1) & (coin1=1) & (counter<range) -> (counter'=counter+1) & (pc1'=2) & (coin1'=0);
        [] (pc1=2) & (counter<=left) -> (pc1'=3) & (coin1'=0);
        [] (pc1=2) & (counter>=right) -> (pc1'=3) & (coin1'=1);
        [] (pc1=2) & (counter>left) & (counter<right) -> (pc1'=0);
        [done] (pc1=3) -> (pc1'=3);

endmodule

// construct remaining processes through renaming
module process2 = process1[pc1=pc2,coin1=coin2] endmodule

// labels
label "finished" = pc1=3 & pc2=3 ;
label "all_coins_equal_0" = coin1=0 & coin2=0 ;
label "all_coins_equal_1" = coin1=1 & coin2=1 ;
label "agree" = coin1=coin2 ;
```

Figure 3: The PRISM code for the *Coin* MDP model.

with two modules *process1* and *process2* while the second module is a copy of the first one. The first module has two variables *pc1* and *coin1* with the defined domain of values. Moreover, several constants with known values and a parameter constant (K) are used in the definition. A global variable *counter* is defined that its upper-bound is determined by K. Thus, using different values for the parameter K, we may have different models with different sizes. In the *process1* module, the first guarded command states that if *pc1* equals to 0, with a probability of 50% *coin1* and *pc1* will be 0 and 1, respectively; while, with a 50% *coin1* and *pc1* will be 1.

Any valid valuation for the set of model variables induces a state of its associated transition system. However, only the set of states that are reachable from the initial state are needed for model checking and are stored explicitly or implicitly. Formally, for the set of model variables, a state $s_i \in S$ maps any of these variables to a value in its domain. For a set $v_i, v_j, \cdots, v_k$ of state variables, we use $\prod_{v_i,v_j,\ldots,v_k}(s)$ as a projection function, which gets a list of the value of these variables in s. For any subset $R \subseteq S$, we define $\prod_{v_i,v_j,\ldots,v_k}(R) = \cup_{s\in R} \prod_{v_i,v_j,\ldots,v_k}(s)$. For a model M of a given PRISM program, we use $Vars(M)$ for the set of its variables (not constants) and use $Params(M)$ for those variables where upper bounds are bounded by a parameter. We call such variables parametric. For the induced MDP M of Figure 3., we have $Vars(M) = \{counter, pc1,\ coin1,\ pc2,\ coin2\}$ and $Params(M) = \{counter\}$. The MDP variable *counter* is parametric because its upper bound is determined by the model parameter K.

3. The proposed approach

In this section, we propose a novel heuristic for approximating the initial partition. The correctness of the approximated result is checked by the partition refinement method (Algorithm 1). If the approximation requires more refinements, it can be used as a more precise initial partition that may result in faster convergence towards the fixed point. In other words, this can be considered as a preprocessing step of the partition refinement method.

For the sake of simplicity, we assume that every program graph has only one parameter. It should be noted that even with this assumption, a variable with a parametric value domain may have several copies in the model definition and also in the induced MDP. The general scheme of our approach is proposed in Algorithm 2.

In the following subsections, we explain each step in detail. Recall that the main purpose of our approach is to facilitate the computation of bisimilar blocks for a large model, where the running time may be an obstacle. Even in non-precise computed blocks, they can be used to reduce the main model to an abstract version to cope with memory limitations.

3.1 Constructing sample models and computing probabilistic bisimulation

As the first step of our approach, we consider several sample models by using smaller values $p_1, p_2, \cdots, p_n$ for the parameter of the given PRISM program (Line 1 of Algorithm 2). Depending on the structure of the given program, the parameters can be so small that result in some tiny models or they may be large enough to have the same structure as the given model. In the next section, we explain more about the values of parameters for several case studies. Although the precision of machine learning may increase by using more samples, in practice using two or

Algorithm 2: Approximating Initial Partition

Input: A PRISM program with known values as parameters
Output: An approximated partition $\mathcal{B}$ for the induced MDP

1 Construct several sample models $M_{p_1}, M_{p_2}, ..., M_{p_n}$, using smaller values for the model parameter (Section 3.1);
2 For each sample model M_{p_i}, apply the probabilistic bisimulation algorithm and compute its equivalence partition $\mathcal{B}_{p_i}$ (Algorithm 1);
3 For each partition $\mathcal{B}_{p_i}$, compute the set $\{sp_1, sp_2, ..., sp_k\}$ of its superblocks (Section 3.2);
4 Let $\eta(s)$ denote the superblock that s belongs to. (Definition 3.1);
5 Fix a classifier and use η for training (Section 3.3);
6 Use the trained classifier to predict which superblock each state of the underlying model belongs to (Section 3.4);
7 Split states of each superblock to their blocks according to their parameter values (Section 3.5);
8 **return** *the partition $\mathcal{B}$ including the computed blocks of step 7;*

three sample MDP models with several thousand states may be enough. In this case, we have at least ten thousand states as training samples that are considered enough in machine learning. Furthermore, we compute probabilistic bisimulation and the equivalence blocks of each model by utilizing the standard algorithm for MDP models of each probabilistic program (Line 2 of Algorithm 2).

3.2 Computing superblocks

The main idea of our approach is to use a classifier to map each state of the underlying model to a block of the bisimulation equivalence relation. To do so, we consider each variable of an MDP as a feature of samples and each block as a class. Considering the variable values of each state as its feature values, the classifier should determine which class (block) the state may belong to. An important challenge of using computed partitions of the sample models is that the number of blocks is different among different samples (versions) of an MDP model. In this case, a classifier is unable to map states to the correct classes. To cope with this challenge, we gather several blocks of a partition to a superblock. We define a superblock as a collection of several bisimulation blocks such that any state of a block has similar states in the other blocks where the variables are the same except the parametric variables (Line 3 of Algorithm 2).

Definition 3.1. A superblock sp of an MDP M is the largest collection of blocks that for each pair of different blocks B_i, B_j, the following condition holds:

$$\forall s \in B_i \; \exists t \in B_j : \prod_{non-params(M)} (s) = \prod_{non-params(M)} (t).$$

Block #61 --> 25:(3,0,0,1,0)	30:(3,1,0,0,0)	346:(13,0,0,1,1)	356:(13,1,1,0,0)
Block #62 --> 39:(3,1,1,2,0)	44:(3,2,0,1,1)	353:(13,1,0,2,0)	363:(13,2,0,1,0)
Block #63 --> 27:(3,0,0,2,0)	42:(3,2,0,0,0)	347:(13,0,0,2,0)	362:(13,2,0,0,0)
Block #64 --> 57:(4,0,0,1,0)	62:(4,1,0,0,0)	314:(12,0,0,1,1)	324:(12,1,1,0,0)
Block #65 --> 58:(4,0,0,1,1)	68:(4,1,1,0,0)	313:(12,0,0,1,0)	318:(12,1,0,0,0)
Block #66 --> 65:(4,1,0,2,0)	75:(4,2,0,1,0)	327:(12,1,1,2,0)	332:(12,2,0,1,1)
Block #67 --> 59:(4,0,0,2,0)	74:(4,2,0,0,0)	315:(12,0,0,2,0)	330:(12,2,0,0,0)
Block #68 --> 38:(3,1,1,1,1)	351:(13,1,0,1,0)		
Block #69 --> 96:(5,1,0,1,1)	101:(5,1,1,1,0)	288:(11,1,0,1,1)	293:(11,1,1,1,0)
Block #70 --> 89:(5,0,0,1,0)	94:(5,1,0,0,0)	282:(11,0,0,1,1)	292:(11,1,1,0,0)
Block #71 --> 88:(5,0,0,0,0)	280:(11,0,0,0,0)		
Block #72 --> 64:(4,1,0,1,1)	69:(4,1,1,1,0)	320:(12,1,0,1,1)	325:(12,1,1,1,0)
Block #73 --> 127:(6,1,0,1,0)	262:(10,1,1,1,1)		
Block #74 --> 121:(6,0,0,1,0)	126:(6,1,0,0,0)	250:(10,0,0,1,1)	260:(10,1,1,0,0)
Block #75 --> 122:(6,0,0,1,1)	132:(6,1,1,0,0)	249:(10,0,0,1,0)	254:(10,1,0,0,0)
Block #76 --> 71:(4,1,1,2,0)	76:(4,2,0,1,1)	321:(12,1,0,2,0)	331:(12,2,0,1,0)
Block #77 --> 129:(6,1,0,2,0)	139:(6,2,0,1,0)	263:(10,1,1,2,0)	268:(10,2,0,1,1)
Block #78 --> 123:(6,0,0,2,0)	138:(6,2,0,0,0)	251:(10,0,0,2,0)	266:(10,2,0,0,0)
Block #79 --> 102:(5,1,1,1,1)	287:(11,1,0,1,0)		
Block #80 --> 160:(7,1,0,1,1)	165:(7,1,1,1,0)	224:(9,1,0,1,1)	229:(9,1,1,1,0)

Figure 4: Some blocks of a Bisimulation Partition for the *Coin* case study.

The intuition behind this definition is that by increasing the value of a parameter, we expect to have new blocks that are similar to some previous ones except for their parametric values that are higher than the others. In this case, the total number of subblocks does not change among different models of a PRISM program.

For more clarification, consider Figure 4 which shows a list of some blocks of bisimilar states for the *Coin* case study with $K = 3$ as its parameter value. For each block, its number as the order that it is computed and the list of its states including state number and its feature values are reported. As an example, the 73'rd block contains two states: s_{127} and s_{262}. For this case, a superblock contains the 68'th, 73'nd and 79'th blocks because the states of these blocks are of the form $(x, 1, 0, 1, 0)$ and $(y, 1, 1, 1, 1)$ where x and y are the parametric variables. To use superblocks for a classification process, we define η as a mapping from states to superblocks (Line 4 of Algorithm 2). For each state $s \in S$ of a sample model, $\eta(s)$ determines its corresponding superblock:

$$\eta(s) = sp_i \ \textit{iff} \ \exists B \in sp_i, \quad s \in B. \tag{1}$$

3.3 Training step

In the approach, a classifier uses a set of superblocks for a training step. A Support Vector Machine (SVM) is used to accomplish this step. The purpose of this step is

to construct a classifier model to predict classes of the state space for a new given model of the same class of the training step. Hence, we use η as a mapping from state space to superblocks (Line 5 of Algorithm 2). For a given PRISM program, we consider its variables as model features. For the example of *Coin* case study with two modules in the PRISM code, there are five variables where each variable is considered as a model feature. Referring to Figure 4, state 58 can be considered as a state where the value of the first feature is 4 and so on. For any parametric variable, we add the difference of the variable value and its domain upper bound (maximum value for the variable) as an additional feature of the model. This additional feature guarantees the uniqueness of states over all training models, i.e., it is not possible to have the same states among different models. For each state of a model, its features are considered as inputs to η. This mapping is used to label each state for the training step.

3.4 Classifying states into superblocks

For any state of a new model, the classifier determines its related superblock. The precision of a classifier for detecting the correct superclass depends on the structure of the models and associated PRISM programs. Because the training models and the given model have the same structure and only differ in their parameters, we expect to have promising results in most cases.

To improve the precision of our approach, we partition the state space of the given model to several subclasses according to its features. A subclass is assigned according to non-parametric variables of the PRISM program and possible values of these variables (reachable from the initial state). We apply these partitions for both training samples and the given model. For each subclass, a classifier is used to predict related superblocks of its states. In our approach, we first separate state space into several subclasses and then use a support vector machine classifier to improve the precision of classification (Line 6 of Algorithm 2).

3.5 Splitting superblocks to bisimilar classes

As the final step of our approach, the states of each superclass should map to the correct bisimilar blocks. According to our definition of superblocks, a relation among parametric variables of bisimilar states of training sample models determines the possible values for the parameters of bisimilar states. For the example of Subsection 3.2, the bisimilar states are as the form tuples $(x, 1, 0, 1, 0)$ and $(y, 1, 1, 1, 1)$ where $x + y = 16$. For the parametric variable of this sample, we have $counter = 16$. For a given model with a known value for the counter parameter, our approach splits states of this superblock to bisimilar blocks where $x + y = counter$ holds (Line 7 of Algorithm 2). It is noteworthy that the proposed approach is an initialization step for partition refinement algorithm. The soundness of the approach is as the following.

Soundness. To ensure the soundness of the approach proposed in this section, we consider several cases, where the initial partition may contain non-exact bisimilar states:

- Case 1: A block B' includes two or more bisimilar blocks $B_{j1}, B_{j2}, ...$ where each B_{jk} is a block of the correct bisimilar partition. This case happens in the standard bisimulation methods where a coarse relation is considered as an initial partition and the bisimulation algorithm terminates with a set of bisimilar blocks.

- Case 2: A block B'' is proposed in the initial partition where it is a mere subset of an exact bisimilar block. In this case, at least one state s exists that is bisimilar with the states of B'', but the method drops it in another block. For such initial partitions, the bisimulation algorithm results in a finer relation than the correct bisimilar one. Although such partition is not the minimized equivalence relation on the state space, it is sound and satisfies the same properties as a minimized model does [2].

- Case 3: A block contains some but not all states of two or more blocks. A standard bisimulation method will eventually divides the states of such block to several blocks of Case 2. In fact, some splitters will be used for this division. This leads to split some other blocks to finer ones of Case 2. Finally, the algorithm terminates where all blocks are either of Case 2 or the correct blocks.

4. Evaluation

4.1 Experimental setup

In order to demonstrate the effectiveness and scalability of the proposed approach, we consider five classes of standard Markov decision process models. These MDP models serve as representative examples that cover a wide range of scenarios and characteristics. These classes include *Coin*, *Wlan*, *Firewire*, *Zeroconf*, and *Brp* case studies from the PRISM benchmark suit [22]. All of them are parametric and are used to compare our machine learning-based approach. More details about these case studies are available at [22].

To compare our implementation of the proposed methods for computing probabilistic bisimulation with the standard approaches, we consider PRISM [21] , STORM [23], and mCRL2 [24] as the well-known and state-of-the-art tools for computing probabilistic model checking for MDPs. It is worth noting that PRISM implemented in [22] does not have supported probabilistic bisimulation. Mohagheghi and Salehi developed an extensions to PRISM for computing probabilistic bisimulation [21]. In this paper, their implementation is used to compare with the proposed approach.

Table 1: PRISM MDP models for training.

Model Name	Number of variables	Parameter names	Number of states for training
$Coin(N = 4)$	9	*counter*	76,032
$Wlan(N = 5)$	13	$TRANS\_TIME\_MAX$	2,794,536
$Firewire(delay = 3)$	3	*deadline*	710,924
$Zeroconf(N = 900)$	22	K	577,128
Brp	18	MAX	39,796

We have implemented our proposed approach as an extension to the PRISM model checker using PRISM 4.7, which is currently the last version. We implemented the proposed algorithm on a machine running Ubuntu 20.04 LTS with Intel(R) Core (TM) i7 CPU Q720@1.6GHz with 8GB of memory.

4.2 Results and discussion

We propose some information on the selected models in Table 1. For each model, we report the parameter name in the third column while fixing another parameter (which is shown in the first column).

For the proposed approach and each computed partition, the set of superblocks are computed by using the proposed technique in the previous section. To simplify our approach, we gather all singular blocks (blocks with only one state) in one superblock. As the output of this step, we define η as a mapping from states of each sample to the index of their corresponding superblock. This information is stored in some files (one file per sample model).

For the classification step, we develop our approach in Python. Our program reads the stored information of sample models including information of their states (variable values of each state) and the computed mapping. It separates the state space into several subclasses as explained in subsection 3.4. For each subclass, we apply SVM with its default parameters for classification. We first train the classifier for each subclass and then apply it to the states of a given model. In some cases, all states of a subclass are mapped to the same superblock and we need not to use a classifier for them. The number of states in training step on small sample models is shown in the last column in Table 1.

The experimental results are presented in Table 2. The number of states, actions, and transitions are shown in the third, fourth, and fifth columns, respectively. The running time for computing probabilistic bisimulation in PRISM, STORM, and mCRL2 as well as our proposed approach, ML-based, on the selected models are demonstrated in seconds. The Terms *killed* and *timeout* in Table 2 refer to the out of memory error and the running time after one hour.

We report the running time for computing the initial partition in sub-column *init-part* and also the total running time after applying the partition refinement algorithm on initial partition in sub-column *total*. The results show that the time

for computing the initial partition of ML-based approach is approximately half of the total time for computing the partition refinement algorithm. In the proposed approach, we use a supervised machine learning classifier (SVM). To compute a labeled targets for small version of sample data requiered in SVM, we sholud run the traditional probabilistic bisimulation algorithm. Thus, computing the initial partition requires a significant time.

Table 2: Performance comparison of computing bisimulation for the selected MDP with large values.

Model name	Parameter	$\|S\|$ $\times 10^{-3}$	$\|Act\|$ $\times 10^{-3}$	$\|Trns\|$ $\times 10^{-3}$	PRISM	STORM	mCRL2	ML-based	
								init-part	total
Coin (N=5)	K=30	2341	7832	9787	2.35	112	18.9	1.02	**1.97**
	K=50	3890	13016	16267	3.45	303	31.4	1.66	**3.08**
	K=70	5439	18200	22747	4.32	554	45.2	1.97	**3.93**
	K=100	7762	25976	32467	7.22	1178	*killed*	3.23	**6.42**
Zeroconf (N=1500)	K=12	3753	6898	8467	9.33	*killed*	22.7	1.01	**2.76**
	K=14	4426	8144	9988	12.57	*killed*	25.9	1.2	**2.97**
	K=16	5010	9223	11307	15.6	*killed*	30.8	1.56	**3.47**
	K=18	5476	10085	12359	18	*killed*	37	1.68	**3.9**
	K=20	5812	10711	13124	21.43	*killed*	39.3	1.75	**4.04**
Firewire (dl = 36)	ddl=3000	2238	3419	4059	2.08	2283	6.9	0.84	**1.71**
	ddl=10000	7670	11742	13936	10.73	*timeout*	27	1.97	**6.5**
	ddl=15000	11550	17687	20991	13.02	*timeout*	*killed*	2.17	**7.62**
Brp (N=400)	max=150	787	787	1087	**0.41**	2.9	2.4	0.35	0.88
	max=300	1567	1567	2167	**0.81**	12.5	5.1	0.6	1.37
	max=600	3127	3127	4327	**2.19**	51.4	10.6	1.45	3.61
Wlan (N=6)	ttm=1000	8093	12543	17668	**0.89**	320	*killed*	0.71	1.61
	ttm=2500	12769	21925	27051	**1.36**	1900	*killed*	1.22	2.57

For *Coin* case study, the total running time of our machine learning-based approach outperforms the other tools. For example, when parameter $K = 100$, our approach runs in 6.42 seconds, while PRISM runs in 7.22, STORM in 1178, and mCRL2 is *killed* by out of memory error.

In case of *Zeroconf*, STORM is *killed* by out of memory error for the whole parameter values. Our approach reduced the total running time by 3 up to 5 times compared to PRISM, and 8 to 9 times compared to mCRL2.

For *Firewire*, STORM has timeout in greater parameter values, while mCRL2 is killed by memory error. PRISM and ML-based approach run effectively on all models; whereas the running time of ML-based approach is half of the PRISM running time.

On *Brp* models, PRISM runs in the best running time compared to the other tools. Our approach takes more time than PRISM, but dominates STORM and mCRL2. In *Wlan* models, mCRL2 is killed by out of memory error. The running time of STORM increases exponentially as the parameter value increases. Similar to *Brp*, our approach takes more time compared to PRISM, but the time is approximately close to each other. According to the structure of these models, a great amount of states are bisimilar and are gathered in the same block. Therfore, computing a probabilistic bisimulation takes a few number of iterations and con-

sequently a littile time. Thus, our approach may not greatly reduce the running time of computing it. In these two cases, using other classifiers rather than SVM may result in better performance. This is left as a future work.

5. Conclusion

In this work, we have proposed a novel approach to improve the performance of the standard algorithms for computing probabilistic bisimulation for MDP models. The approach uses a machine learning classification technique to even directly determine a bisimulation partition. Experimental results show that our approach outperforms the other available tools. For future work, we aim to extend the proposed technique to the other classes of transition systems such as probabilistic automata or discrete-time and continuous-time Markov chains. On can use other classifiers rather than SVM and compare their running time with the state-of-the-art tools. As another future work, we plan to apply the proposed approach to analyze security protocols, especially anonymity protocols [25] such as dining cryptographers [26], single preference voting [27], crowds [28], and TOR [29].

Conflicts of Interest. The authors declare that they have no conflicts of interest regarding the publication of this article.

References

[1] E. M. Clarke, T. A. Henzinger and H. Veith, Introduction to model checking, *Handbook of model checking* (2018) 1 – 26.

[2] C. Baier and J. P. Katoen, *Principles of Model Checking*, MIT press, 2008.

[3] J. P. Katoen, The probabilistic model checking landscape, *In: Proc. of the 31st Annual ACM-IEEE Symp. on Logic in Comput. Sci.* (2016) 31 – 45, https://doi.org/10.1145/2933575.2934574.

[4] K. L. McMillan, *Symbolic Model Checking*, Springer, 1993.

[5] D. A. Parker, *Implementation of Symbolic Model Checking for Probabilistic Systems*, Ph.D. thesis, University of Birmingham, 2003.

[6] E. M. Clarke, D. E. Long and K. L. McMillan, Compositional model checking, *In: Proc. of the 4th IEEE Symp. on Logic in Comput. Sci.* (1989) 353 – 362, https://doi.org/10.1109/LICS.1989.39190.

[7] L. Feng, M. Kwiatkowska and D. Parker, Compositional verification of probabilistic systems using learning, *In: Proc. 7th Int. Conf. on Quantitative Evaluation of Systems, IEEE CS Press* (2010) 133 – 142, https://doi.org/10.1109/QEST.2010.24.

[8] G. Agha and K. Palmskog, A survey of statistical model checking, *ACM Trans. Model. Comput. Simul.* **28** (2018) 1 − 39, https://doi.org/10.1145/3158668.

[9] A. Legay, A. Lukina, L. M. Traonouez, J. Yang, S. A. Smolka and R. Grosu, Statistical model checking, *Computing and software science: state of the art and perspectives, Springer* (2019) 478 − 504.

[10] A. Legay and M. Viswanathan, Statistical model checking: challenges and perspectives, *Int. J. Softw. Tools Technol. Transfer* **17** (2015) 369 − 376, https://doi.org/10.1007/s10009-015-0384-z.

[11] F. Ciesinski, C. Baier, M. Größer and J. Klein, Reduction techniques for model checking markov decision processes, *In: Proc. 5th Int. Conf. on Quantitative Evaluation of Systems, IEEE CS Press.* (2008) 45 − 54, https://doi.org/10.1109/QEST.2008.45.

[12] H. Hansen, M. Kwiatkowska and H. Qu, Partial order reduction for model checking markov decision processes under unconditional fairness, *In: Proc. 8th Int. Conf. on Quantitative Evaluation of Systems* (2011) 203 − 212, https://doi.org/10.1109/QEST.2011.35.

[13] M. Kwiatkowska, G. Norman and D. Parker, Symmetry reduction for probabilistic model checking, *In: Proc. 18th Int. Conf. Computer aided verification, Springer* (2006) 234 − 248.

[14] C. Baier, P. R. D'Argenio and H. Hermanns, On the probabilistic bisimulation spectrum with silent moves, *Acta Inform.* **57** (2020) 465 − 512, https://doi.org/10.1007/s00236-020-00379-2.

[15] K. Salehi, A. A. Noroozi, S. Amir-Mohammadian and M. Mohagheghi, An automated quantitative information flow analysis for concurrent programs, *In: Proc. 19th Int. Conf. on Quantitative Evaluation of Systems, Springer* (2022) 43 − 63.

[16] J. F. Groote, J. Rivera Verduzco and E. P. De Vink, An efficient algorithm to determine probabilistic bisimulation, *Algorithms* **11** (2018) p. 131, https://doi.org/10.3390/a11090131.

[17] S. Cattani and R. Segala, Decision algorithms for probabilistic bisimulation, *In: Proc. 13th Int. Conf. on Concurrency Theory, Springer* (2002) 371 − 385.

[18] K. G. Larsen and A. Skou, Bisimulation through probabilistic testing, *Inform. and Comput.* **94** (1991) 1 − 28.

[19] R. Segala, *Modeling and Verification of Randomized Distributed Real-Time Systems*, Ph.D. thesis, Massachusetts Institute of Technology, 1995.

[20] M. I. A Stoelinga, *Alea Jacta Est: Verification of Probabilistic, Real-Time and Parametric Systems*, Ph.D. thesis, Radboud University, Nijmegen, 2002.

[21] M. Mohagheghi and K. Salehi, Splitter orderings for probabilistic bisimulation, *arXiv:2307.08614* (2023)

[22] M. Kwiatkowska, G. Norman and D. Parker, Prism 4.0: Verification of probabilistic real-time systems, *In: Proc. Int. Conf. on Computer aided verification, Springer* (2011) 585 – 591.

[23] C. Hensel, S. Junges, J. P. Katoen, T. Quatmann and M. Volk, The probabilistic model checker storm, *Int. J. Softw. Tools Technol. Transfer* **24** (2022) 589 – 610, https://doi.org/10.1007/s10009-021-00633-z.

[24] O. Bunte, J. F. Groote, J. J. A. Keiren, M. Laveaux, T. Neele, E. P. de Vink, W. Wesselink, A. Wijs and T. A. C. Willemse, The mcrl2 toolset for analysing concurrent systems: improvements in expressivity and usability, *In: Proc. 25th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems* (2019) 21–39.

[25] C. Shields and B. N. Levine, A protocol for anonymous communication over the internet, *In: Proc. ACM Conf. Comput. Commun. Secur.* (2000) 33 – 42, https://doi.org/10.1145/352600.352607.

[26] A. A. Noroozi, K. Salehi, J. Karimpour and A. Isazadeh, Secure information flow analysis using the prism model checker, *Int. Conf. on Information Systems Security, Springer* (2019) 154 – 172.

[27] K. Salehi, J. Karimpour, H. Izadkhah and A. Isazadeh, Channel capacity of concurrent probabilistic programs, *Entropy* **21** (2019) p. 885, https://doi.org/10.3390/e21090885.

[28] M. K. Reiter and A. D. Rubin, Crowds: anonymity for web transactions, *ACM Trans. Inf. Syst. Secur.* **1** (1998) 66 – 92, https://doi.org/10.1145/290163.290168.

[29] M. G. Reed, P. F. Syverson and D. M. Goldschlag, Anonymous connections and onion routing, *IEEE J. Sel. Areas Commun.* **16** (1998) 482 – 494, https://doi.org/10.1109/49.668972.

Mohammadsadegh Mohagheghi
Department of Computer Science,
Vali-e-Asr University of Rafsanjan,
Rafsanjan, I. R. Iran
e-mail: mohagheghi@vru.ac.ir

Khayyam Salehi
Department of Computer Science,
Shahrekord University,
Shahrekord, I. R. Iran
e-mail: kh.salehi@sku.ac.ir