Computer Science • 26(2) 2025 https://doi.org/10.7494/csci.2025.26.2.6411

MohammadSadegh Mohagheghi
Khayyam Salehi

USING SPLITTER ORDERING HEURISTICS TO IMPROVE BISIMULATION IN PROBABILISTIC MODEL CHECKING

Abstract *Model checking is used to verify computer-based and cyber-physical systems, but faces challenges due to state space explosion. Bisimulation minimization reduces states in transition systems, easing this issue. Probabilistic bisimulation further simplifies models with stochastic behaviors. Recent techniques aim to improve the time complexity of iterative methods in computing probabilistic bisimulation for stochastic systems with nondeterministic behaviors. In this paper, we propose several techniques to accelerate iterative processes to partition the state space of a given probabilistic model to its bisimulation classes. The first technique applies two ordering heuristics for choosing splitter blocks. The second technique uses hash tables to reduce the running time and the average time complexity of the standard iterative method. The proposed approaches are implemented and run on several conventional case studies and reduce the running time by one order of magnitude on average.*

Keywords probabilistic bisimulation, Markov decision process, model checking, splitter ordering

Citation Computer Science 26(2) 2025: 1–26

1. Introduction

Ensuring the correctness of computer systems is a crucial issue since it may jeopardize human life if specific safety systems fail. For example, a miscalculation in launching a rocket can adversely affect the whole project [14]. Testing is a promising technique to assure system correctness. Although it is a common technique, testing cannot cover the whole behavior to verify the correctness of the system [14]. Alternatively, formal methods like model checking provide a more thorough approach to ensuring system correctness. This paper focuses on model checking, which automatically checks system behaviors to confirm it meets desired properties [7].

Due to some probabilistic aspects of many computer systems, it is possible to perform a probabilistic model checking to verify the necessary properties of those systems. Markov decision processes (MDPs) and discrete-time Markov chains (DTMCs) are used to model such systems. In addition, a temporal logic or automaton is utilized to propose the required properties. A model checker automatically verifies properties in the proposed model [7, 15]. DTMCs can model probabilistic systems, while MDPs extend DTMCs with non-determinism [29]. Probabilistic computational tree logic (PCTL) is used to express properties to verify MDP and DTMC models [7].

The main challenge of model checking is the state space explosion problem; that is, by increasing the number of state variables, the size of the system state space grows exponentially [7, 29, 39]. This challenge limits the explicit state space representation to small ones. Moreover, due to the iterative numerical computations in probabilistic model checking, the running time is of importance. Hence, any attempt to alleviate state space explosion problem can also reduce the overall running time of probabilistic model checking. Various techniques have been developed in recent decades to cover this problem. Symbolic model representation [30, 39], compositional verification [20, 22], statistical model checking [3,33], and reduction techniques [25,28,31] are the most major techniques that are widely used in probabilistic model checking of DTMC and MDP models.

Bisimulation minimization is one of the model reduction techniques [4]. This approach can be applied to the verification of security protocols [1, 38, 42] and even on energy games for systems of bounded resources [8]. It defines an equivalence class on the state space of the model that can be applied to it. States of each equivalence class are called bisimilar states and satisfy the same set of properties. For a bisimulation relation, the states of any class can be collapsed to one state, and the model will be reduced to a minimized but equivalent one. The quotient model is guaranteed to meet the same set of properties and can be used by a model checker instead of the original [7]. For the case of probabilistic model checking, applying bisimulation minimization can reduce memory consumption and running time of the iterative computation methods.

Depending on the class of systems and the underlying properties, several types of bisimulation are defined in the literature. In strong bisimulation, two states, s and t, are bisimilar if and only if for each successor state of s, there is at least one

bisimilar successor state of t and vice versa [24]. Two (strongly) bisimilar models satisfy the same set of PCTL formulas [4]. In weak bisimulation, silent transitions are disregarded, and bisimilar states are defined based on a path with some silent moves and a move with the same action [12, 40]. In this paper, we focus on strong bisimulation and propose several heuristic methods to reduce the running time of iterative algorithms to compute this kind of bisimulation relation in probabilistic systems. More information about other classes of bisimulation and their algorithms is available in [4, 12].

Contributions. Briefly, the main contributions of the paper include:

- proposing a backward ordering for selecting blocks to split the states,
- proposing a size-based ordering in which the smaller a block is, the faster probabilistic bisimulation is computed,
- utilizing hash tables in order to reduce the time complexity of the current methods,
- experimentally comparing the performance of computing probabilistic bisimulation using the proposed heuristics in several case studies.

To evaluate the performance of our proposed approaches to compute probabilistic bisimulation, we implement and run them on a set of standard case study models. We also consider state of the art tools to compare the running time of our implementations with the running time of these tools. The results of these experiments demonstrate that in most cases, our proposed approaches dominate them.

Paper outline. The structure of the paper reads as follows. In Section 2, some preliminary definitions of MDPs, the PCTL logic, probabilistic bisimulation, and the standard algorithm for computing a probabilistic bisimulation partition are provided. In Section 3, the ordering heuristics are presented. Section 4 proposes the approach utilizing hash tables for improving the standard probabilistic bisimulation algorithm. Section 5 demonstrates the experimental results running on several classes of the standard benchmark models. Related works are presented in Section 6. Finally, Section 7 concludes the paper and defines some future work.

2. Preliminaries

In this section, the related definitions and algorithms used in probabilistic bisimulation are provided.

2.1. General notations and definitions

For a finite set S, a distribution μ over S is a function $\mu\colon S \to [0,1]$ such that $\sum_{s\in S} \mu(s) = 1$. We consider S as state space and call every member $s \in S$ a state of S. The set of all distributions over S is denoted by $\mathcal{D}(S)$. For any subset $T \subseteq S$ and a distribution μ, the accumulated distribution over T is defined as $\mu[T] = \sum_{s\in T} \mu(s)$.

A partition of S is a set $\mathcal{B} = \{B_i \subseteq S \mid i \in I\}$ of non-empty subsets satisfying $B_i \cap B_j = \emptyset$ for all $i, j \in I$ $(i \neq j)$, and $\cup_{i \in I} B_i = S$. The subsets $B_i \in \mathcal{B}$ are called equivalence classes. For each partition $\mathcal{B}$ of S, an equivalence relation $R_{\mathcal{B}}$ is defined where for each states $s, t \in S$, we have $s \; R_{\mathcal{B}} \; t$ *iff* there is a block $B_i \in \mathcal{B}$ where $s, t \in B_i$. For the sake of simplicity, if there is no ambiguity, we only denote it by R instead of $R_{\mathcal{B}}$. For an equivalence relation R on S, the set of equivalence classes of R are denoted by S/R. For a state $s \in S$, we define $[s]_R = \{t \in S \mid s \; R \; t\}$ and $[s]_{\mathcal{B}} = \{t \in S \mid \exists B_i \in \mathcal{B}, \;\; s, t \in B_i\}$. For a subset $T \subseteq S$, we define $T/R = \{s \in S \mid \exists t \in T \mid s \; R \; t\}$. A partition $\mathcal{B}_1$ is finer than $\mathcal{B}_2$, or $\mathcal{B}_1$ is a refinement of $\mathcal{B}_2$ *iff* for every block $B \in \mathcal{B}_1$, there is a block $B' \in \mathcal{B}_2$ with $B \subseteq B'$. One can lift an equivalence relation R on $\mathcal{D}(\mathrm{S})$ by defining $\mu R \nu$ *iff* $\mu[C] = \nu[C]$ for every block $C \in S/R$.

Definition 2.1 *A Markov decision process (MDP) [6] is a tuple (S, s_0, Act, δ, G) where S is a finite set of states, $s_0 \in S$ is an initial state, $\delta : S \times Act \to \mathcal{D}(S)$ is a (partial) probabilistic transition function, which maps a state of S and an action to a distribution of states, and $G \subseteq S$ is the set of goal states.*

MDPs are used to model systems with both nondeterministic and stochastic behavior. For each state $s \in S$, $Act(s)$ denotes the set of all enabled actions in s. Given a state $s \in S$ in an MDP M, an enabled action $a \in Act(s)$ is selected nondeterministically and according to the induced distribution $\mu = \delta(s, a)$ the next state s' is probabilistically selected. We use $|S|$ for the number of states, $|Act|$ for the number of actions, and $|M|$ for the size of the model M which is defined as $|M| = |S| \times |Act|$.

A *path* in an MDP is a sequence of states and actions of the form (s_i, a_i) where $s_i \in S$, $a_i \in Act(s_i)$, $\delta(s_i, a_i)(s_{i+1}) > 0$ for each $i \geq 0$, and s_0 is the initial state of the model. A probabilistic transition is defined as any tuple (s, a, s') where $\delta(s, a)(s') > 0$. To resolve nondeterministic choices of an MDP, the notion of policies (also called adversaries) is used. In this paper, we focus on a *memory-less policy* that maps each state $s \in S$ to an enabled action $a \in Act(s)$, hence only depends on s.

This mapping may depend on the sequence of state-actions of a path before reaching s in memory dependent policies and only depends on s in memory-less ones.

For any state $s \in S$, we use $Pre(s)$ and $Post(s)$ for the set of predecessor and successor states of s and define them as:

$$Post(s) = \{s' \in S | \exists a \in Act(s), \delta(s, a)(s') > 0\},$$

$$Pre(s) = \{s' \in S | s \in Post(s')\}.$$

For a subset of states $C \subseteq S$, $pre(C)$ is the set of predecessor states of C and defined as $Pre(C) = \cup_{s \in C} Pre(s)$.

Reachability probability properties of probabilistic systems are defined as the probability of reaching a set of states of the model. For MDPs, these properties are defined as the extremal (minimal or maximal) probability of reaching a goal state G over all possible policies.

Example 2.2 *Consider the MDP model shown in Fig. 1, which consists of 8 states including s_1 as the initial state and G as the goal state. State s_7 is a dead-end and cannot reach the goal G. When the MDP is in state s_5, it can nondeterministically choose between actions a and b. If action a is selected, the next state is determined probabilistically, with an equal probability of 0.5 of moving to either s_7 or G. In this MDP, $Pre(G) = s_4, s_5, s_6, G$, meaning these states can precede G. There is no transition from s_2 to G with a positive probability, so s_2 is not in $Pre(G)$.*

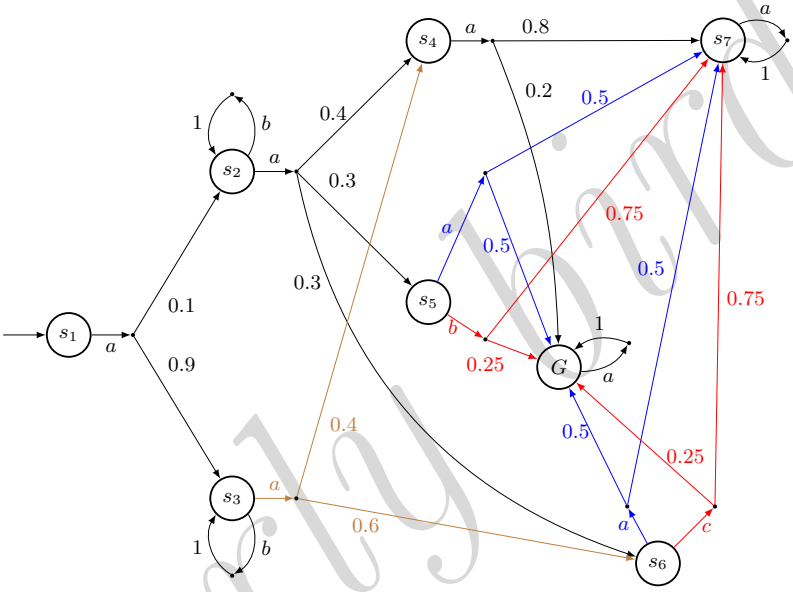

Figure 1. A simple MDP

Definition 2.3 (PCTL syntax) *Considering AP as the set of atomic propositions, the syntax of PCTL is as $\phi ::= true \mid c \mid \phi \mid \phi \wedge \phi \mid \neg\phi \mid \mathtt{P}_{\theta p}[\psi]$ where $\psi ::= \mathtt{X}\phi \mid \phi \ \mathtt{U}^{\leq k} \phi \mid \phi \ \mathtt{U} \ \phi$, in which $c \in AP$, $\theta \in \{<, \leq, >, \geq\}$, $p \in [0,1]$ and $k \in N$.*

In the PCTL syntax, ϕ is a state formula and is evaluated over the states of an MDP model M while ψ is a path formula and is evaluated over the possible paths of the model. State formulae are directly used in probabilistic model checking and path formulae can only occur inside a state formula such as $\mathtt{P}_{\theta p}[\psi]$. In the semantics of PCTL, a model state $s \in S$ satisfies the formula $\mathtt{P}_{\theta p}[\psi]$ *iff* for each possible policy of M the probability of following a path from s satisfying ψ is in the interval determined by θp [32,36]. For the path operators X (next), U (until), and $U^{\leq k}$ (bounded until), the semantics of a path formula is defined as in standard CTL [7].

Intuitively, PCTL is used to define a set of requirement properties in verification of stochastic systems. More details about probabilistic model checking of PCTL formula and iterative methods for computing reachability properties are available in [7, 22, 29].

Definition 2.4 (Probabilistic Bisimulation) *For an MDP M, an equivalence relation $R \subseteq S \times S$ is a probabilistic (strong) bisimulation for M if and only if for all pairs of states $s, t \in S$, the property $s\ R\ t$ implies that for every action $a \in Act(s)$, there is an action $b \in Act(t)$ such that $\delta(s, a)\ R\ \delta(t, b)$ [5].*

In probabilistic bisimulation, the probability of going to each block is the same for both actions. Two states $s, t \in S$ are probabilistically bisimilar (denoted by $s \simeq_p t$) if and only if there is a probabilistic bisimulation R such that $s\ R\ t$.

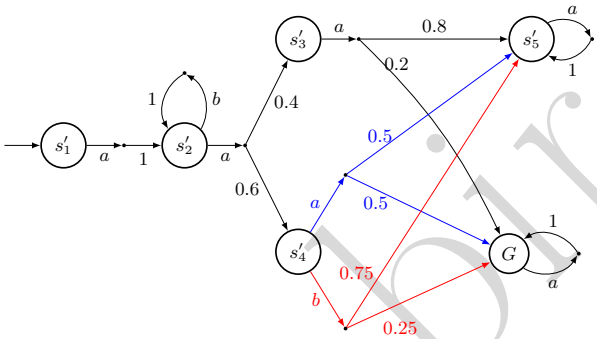

Figure 2. The probabilistic bisimular of MDP Fig. 1

Example 2.5 *Consider the MDP illustrated in Fig. 1. According to Definition 2.4, states s_5 and s_6 are bisimilar because they share identical probability distributions for each action across the state models. Specifically, in state s_5, selecting action a results in the same probability distribution for transitioning to s_7 and G as action a does in state s_6. However, states s_7 and G are not bisimilar to any other states. Due to the bisimilarity between s_5 and s_6, states s_2 and s_3 are also bisimilar. These states have a single action that leads to the same cumulative probability of reaching the bisimilar groups $\{s_4\}$ and $\{s_5, s_6\}$. The initial state s_1 is not bisimilar to any other states. Consequently, the partition $\mathcal{B}_1 = \{s'_1, s'_2, s'_3, s'_4, s'_5, G\}$ is defined, where $s'_1 = \{s_1\}, s'_2 = \{s_2, s_3\}, s'_3 = \{s_4\}, s'_4 = \{s_5, s_6\}, s'_5 = \{s_7\}$. A trivial partition is $\mathcal{B}_0 = \{\{s_1, s_2, s_3, s_4, s_5, s_6, s_7, G\}\}$. Clearly, $\mathcal{B}_1$ is finer than $\mathcal{B}_0$ because each block in $\mathcal{B}_1$ is contained within a block of $\mathcal{B}_0$. The probabilistic bisimulation for the MDP in Fig. 1 is shown in Fig. 2.*

The main characteristic of probabilistic bisimulation is that if two states $s, t \in S$ are bisimilar, then both satisfy the same set of bounded and unbounded PCTL formulae [7]. As a result, an MDP M can be replaced by a reduced bisimilar one where all bisimilar states of any block $B_i \in \mathcal{B}$ are replaced by one state. Note that in Definition 2.4, the action names are not important to decide the bisimilarity of two states [7], and a and b may be two different actions or the same. For probabilistic automata (an extension to MDPs), actions names should be considered in the computation of probabilistic bisimulation [24, 45].

2.2. The standard algorithm for computing a probabilistic bisimulation

Partition refinement is a general algorithm to compute a bisimulation relation for MDPs. Starting from an initial partition, the algorithm iteratively refines partitions by splitting some blocks into smaller (i.e. finer) ones. The iterations continue until reaching a fixed point where none of the blocks can be split anymore (Fig. 3).

Several approaches can be used to compute the first partition $\mathcal{B} \subseteq S \times S$ (Line 1 of Algorithm 2). One can consider G (the set of goal states) and $S \setminus G$ as the only two blocks of the initial partition and use G as the first splitter. This consideration follows the definition of bisimulation and is based on the fact that no goal state can be bisimilar with a non-goal state. Another option can be to consider AP to compute the first partition. In this case, two states are in the same block of the first partition *iff* they have the same AP. It also includes the first case where the goal states are considered to define the initial partition.

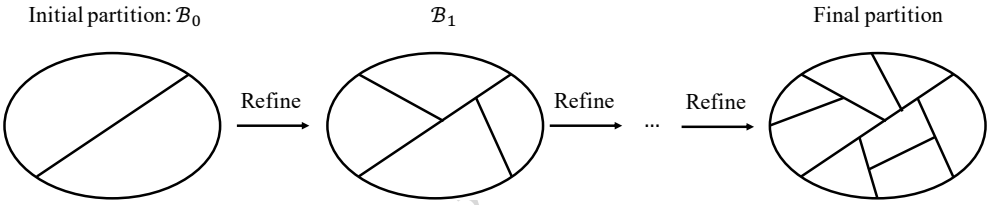

Figure 3. Successive partition refinement procedure

In each iteration, a splitter block (also called splitter in this paper) is selected to divide some of its predecessor blocks into finer ones. The way that the algorithm splits a block depends on the definition of bisimulation for the underlying MDP. Algorithm 1 presents this approach [7]. After splitting a block B_i to several finer blocks, they are considered as new splitters for the next iterations. In practice, a list can be used to keep the splitters for next computations.

Algorithm 1: Partition Refinement

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** $\mathcal{B}$;

In probabilistic bisimulation, the refinement method follows Definition 2.4 and splits the blocks by considering the probability of reaching the splitter C. This method is explained in Algorithm 2.

Algorithm 2: The *Refine* Algorithm

```
1  H]
   Input: A partition B and splitter C
   Output: A refined partition B according to splitter C
2  Initialize B to a first partition;
3  Set Q to an empty list;
4  forall B_i ∈ B s.t. B_i ∩ Pre(C) ≠ ∅ do
5      forall s ∈ B_i do
6          forall a ∈ Act(s) do
7              q = δ(s, a)[C];
8              if q ∉ Q then
9                  Add q to Q;
10     Sort elements of Q;
11     B_p = {B_i};
12     forall q ∈ Q do
13         forall B_{i,j} ∈ B_p do
14             B' = {s ∈ B_{i,j} | ∃a ∈ Act(s) : δ(s, a)[C] = q};
15             if B' ≠ B_{i,j} and B' ≠ ∅ then
16                 Remove B_{i,j} from B_p;
17                 Add B' and B_{i,j} \ B' to B_p;
18     Remove B_i from B;
19     Add members of B_p to B;
20 return B;
```

For a given splitter C, Algorithm 2 splits any block $B_i \in Pre(C)$ of the current partition $\mathcal{B}$ into k sub-blocks $B_{i,1}, B_{i,2}, \ldots, B_{i,k}$ such that

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

For a given splitter C and every predecessor block B_i, the method considers all incoming transitions to the states of C to compute $\delta(s, a)[C]$, for the related states and actions (Line 6). After computing these values, the method stores different probability values in the list Q (Lines 7-8). For each probability q and every computed sub-block $B_{i,j}$ of B_i, the set of states that can reach C via an action with the probability q is considered in B' (Line 13). Based on this computation, every sub-block $B_{i,j}$ is split into two new ones (Line 16).

Using an appropriate data structure that keeps the backward information of incoming transitions to each state, the time complexity of most parts of Algorithm 2 (except the sorting computation) is linear in the number of incoming transitions

to C [24]. On the other hand, the time complexity of sorting members of Q is in $O(|Q| \cdot \log(|Q|))$. In the worst case, for each state $s \in B_i$ and action $a \in Act(s)$ (lines 4 and 5 of Algorithm 2, we have a different probability value for $\delta(s, x)[C]$. In cases where a main part of states are in B_i, we have $|Q| \in O(|S| \times |Act|)$ and hence the time complexity becomes $O(|S| \cdot |Act| \cdot \log(|S| \cdot |Act|))$. More details about this method are available in [24].

Algorithm 1 uses a list of blocks to keep them as splitters. After refining each block, all computed sub-blocks except the largest one (based on the number of states) are added to the list to be used as potential splitters. Following this strategy, each state is considered in some splitters for at most $\log(|S|)$ times. As a result, the worst case time complexity of Algorithm 1 for computing probabilistic bisimulation is in $O(|M| \cdot (\log(|S|) + \log(|Act|)))$ [24]. For the case of DTMCs where $|Act| = 1$, this time complexity becomes $O(|M| \cdot \log(|S|))$. In the case of state space explosion problem, we may have a high running time for computing $\delta(s, a)[C]$ (in Line 6). This running time, however, can be much less than the overall running time of applying iterative numerical methods that are used for probabilistic bisimulation.

3. Ordering heuristics for choosing splitters

The probabilistic bisimulation presented in [24] randomly selects blocks of $\mathcal{B}$ as splitters. In the worst case, each state is considered $\log(|S|)$ times in the splitters. In practice, the running time of Algorithm 1 depends on the order of selecting splitter blocks. Thus, an optimal ordering may reduce the running time. In this section and as one of the contributions of our work, we propose two heuristics for splitter ordering to reduce the running time of probabilistic bisimulation. Although these heuristics do not affect the overall time complexity of computations, our experiments demonstrate that they accelerate the convergence to the final partition in many instances. In rare cases, they do not change the performance dramatically (Section 5).

3.1. Backward ordering for choosing splitters

For acyclic models, a topological ordering is defined that can decrease the running time of some standard computations in probabilistic model checking. For example, unbounded reachability probabilities or expected rewards are computed in linear time for an acyclic model if a topological ordering is used for updating state values [13]. Models with at least one non-decreasing variable are acyclic. This can happen, for example, if the model uses one or several clock variables that never reset. These models are typical in specifying probabilistic systems. For the case of cyclic models, however, it is not easy to find an optimal ordering that minimizes the running time of the iterative computations. For such models, the idea of selecting splitters according to an appropriate ordering can reduce the overall number of iterations and the running time of Algorithm 2.

Example 3.1 *Consider three blocks of a given MDP model during the partition refinement steps, as shown in Fig. 4. While block A is selected before block B in Fig. 4a,*

it splits block B into two new blocks B_{left} and B_{right} and considers the smaller one as the next splitter splits C into four blocks. On the other hand, selecting B before A splits C into two new blocks, as demonstrated in Fig. 4b. However, C_{left} and C_{right} will be split into four blocks $C_1, \ldots, C_4$ after several steps, where A splits B into B_{left} and B_{right}. In the latter case, the computations for splitting C into C_{left} and C_{right} are redundant because the same computations (as in the former case) are needed to split these two blocks into $C_1, \ldots, C_4$. These redundancies can also influence the predecessor blocks of C and bring some other redundancies in the computations of the selected blocks.

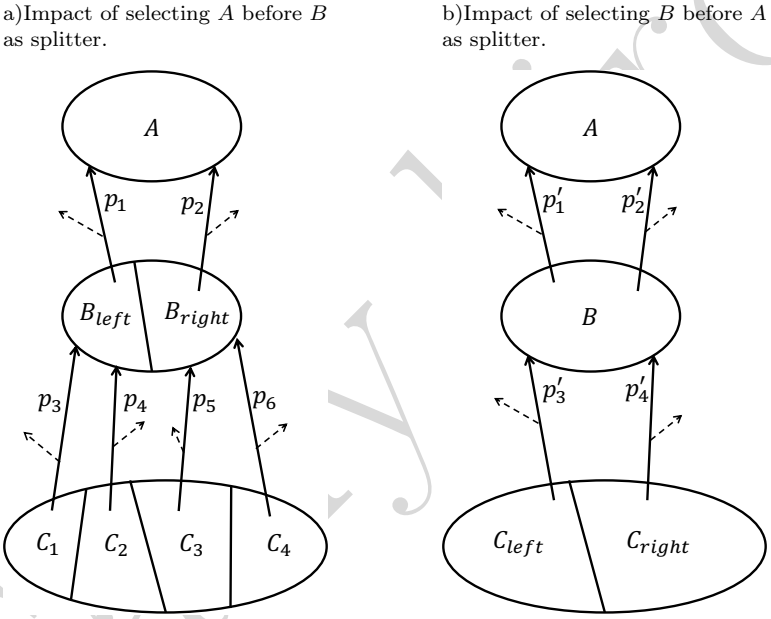

Figure 4. Topological ordering.

For an acyclic model, Algorithm 3 applies a topological ordering and computes the bisimilar partition with linear time complexity in the size of the model. This algorithm uses a list L of splitters. Each block should be added to L when all of its successor states have been previously used in some splitters (Lines 7-9). To do so, the incoming transitions to each selected splitter are marked, and a counter can be used to compute the number of unmarked outgoing transitions. Marking all outgoing transitions of a block means that no successor splitter is available for the block and all of its states are bisimilar. The algorithm adds such a block to L to use it as a splitter for its predecessor blocks. The correctness of Algorithm 3 relies on the fact that there is no cycle among the states of the bisimilar blocks of the model.

Algorithm 3: Topological Partition Refinement

Input: An MDP model M
Output: A bisimulation partition $\mathcal{B}$

```
1  Initialize B to a first partition;
2  L = {G};
3  while L is not empty do
4      Remove a splitter C from L;
5      B := Refine(B, C);
6      forall new blocks D do
7          Mark all transitions between the D and C;
8          if all outgoing transitions from D are marked then
9              Add D to L;
10 return B;
```

For cyclic models, there is no topological ordering. For this case, one may apply the SCC decomposition method [13] and selects SCCs in a right order. This technique provides an efficient ordering among SCCs of a model, but do not provide useful ordering for the states of any SCC. Hence, it is important to consider such cases, where the model may have some large SCCs and the current approaches do not utilize efficient ordering for the states of these SCCs. To cover this weakness, we provide an ordering technique to order all states of a model.

Extending the idea of Algorithm 3 to cyclic models, our first heuristic for choosing splitters is to consider the shortest path to the set of goal states to compute the initial block. It can use a queue to keep the new sub-blocks. A breadth-first search is applied (as explained in the previous section). In each step, the block B_i is defined as the set of states that have not been selected yet and have a transition to some states of predecessor block B_{i-1}. The blocks are added to the end of the queue.

In each iteration of the partition refinement procedure, a splitter block C is removed from the queue, and if no sub-block of C has been previously added, then the partition refinement Algorithm 1 uses it as a new splitter. Otherwise, C has been previously split into some sub-blocks by other splitters, and because its sub-blocks are in the queue (except the largest one), the heuristic disregards C for refinement. For any block $B_i \in Pre(C)$, all split sub-blocks $B_{i1}, B_{i2}, \ldots$, except the largest one, are added to the end of queue. This heuristic stems from using a splitter before its predecessors' blocks as much as possible. This approach can be considered as an extension to Algorithm 2 that uses a queue to select splitters in each step, whereas Algorithm 2 does not utilize any special ordering to select splitters.

3.2. Choosing smallest blocks first

In order to minimize the average number of using each state in a splitter, redundant computations should be avoided. Consider a large block C_i. It can be split by another splitter C_j into several small blocks $C_{i1}, C_{i2}, \ldots, C_{ik}$. Using C_i as a splitter,

Algorithm 2 may split several predecessor blocks into their sub-blocks. The algorithm will also split these predecessor blocks into some finer sub-blocks when it considers $C_{i1}, C_{i2}, \ldots, C_{ik}$ as splitters.

Considering $C_{i1}, C_{i2}, \ldots, C_{ik}$ instead of C_i as splitters can avoid redundant computations needed to partition predecessor blocks. To avoid such redundancies, our heuristic method prioritizes smaller blocks to be used as splitters. This heuristic utilizes the fact that larger blocks are more prone to split to smaller ones and hence, are not suitable as a splitter. It can use a priority queue to select the smallest block as the splitter at each iteration. This priority queue stores the identified and size of each splitter block and prioritize smaller ones. After splitting each block, the heuristic removes it from the queue and adds its sub-blocks to the queue.

Example 3.2 *Consider Fig. 5a. Block B is split into* B_{left} *and* B_{right} *by considering* C_i *as a splitter. In Fig. 5b, the sub-blocks* C_{i1}, C_{i2} *and* C_{i3} *are used to split block B into sub-blocks* B_1 *to* B_6*. While* B_1 *and* B_2 *are sub-blocks of* B_{left} *and* B_3 *to* B_6 *are sub-blocks of* B_{right}*, the computations of* B_{left} *and* B_{right} *are redundant and all sub-blocks* B_1 *to* B_6 *can be computed after computing* C_{i1}, C_{i2} *and* C_{i3}*.*

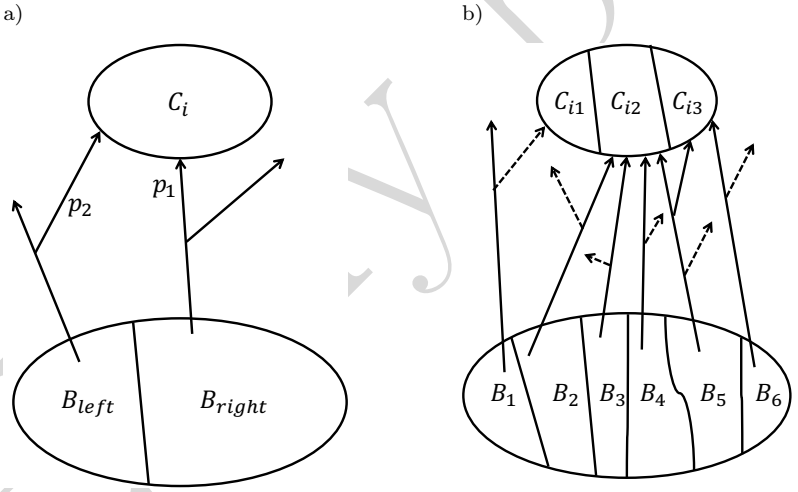

Figure 5. Size-based splitting: refinement with large splitters (a); refinement with small splitters (b)

Using *heap* as a standard data structure for a priority queue, each insert and delete operation imposes an $O(\log(n))$ extra computation that can affect the total running time of the computations. For large blocks, where the number of states is much more than $\log(|S|)$, the running time of using priority queue is negligible, and this data structure can be useful. An alternative approach for small blocks is to use several queues to keep blocks of different sizes. In this approach, we use a queue for the blocks whose size is less than or equal to $\log_2 |S|$, and a second queue for those

blocks whose size is more than $\log_2 |S|$ and less than $c \cdot \log_2 |S|$ (for some constant c). In practice, we use $c = 6$ as a heuristic that shows promising results for most case studies.

The remaining blocks are handled by the priority queue. Hence, in this alternative, we use two queues and one priority queue to store blocks. The algorithm decides to add each block to the related queue based on its size.

4. Using hash tables for improving partition refinement

For each splitter C, the standard version of Algorithm 2 (presented in [24]) sorts the members of Q to use its different members for splitting the states of the predecessor blocks. The main reason for this sorting is that the refinement algorithm for probabilistic systems need to separate the states of any block to several sub-blocks with the same probability of reaching C. These computations are required in lines 11-16 of Algorithm 2 and can be done in an efficient way if it sorts the member of Q. Using a sorted list, determining the related sub-block can be done in $O(\log |Q|)$. Applying an appropriate sorting algorithm, such as heap sort, the worst-case time complexity is in $O(|Q| \cdot \log(|Q|))$.

Consider a case where the average number of using each state as a splitter is K ($K \ll log(|S|)$), the running time of bisimulation (except sorting) becomes in $O(|M| \times K)$. If $|M| \in O(|S| \times |Act|)$ (the model is sparse), the time complexity of sorting dominates the running time of the other parts. To avoid sorting and alleviate its overhead, our approach is to use a hash table to capture different members of the set Q. In this manner, the computed value $\delta(s, a)[C]$ of each splitter C is assigned to its associated entry in the hash table, i.e., for each probability $\delta(s, a)[C]$ there is an entry in the hash table. If the entry is empty or its elements differ from $\delta(s, a)[C]$ (a collision happens), this value should be added to the set Q and also to the hash table. To have an efficient hash table and to avoid memory overhead, we consider small and almost enough digits of $\delta(s, a)[C]$, (e.g., the four first meaningful digits is a heuristic in this paper). For non-zero values, our hash function h is defined as:

$$h(\delta(s, a)[C]) = \lfloor 10^{4+j} \times \delta(s, a)[C] \rfloor,$$

where j is the number of immediate zeros after the floating point. As an example, if for some states, actions, and a splitter, we have $\delta(s, a)[C] = 0.00013760908$, then the hash function computes $h(\delta(s, a)[C]) = 1376$. For the zero value, we define $h(0) = 0$. The main aim of introducing this hash function is to have low overhead in our approach.

Considering the constant time for the computations of a hash table, the total time complexity of Lines 9-16 of Algorithm 2 reduces to $O(|Act| \cdot |S|)$ which is less than the time complexity of sorting Q, as explained in Section 2. To keep the keys with collisions, a linked list can be used for each entry; for adding a key to its related entry, it should be compared with all stored keys in the corresponding linked list. Supposing a low frequency of collisions in the hash table, any state s in predecessor

of a splitter C can be assigned to a new sub-block in constant time. Hence, the total time complexity of Algorithm 1 is in $O(|M| \times K)$.

In the worst case for each splitter C, we may have more than $\log(|Act|)$ collisions in the hash table that increase the time complexity of Algorithm 2, consequently, increasing the time complexity of Algorithm 1 to more than $O(|M| \cdot \log(|Act|))$. However, the probability of using a hash table with $|Act|$ entries is less than $2^{-\sqrt{(2 \cdot |Act| \cdot \log(|Act|))}}$. For this case, the hash function is defined as

$$h(\delta(s,a)[C]) = \lfloor |Act| \times 10^{j+4} \times \delta(s,a)[C] \rfloor.$$

Lemma 4.1 *Consider k different keys and a positive integer t. For a hash table with k entries and a function h with a uniform distribution of mapping any key to each entry, the probability of having at least $k \cdot t$ collisions is bounded by $2^{-\sqrt{2 \cdot k \cdot t}}$.*

proof. First, consider a case that some states collide with only one entry. In this case, for l keys that are mapped to the same entry, we have $0+1+2+\cdots+(l-1) = l \cdot (l-1)/2$ collisions in total. To have at least $k \cdot t$ collisions, in this case, we should have $l \cdot (l-1)/2 > k \cdot t$ that results in $l > \sqrt{2 \cdot k \cdot t}$.

Next, consider a case where some keys have at least $2 \cdot t$ collisions; that is, they are mapped to some entries that already have $2 \cdot t$ different keys with the same hash value. This condition is necessary to have at least $k \cdot t$ collisions. On the other hand, the maximum number of entries with a collision is $\frac{k}{2}$. Hence, the probability of mapping a key to an entry with more than one collision is at most $\frac{1}{2}$. As a result, the probability of having $\sqrt{2 \cdot k \cdot t}$ keys in the entries with more than one collision (as a necessary condition for having at least $k \cdot t$ collisions) is at most $2^{-\sqrt{2 \cdot k \cdot t}}$.

Using Lemma 4.1, one can control the probability of reaching the worst-case time complexity of Algorithm 1. This probability can be reduced by increasing the size of the hash table. For instance, consider a case where there are $k = 10^6$ different keys (i.e. $|Act| = 10^6$ in MDPs) and $t = 2$ as the expectation of collisions per key. This consideration makes sense according to the MDPs analyzed in experimental results and in Table 1. Applying Lemma 4.1, we have the probability of having in average more than 2 collisions per state is less than 2^{-2000}.

5. Evaluation

In this section, the performance of computing probabilistic bisimulation using the proposed heuristics in several case studies is experimentally compared (Section 5.2). Furthermore, the impact of bisimulation reduction on probabilistic model checking is investigated in Section 5.3.

5.1. Experimental setup

To show the applicability and scalability of the proposed approaches, we consider nine classes of standard models from the PRISM benchmark suite [32]. These classes include *Coin, Wlan, firewire, Zeroconf, CSMA, mer, brp, leader*, and *Israeli-Jalfon* case

studies. Except for the *brp* class, which includes DTMC models, the others are MDP ones. In *Coin*, *Zeroconf*, and *CSMA* cases, K is considered as the parameter to have different models. In *firewire*, *Wlan*, *mer*, and *Israeli-Jalfon* cases, we, respectively, consider *ddl (deadline)*, *TTM (Trans_ Time_ Max)*, n, and m as parameters. More details about these case studies are available in [6, 9, 32]. To compare our implementation of the proposed heuristics for probabilistic bisimulation with the other tools, we select two state-of-the-art tools, mCRL2 [10] and STORM [16], that provide the most recent approaches [23].

Some information on the selected models, the experimental results for our implementation, and the results for the mCRL2 and STORM tools are demonstrated in Table 1. We implemented the proposed bisimulation algorithm. The provided information includes the number of states, actions, and transitions of the original case study models and the number of states after applying the bisimulation reduction technique. The experiments have been performed on a machine running Ubuntu 22.04 LTS with Intel(R) Core(TM)i7 CPU Q720@1.6 GHz with 8 GB of memory. The results include the running time and memory consumption.

We implemented the proposed approaches as an extension of PRISM, using its sparse engine that is mainly developed in C language. The implementations and log files of experiments are available online at the GitHub repository of the paper [35].

It should be noted that the current version of PRISM does not support probabilistic bisimulation for MDPs. While PRISM and STORM use 8-bytes floating-point representation for storing probabilities, mCRL2 uses fractions of 4-bytes integers. In this way, mCRL2 compensates for the running time and memory consumption for precise computations. On the other hand, STORM follows the proposed algorithm in [5] with $O(|M| \cdot |S| \cdot (\log\ |M| + \log\ |S|))$ time complexity. It supports sparse and BDD-based data structures. For each case study, the lower running times of these two STORM engines are reported. While STORM supports the PRISM modeling language, mCRL2 follows a probabilistic labeled transition system (plts) and considers action labels to distinguish bisimilar states. For a correct comparison between mCRL2 and others, we translated the PRISM plain text models to a corresponding plts, which is executable by the mCRL2 tool. For the initial partition, our implementation considers the sets G and $S \setminus G$ as the first class of blocks. We disable graph-based pre-computations for qualitative reachability analysis in our experiments because different approaches with different impacts on the overall running times are used in the selected tools.

5.2. Performance analysis for tools and heuristics

The results of our experiments are demonstrated in Tables 1 and 2. All times are reported in seconds, and are the average of 5 runs. In Table 1, we report some information about the selected models and running the tools on them. The running times of our implementation in PRISM are based on a random splitter ordering, where the splitters are selected randomly. In our implementation and mCRL2, the reported

times include the running time for writing the result bisimilar blocks to the files. In mforost cases, our implementation outperforms both the mCRL2 and STORM tools.

Table 1
Comparing the performance of computing bisimulation in three tools for the selected MDP and DTMC models

Model name	Parameter Val	$\|S\|$ $\times 10^{-3}$	$\|Act\|$ $\times 10^{-3}$	$\|Trns\|$ $\times 10^{-3}$	$\|S\|$ after reduction	PRISM time	PRISM mem	STORM time	STORM mem	mCRL2 time	mCRL2 mem
Coin ($N = 4$)	$K = 200$	2050	5534	6918	91218	1.13	420 MB	1139	3.9 GB	9.02	2398 MB
	$K = 300$	3074	8300	10374	136818	1.52	608 MB	2223	4 GB	12.8	3672 MB
	$K = 400$	4098	11064	13830	182418	2.07	760 MB	> 1 h	4.1 GB	21	4803 MB
	$K = 500$	5122	13829	17286	228018	2.43	890 MB	> 1 h	4.2 GB	27.7	6119 MB
Coin ($N = 5$)	$K = 30$	2341	7832	9787	33825	1.71	513 MB	112	3.8 GB	18.9	3225 MB
	$K = 50$	3890	13016	16267	56325	2.98	796 MB	303	3.9 GB	31.4	5410 MB
	$K = 70$	5439	18200	22747	78825	4.1	1127 MB	554	3.9 GB	45.2	7202 MB
	$K = 100$	7762	25976	32467	112575	6.1	1620 MB	1178	4 GB	–	Killed
Coin ($N = 6$)	$K = 5$	2936	11727	14635	12212	2.88	703 MB	19.9	3.4 GB	38.9	4725 MB
	$K = 10$	5731	22924	28632	24212	5.9	1329 MB	853	309 MB	–	killed
Zeroconf ($N = 1500$)	$K = 12$	3753	6898	8467	1393850	8.13	895 MB	–	Killed	22.7	4020 MB
	$K = 14$	4426	8144	9988	1666790	10.3	962 MB	–	Killed	25.9	4843 MB
	$K = 16$	5010	9223	11307	1905323	11.9	987 MB	–	Killed	30.8	5626 MB
	$K = 18$	5476	10085	12359	2097569	13.2	1214 MB	–	Killed	37	5978 MB
	$K = 20$	5812	10711	13124	2237272	14	1530 MB	–	Killed	39.3	6522 MB
CSMA ($N = 3$)	$K = 4$	1460	1471	2397	23538	0.8	246 MB	20.3	3.9 GB	4.5	880 MB
	$K = 5$	12070	12108	20215	119440	7.96	1508 MB	143	4 GB	43.1	6834 MB
CSMA ($N = 4$)	$K = 2$	762	826	1327	9183	0.3	161 MB	9.2	3.6 GB	2.5	510 MB
	$K = 3$	8218	8516	15385	45793	5.42	1197 MB	52.9	3.9 GB	37.4	4311 MB
firewire ($dl = 3$)	$ddl = 3000$	1634	1853	1919	622127	1.24	115 MB	1532	4 GB	3.43	1376 MB
	$ddl = 10000$	5911	6711	6945	2421127	5.95	924 MB	> 1 h	4 GB	14.6	4813 MB
	$ddl = 15000$	8966	10181	10535	3706127	9.62	1715 MB	> 1 h	4.1 GB	–	Killed
firewire ($dl = 36$)	$ddl = 3000$	2238	3419	4059	999607	2.17	439 MB	2283	3992	6.9	2209 MB
	$ddl = 10000$	7670	11742	13936	3491643	9.54	1323 MB	> 1 h	4 GB	27	6030 MB
	$ddl = 15000$	11550	17687	20991	5271643	14.8	2142MB	> 1 h	4.1 GB	–	Killed
Israeli-Jalfon	$m = 17$	131	1114	19497	4112	0.4	100 MB	4.3	3.8 GB	6.6	730 MB
	$m = 18$	262	2359	4129	7685	1.05	202 MB	9	3825 MB	15.2	2840 MB
	$m = 19$	524	4981	8716	14310	2.81	401 MB	18.5	3937 MB	34.1	3288 MB
	$m = 20$	1049	10486	18350	27012	7.31	798 MB	55.6	3.9 GB	79.2	6887 MB
	$m = 21$	2097	22020	38535	50964	17.76	1620 MB	153	4 GB	–	Killed
mer	$n = 1000$	5909	22688	23273	560048	2.34	693 MB	> 1 h	3.9 GB	–	Killed
	$n = 2000$	11816	45302	46540	1120048	5.01	1217 MB	> 1 h	3.9 GB	–	Killed
	$n = 3000$	17723	68052	69807	1680048	8.2	1812 MB	> 1 h	4 GB	–	Killed
	$n = 4000$	23630	90734	93074	2240048	10.73	2447 MB	> 1 h	4 GB	–	Killed
	$n = 5000$	29537	113416	116341	2800048	12.98	2935 MB	> 1 h	4.1 GB	–	Killed
brp ($N = 400$)	$max = 150$	787	787	1087	422554	0.44	169 MB	2.9	210 MB	2.4	606 MB
	$max = 300$	1567	1567	2167	842704	1.24	292 MB	12.5	37 MB	5.1	1342 MB
	$max = 600$	3127	3127	4327	1683004	2.66	500 MB	51.4	691 MB	10.6	2732 MB
brp ($N = 800$)	$max = 150$	1573	1573	2174	844954	0.84	232 MB	6.2	373 MB	5	1371 MB
	$max = 300$	3133	3133	4334	1685104	2.13	446 MB	26.7	692 MB	10.7	2606 MB
	$max = 600$	6253	6253	8654	3365404	6.17	954 MB	103	1.3 GB	22.5	5232 MB
Wlan ($N = 5$)	$ttm = 1500$	3635	6351	7635	35768	0.27	181 MB	490	3.9 GB	8.1	3420 MB
	$ttm = 3000$	5989	11088	12372	65768	0.54	365 MB	1792	4 GB	14.4	5167 MB
	$ttm = 4500$	8345	15825	17109	95768	0.8	539 MB	> 1 h	4.1 GB	20.9	7212 MB
Wlan ($N = 6$)	$ttm = 1000$	8093	12543	17668	36006	0.75	290 MB	320	3.9 GB	–	Killed
	$ttm = 2500$	12769	21925	27051	72006	1.15	355 MB	1900	4 GB	–	Killed

In the *Coin* cases, the running time of our implementation outperforms mCRL2 by one and STORM by two orders of magnitude. In some cases, the mCRL2 process is killed due to the out-of-memory run-time errors. For example, for the case when $n = 5$ and $k = 100$, our implementation in PRISM computes the bisimulation in 6.1 seconds and consumes 1620 MB, while STORM computes the same bisimulation in

1178 seconds and consumes 4 GB memory, and mCRL2 is killed by out-of-memory error.

In *Zeroconf* models, the running time and memory consumption of our implementation are around 50% and 20% of the ones for mCRL2, respectively. In these models, STORM encounters the memory exception.

For all the reported *CSMA* models, our implementation is faster than mCRL2 and STORM. However, for larger values of the parameter K (that are not reported here), all tools terminate because of memory limitation. In *firewire* and *Wlan* models, our implementation outperforms STORM by around three orders of magnitude. For large models of these two classes, mCRL2 terminated because of memory limitation, but STORM is able to continue the computations in its BDD-based engine. For *Israeli-Jalfon* models, STORM outperforms mCRL2 in both running-time and memory computations for large models. For the *mer* models, our implantation proposes promising results, while the STORM model checker needs more than one hour for all cases. The mCRL2 tool encounters the segmentation fault error for our given plain-text models.

In most cases, the memory consumption of the proposed implementation in PRISM is less than the other tools, e.g., STORM reports around 4 GB as the peak of memory consumption regardless of the size of the models (except the *brp* ones). For the PRISM and mCRL2, memory consumption depends on the size of the models. Because mCRL2 uses integer value fractions to store probability values, it consumes more memory than the PRISM implementation, and in some cases, it terminates due to memory exceptions while PRISM can compute the bisimulation.

In Table 2, we compare the impact of our proposed methods implemented in PRISM on the performance of the probabilistic bisimulation algorithm. We propose the running time and memory overhead of our methods for the selected case study models. The running times only include the computations of bisimulation blocks, excluding the time for writing reduced models to the files and computing quantitative properties. Moreover, the average number of using each state in a splitter is considered as a criterion to compare the performance of the applied methods. Let *SPLITTERS* be the set of all blocks that are used as a splitter during partition refinement computations (Algorithm 2). We define the average number of using each state in a splitter as $SplAvg = (\sum_{C \in SPLITTERS} |C|)/|S|$ and report this value for each approach in the table.

We recall the running times for our implementation of the proposed method in [24] based on a random ordering for selecting splitter blocks in the column "Random ordering". For the proposed ordering heuristics, memory overhead includes the maximum extra memory usage for keeping the states in the related lists and priority queues. In our implementation, the hash tables contain 10000 entries, which need at least 40 KB of memory to point to the related elements. Because each splitter C is used only once during the computations, our approach needs one hash table for all computations.

Table 2

Comparing the impact of the proposed heuristics implemented in PRISM on the performance of the partition refinement algorithm for the selected MDP classes

Model name	Parameter Val	random ordering		topological ordering			size-based ordering			hash-table	
		running time	AvgSpl	running time	memory overhead	AvgSpl	running time	memory overhead	AvgSpl	running time	AvgSpl
Coin ($N = 4$)	$K = 200$	0.42	2.43	0.38	2.3 KB	1.69	0.22	245 KB	1.02	0.22	1.02
	$K = 300$	1.32	2.48	0.98	2.3 KB	1.69	0.66	373 KB	1.02	0.66	1.02
	$K = 400$	1.77	2.48	1.73	2.3 KB	1.69	0.9	497 KB	1.02	0.86	1.02
	$K = 500$	2.26	2.49	1.65	2.4 KB	1.69	1.14	621 KB	1.02	1.09	1.02
Coin ($N = 5$)	$K = 30$	1.55	2.42	1.35	17.8 KB	1.99	1	80 KB	1.14	0.87	1.14
	$K = 50$	2.72	2.52	2.12	17.8 KB	1.99	1.45	132.3 KB	1.14	1.3	1.14
	$K = 70$	3.72	2.5	2.92	17.8 KB	2	2.05	186.5 KB	1.14	1.85	1.14
	$K = 100$	5.53	2.51	4.19	17.8 KB	2	2.96	266.3 KB	1.14	2.62	1.14
Coin ($N = 6$)	$K = 5$	2.64	2.31	3.1	135.7 KB	2.13	1.82	40.3 KB	1.13	1.57	1.14
	$K = 10$	5.53	2.38	6.36	135.7 KB	2.17	3.73	80.2 KB	1.14	3.16	1.14
Zeroconf ($N = 1500$)	$K = 12$	7.7	3.09	3.15	206 KB	2.5	1.4	9.67 MB	0.7	1.32	0.72
	$K = 14$	9.8	3.1	3.9	218 KB	2.54	1.58	11.5 MB	0.73	1.49	0.73
	$K = 16$	11.8	3.14	4.5	227 KB	2.58	1.39	13 MB	0.73	1.79	0.73
	$K = 18$	12.6	3.04	5.03	233 KB	2.6	2.11	14.1 MB	0.72	2.04	0.72
	$K = 20$	13.3	3.36	5.38	239 KB	2.6	2.26	15.7 MB	0.7	2.15	0.72
CSMA ($N = 3$)	$K = 4$	0.7	2.36	0.66	0.27 MB	2.47	0.38	85.8 KB	1.05	0.38	1.06
	$K = 5$	7.06	2.55	7.44	2.48 MB	2.2	5.11	425 KB	1.26	5.87	1.26
CSMA ($N = 4$)	$K = 2$	0.25	2.09	0.28	53.5 KB	2.13	0.18	31.9 KB	1	0.16	1
	$K = 3$	4.87	2.27	5.6	931 KB	2.66	3.22	160 KB	1.09	3.03	1.1
firewire ($dl = 3$)	$ddl = 3K$	3.4	2.33	2.91	0.51 MB	1.64	2	2.29 MB	0.65	1.87	0.65
	$ddl = 10K$	5.4	2.33	4.89	1.88 MB	1.64	2.01	9.43 MB	0.65	1.87	0.65
	$ddl = 15K$	8.74	2.39	7.95	1.95 MB	1.64	3.26	14.5 MB	0.63	3.01	0.64
firewire ($dl = 36$)	$ddl = 3K$	1.97	2.71	1.6	2.05 MB	1.88	0.64	3.52 MB	0.67	0.59	0.67
	$ddl = 10K$	8.76	2.82	7.27	7.12 MB	1.89	2.8	12.5 MB	0.66	2.62	0.66
	$ddl = 15K$	13.6	2.79	11.6	10.7 MB	1.89	4.43	34.7 MB	0.66	4.25	0.66
Israeli-Jalfon	$m = 17$	0.4	2.12	0.42	0.15 MB	2.17	0.32	19.3 KB	0.97	0.2	0.97
	$m = 18$	1.03	2.09	1.15	0.32 MB	2.24	0.79	35.8 KB	0.97	0.53	0.97
	$m = 19$	2.78	2.08	2.85	0.64 MB	2.25	2.01	66.6 KB	0.97	1.46	0.97
	$m = 20$	7.24	2.1	7.43	1.26 MB	2.3	5.49	124.6 KB	0.97	3.99	0.97
	$m = 21$	17.6	2.1	18.6	2.50 MB	2.28	13	234 KB	0.98	9.94	0.98
mer	$n = 1000$	1.92	1.39	1.85	14.9 MB	1.23	0.8	1.42 MB	0.96	0.81	0.96
	$n = 2000$	4.13	1.25	4.19	29.9 MB	1.23	1.64	2.86 MB	0.96	1.68	0.96
	$n = 3000$	6.87	1.29	6.64	44.9 MB	1.24	2.54	4.29 MB	0.96	2.53	0.96
	$n = 4000$	8.96	1.18	9.23	59.86 MB	1.23	3.5	5.73 MB	0.96	3.43	0.96
	$n = 5000$	10.79	1.33	11.78	74.84 MB	1.23	4.31	7.16 MB	0.96	4.32	0.96
brp ($N = 400$)	$max = 150$	0.34	2.35	0.31	-	1.64	0.26	-	0.64	0.34	0.64
	$max = 300$	0.98	2.33	0.61	-	1.65	0.54	-	0.63	0.67	0.63
	$max = 600$	2.31	2.32	1.4	-	1.65	1.28	-	0.63	1.29	0.63
brp ($N = 800$)	$max = 150$	0.67	2.45	0.54	-	1.65	0.48	-	0.55	0.46	0.55
	$max = 300$	1.68	2.38	1.22	-	1.65	1.06	-	0.55	1.05	0.55
	$max = 600$	5.46	2.36	3.5	-	1.65	2.95	-	0.55	2.84	0.55
Wlan ($N = 5$)	$TTM = 1500$	0.09	1.03	0.09	10.9 MB	1.01	0.08	90.5 KB	0.99	0.08	0.99
	$TTM = 3000$	0.14	1.03	0.15	12.4 MB	1.01	0.14	172 KB	0.99	0.15	0.99
	$TTM = 4500$	0.23	1.02	0.21	15.3 MB	1.01	0.21	270.5 KB	0.99	0.23	0.99
Wlan ($N = 6$)	$TTM = 1000$	0.22	1.01	0.17	26.5 MB	1	0.17	75.1 KB	1	0.18	1
	$TTM = 2500$	0.29	1.01	0.28	38.8 MB	1	0.26	183 KB	1	0.33	1

In most cases, our proposed methods reduce the running time of the original probabilistic bisimulation algorithm. Experimental results are promising for size-based ordering where, in most cases, reduce the running time to half or less compared to the random ordering approach.

The topological ordering approach reduces the running time of bisimulation minimization for most cases of *Zeroconf*, *firewire*, *brp*, and *Coin* models, although it is not as good as the size-based approach. For these cases, the value of *SplAvg* is reduced when the topological ordering is used for selecting the splitters. For most

models of the *CSMA* and *Israeli-Jalfon* cases, the value of *SplAvg* is increased when our topological-based approach is applied. For the *Mer* and *Wlan* cases, the value of *SplAvg* and the running times are near each other for the random and topological ordering approaches. While for most cases, *SplAvg* is more than two, this value is near one for the *Mer* and *Wlan* cases. For these two classes, a large part of states are either in G, or cannot reach this set. Hence, a small part of S remains for the iterative partition refinement computations. The size-based ordering method reduces the value of *SplAvg* to around one or less in most cases. It shows that the running time of the partition refinement approach with this heuristic is approximately linear in the size of MDP models and one can expect to have similar performance of the larger models of the selected case studies. On the other hand, this value is independent of the size of MDPs among the same models of most classes. The only exception is for the *CSMA* models. As a result, the running time of the bisimulation method with size-based ordering is near linear in the size of models for the selected benchmark sets.

The proposed technique in Section 4 for using the hash table to improve the time complexity of the bisimulation algorithm proposes a slight improvement in practice in the performance of the bisimulation algorithm. Note that the main benefit of using a hash table is to reduce the probability of reaching the worst-case time complexity of Algorithm 1 for MDP models, as is described in Lemma 4.1. However, this method may work faster or slower than the others in practice. In all cases, memory overheads are less than 100 MB and less than 5% of the memory consumption of the original implementation, which shows that the proposed heuristics are completely feasible. While the size-based ordering approach is faster than the topological ordering, for some cases, its memory overhead is more than memory overhead of the topological ordering approach, and for other cases, it is less. The main reason that the memory overhead of these approaches is different from one class to the other is that it depends on the structure of the model and the maximum number of splitters that are kept in the related list or priority queue. In all cases, the memory overhead of applying hash table is less than 50 KB because of rare collisions in its entries in our experiments.

5.3. Impact of bisimulation in probabilistic model checking

To study the impact of bisimulation reduction on the running time of probabilistic model checking, we consider seven classes of case study models, including *Coin*, *Zeroconf*, *Israeli-Jalfon*, *firewire*, *Wlan*, *mer*, and *brp*. For each class, we consider a set of models by setting different values to their parameters. For models of each class, we follow three approaches: (1) running the standard probabilistic model checking without bisimulation, (2) running with the random ordering bisimulation, and (3) running with size-based bisimulation. For an iterative computation method to approximate reachability probabilities, the Gauss–Seidel version of the value iteration method is used. The results of these experiments are demonstrated in Figures 6–7. In each figure, the vertical axis shows the parameter value for the models, and the horizontal axis determines the running time in seconds.

Figure 6 for the *Israeli-Jalfon* class shows that the running times are reduced to around half when the bisimulation minimization is applied. In this case, the bisimulation method reduces the size of models to less than 10% of the original models. Size-based bisimulation reduces around 25% of the overall running times comparing to the random ordering approach.

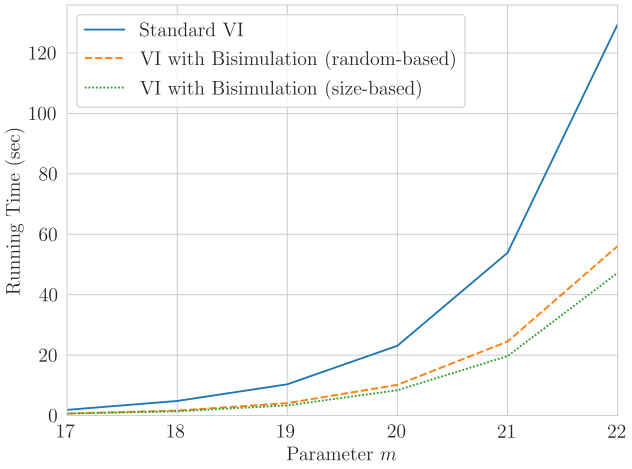

Figure 6. Standard model checking vs. applying bisimulation for the *Israeli-Jalfon* models in seconds

The best results are for the class of *Coin* models in Figure 7a. In this case, the running time of computing bisimulation is negligible compared to the running time of the other numerical computations, and applying this minimization approach reduces the overall running times by one order of magnitude. The number of states of these models after applying bisimulation minimization is less than 10% of the number of states of the original models, as reported in Table 2. In this case, due to the significant difference between running time of model checking with/without probabilistic bisimulation, the overall times using different splitter ordering are not depicted in the figure.

In the case of *Zeroconf* (see Fig. 7b), there is a meaningful difference among the applied bisimulation heuristics. Using bisimulation with the random splitter ordering, the computation overhead is more than its benefit, and the overall running times are increased. In this class, iterative computations for reachability probabilities converge fast. Using the proposed size-based heuristic improves the performance such that the running time of applying standard value iteration on the original *Zeroconf* models is near the overall running time of computing bisimulation and applying value iteration on the reduced models. For this class, the size of the reduced models is around 40% of the size of the original ones, which prohibits bisimulation from reducing the overall running time.

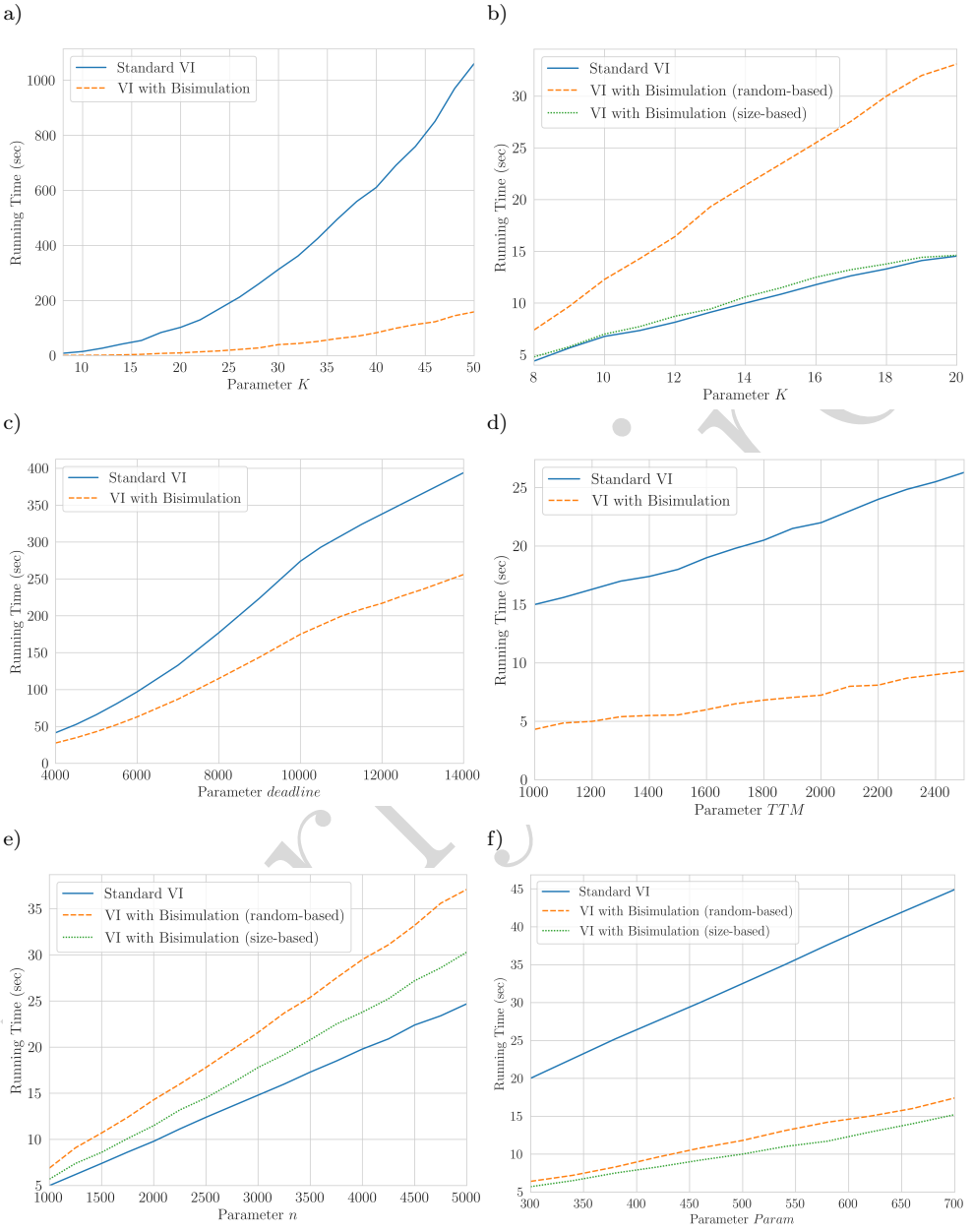

Figure 7. The running time of model checking with/without applying bisimulation: a) *Coin* models; b) *Zeroconf* models; c) *firewire* models; d) *Wlan* models; e) *Mer* models; f) *brp* models

In Figure 7c for the set of *firewire* models, bisimulation reduces overall running times by 30%. Because the running times of the iterative computations are high, there is no significant difference in the overall times when using different splitter ordering approaches. The results of our experiments for the *Wlan* class of models are proposed in Figure 7d. Although bisimulation results in a significant reduction in the model in these cases, the overall running times are reduced to less than 40% after applying bisimulation because a large part of the states are in the goal states G.

The results for the *Mer* cases show that the overall running time is increased when the bisimulation minimization methods are applied (Fig. 7e). For this class, as provided in the logs of the repository of the codes, more than 60% of states are in G and disregarded for the iterative computations.

In Figure 7f for the *brp* DTMC models, applying bisimulation reduces the running times to less than 30% of the running time of the standard iterative method without using bisimulation. Although the number of states of the reduced models is around 50% of the number of states of the original ones, applying bisimulation reduces the number of iterations for computing reachability probabilities.

As a meta-heuristic, size-base ordering is preferred than topological ordering. Furthermore, hash-table approach is more efficient where the model is more dense or the transition probability functions have various distributions.

6. Related works

Several techniques have been proposed to compute probabilistic bisimulation in the literature. The first work to define bisimulation for MDPs returns to [34, 44]. Definitions of strong and weak bisimulation for probabilistic systems with non-determinism and their related algorithms were first proposed in [45]. Baier et al. proposed an iterative algorithm for computing probabilistic bisimulation with a time complexity of $O(|M| \cdot |S| \cdot \log(|M| \cdot |S|))$, where $|S|$ is the number of states and $|M|$ is the number of transitions of the model [5]. Although several algorithms have been developed for other types of probabilistic systems (such as discrete-time and continuous-time Markov chains or Markov reward models), they rarely consider nondeterministic systems with probabilistic transitions. An efficient algorithm with $O(|M| \cdot \log(|M| + |S|))$ time complexity has been proposed in [24]. It is a special case of a more generic partition refinement algorithm, proposed in [18], that has the $O(|M| \cdot \log(|M| + |S|))$ time complexity. This generic algorithm can be used for a wide class of transition systems, including nondeterministic and probabilistic ones. Also recently, explicit Hopcroft's tricks are applied on categorical partition refinement [43]. In [27], an efficient implementation and tool are presented to handle large automata for co-algebraic bisimilarity minimization. A scalable method has been proposed in [11] that defines a bisimulation metric to approximate the bisimulation relation on a given MDP model. Based on the provided metric, any two states that are close to each other are considered bisimilar and are in the same equivalent class. The computed partition may or may not coincide with the exact bisimulation relation.

To avoid storing the entire state space and transitions of a model, several symbolic bisimulation approaches have been proposed and implemented [17, 26]. The STORM model checker employs a decision diagram-based data structure as a standard symbolic approach and reports promising results in reducing the running time and consumed memory [26]. However, symbolic approaches for bisimulation minimization bring several challenges that may influence their performance [21]. STORM also supports the multi-core bisimulation minimization approach to accelerate the computation of bisimilar partitions [19].

Due to the increasing application of machine learning, recently, some approaches are developed to use machine learning in probabilistic bisimulation. In [37], the authors use support vector machine to classify the state space of a given MDP model to its bisimulation classes. In [2], a learning-based approach for bisimulation is proposed that can be used on very large transitions systems. It applies a data-driven technique to compute bisimilar blocks from sample states and transitions of a given system.

7. Conclusion

In this paper, two approaches to improve the performance of the standard algorithms for computing probabilistic bisimulation in MDPs were proposed. In the first approach, two heuristics, including topological and size-based ordering, were proposed to determine the ordering of splitters. The second approach used hash tables to reduce the number of comparisons for splitting the blocks of states. Experimental results demonstrated that, in the majority of cases, our approaches outperform the previous algorithms and the other state-of-the-art tools. The impact of bisimulation minimization on the running time of probabilistic model checking was reported as well. For future work, the applicability of the proposed techniques can be studied on other classes of transition systems, such as probabilistic automata or continuous-time Markov chains. Also, we aim to apply these approaches to other fields, for example, protocol security and probabilistic programs [41].

Disclosure statement

The authors declare that they have no conflict of interest.

References

[1] Abadi M., Gordon A.D.: A bisimulation method for cryptographic protocols. In: *European Symposium on Programming*, pp. 12–26, Springer, 1998.

[2] Abate A., Giacobbe M., Schnitzer Y.: Bisimulation Learning, 2024. https://arxiv.org/abs/2405.15723.

[3] Agha G., Palmskog K.: A survey of statistical model checking, *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 28(1), pp. 1–39, 2018.

[4] Baier C., DArgenio P.R., Hermanns H.: On the probabilistic bisimulation spectrum with silent moves, *Acta Informatica*, vol. 57(3), pp. 465–512, 2020.

[5] Baier C., Engelen B., Majster-Cederbaum M.: Deciding bisimilarity and similarity for probabilistic processes, *Journal of Computer and System Sciences*, vol. 60(1), pp. 187–231, 2000.

[6] Baier C., Hermanns H., Katoen J.P.: The 10,000 facets of MDP model checking. In: *Computing and Software Science*, pp. 420–451, Springer, 2019.

[7] Baier C., Katoen J.P.: *Principles of model checking*, MIT press, 2008.

[8] Bisping B.: Process Equivalence Problems as Energy Games, *arXiv preprint arXiv:230308904*, 2023.

[9] Budde C.E., Hartmanns A., Klauck M., Křetínskỳ J., Parker D., Quatmann T., Turrini A., Zhang Z.: On correctness, precision, and performance in quantitative verification: QComp 2020 competition report. In: *International Symposium on Leveraging Applications of Formal Methods*, pp. 216–241, Springer, 2020.

[10] Bunte O., Groote J.F., Keiren J.J., Laveaux M., Neele T., de Vink E.P., Wesselink W., Wijs A., Willemse T.A.: The mCRL2 toolset for analysing concurrent systems. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 21–39, Springer, 2019.

[11] Castro P.S.: Scalable methods for computing state similarity in deterministic Markov decision processes. In: *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34(06), pp. 10069–10076, 2020.

[12] Cattani S., Segala R.: Decision algorithms for probabilistic bisimulation. In: *International Conference on Concurrency Theory*, pp. 371–386, Springer, 2002.

[13] Ciesinski F., Baier C., Größer M., Klein J.: Reduction techniques for model checking Markov decision processes. In: *2008 Fifth International Conference on Quantitative Evaluation of Systems*, pp. 45–54, IEEE, 2008.

[14] Clarke E.M., Henzinger T.A., Veith H.: Introduction to model checking, *Handbook of Model Checking*, pp. 1–26, 2018.

[15] Clarke E.M., Henzinger T.A., Veith H., Bloem R., *et al.*: *Handbook of model checking*, vol. 10, Springer, 2018.

[16] Dehnert C., Junges S., Katoen J.P., Volk M.: A STORM is coming: A modern probabilistic model checker. In: *International Conference on Computer Aided Verification*, pp. 592–600, Springer, 2017.

[17] Dehnert C., Katoen J.P., Parker D.: SMT-based bisimulation minimisation of Markov models. In: *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pp. 28–47, Springer, 2013.

[18] Deifel H.P., Milius S., Schröder L., Wißmann T.: Generic partition refinement and weighted tree automata. In: *International Symposium on Formal Methods*, pp. 280–297, Springer, 2019.

[19] van Dijk T., van de Pol J.: Multi-core symbolic bisimulation minimisation, *International journal on software tools for technology transfer*, vol. 20(2), pp. 157–177, 2018.

[20] Feng L., Kwiatkowska M., Parker D.: Compositional verification of probabilistic systems using learning. In: *2010 Seventh International Conference on the Quantitative Evaluation of Systems*, pp. 133–142, IEEE, 2010.

[21] Fisler K., Vardi M.Y.: Bisimulation minimization and symbolic model checking, *Formal Methods in System Design*, vol. 21(1), pp. 39–78, 2002.

[22] Forejt V., Kwiatkowska M., Norman G., Parker D.: Automated verification techniques for probabilistic systems. In: *International school on formal methods for the design of computer, communication and software systems*, pp. 53–113, Springer, 2011.

[23] Garavel H., Lang F.: Equivalence Checking 40 Years After: A Review of Bisimulation Tools, *A Journey from Process Algebra via Timed Automata to Model Learning*, pp. 213–265, 2022.

[24] Groote J.F., Rivera Verduzco J., De Vink E.P.: An efficient algorithm to determine probabilistic bisimulation, *Algorithms*, vol. 11(9), p. 131, 2018.

[25] Hansen H., Kwiatkowska M., Qu H.: Partial order reduction for model checking Markov decision processes under unconditional fairness. In: *2011 Eighth International Conference on Quantitative Evaluation of SysTems*, pp. 203–212, IEEE, 2011.

[26] Hensel C., Junges S., Katoen J.P., Quatmann T., Volk M.: The probabilistic model checker STORM, *International Journal on Software Tools for Technology Transfer*, vol. 24(4), pp. 589–610, 2022.

[27] Jacobs J., Wißmann T.: Fast coalgebraic bisimilarity minimization, *Proceedings of the ACM on Programming Languages*, vol. 7(POPL), pp. 1514–1541, 2023.

[28] Kamaleson N.: *Model reduction techniques for probabilistic verification of Markov chains*, Ph.D. thesis, University of Birmingham, 2018.

[29] Katoen J.P.: The probabilistic model checking landscape. In: *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, pp. 31–45, 2016.

[30] Klein J., Baier C., Chrszon P., Daum M., Dubslaff C., Klüppelholz S., Märcker S., Müller D.: Advances in symbolic probabilistic model checking with PRISM. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 349–366, Springer, 2016.

[31] Kwiatkowska M., Norman G., Parker D.: Symmetry reduction for probabilistic model checking. In: *International Conference on Computer Aided Verification*, pp. 234–248, Springer, 2006.

[32] Kwiatkowska M., Norman G., Parker D.: PRISM 4.0: Verification of probabilistic real-time systems. In: *International conference on computer aided verification*, pp. 585–591, Springer, 2011.

[33] Larsen K.G., Legay A.: Statistical model checking: Past, present, and future. In: *International Symposium on Leveraging Applications of Formal Methods*, pp. 3–15, Springer, 2016.

[34] Larsen K.G., Skou A.: Bisimulation through probabilistic testing, *Information and computation*, vol. 94(1), pp. 1–28, 1991.

[35] Mohagheghi M.: Probabilistic Bisimulation, https://github.com/sadeghrk/prism/tree/improved-bisimulation, Accessed: 2024-12-01.

[36] Mohagheghi M., Karimpour J., Isazadeh A.: Improving modified policy iteration for probabilistic model checking, *Computer Science*, vol. 23(1), 2022.

[37] Mohagheghi M., Salehi K.: Improving Probabilistic Bisimulation for MDPs Using Machine Learning, *Mathematics Interdisciplinary Research*, vol. 9(2), pp. 151–169, 2024. doi: 10.22052/mir.2023.253367.1431.

[38] Noroozi A.A., Karimpour J., Isazadeh A.: Bisimulation for Secure Information Flow Analysis of Multi-Threaded Programs, *Mathematical and Computational Applications*, vol. 24(2), 2019.

[39] Parker D.A.: *Implementation of symbolic model checking for probabilistic systems*, Ph.D. thesis, University of Birmingham, 2003.

[40] Philippou A., Lee I., Sokolsky O.: Weak bisimulation for probabilistic systems. In: *International Conference on Concurrency Theory*, pp. 334–349, Springer, 2000.

[41] Salehi K., Noroozi A.A., Amir-Mohammadian S.: Quantifying information leakage of probabilistic programs using the PRISM model checker. In: *Proceedings of the 15th International Conference on Emerging Security Information, Systems and Technologies (SECURWARE 2021)*, pp. 47–52, 2021.

[42] Salehi K., Noroozi A.A., Amir-Mohammadian S., Mohagheghi M.: An Automated Quantitative Information Flow Analysis for Concurrent Programs. In: *International Conference on Quantitative Evaluation of Systems*, pp. 43–63, Springer, 2022.

[43] Sanada T., Kojima R., Komorida Y., Muroya K., Hasuo I.: Explicit Hopcroft's Trick in Categorical Partition Refinement, *arXiv preprint arXiv:230715261*, 2023.

[44] Segala R.: *Modeling and verification of randomized distributed real-time systems*, Ph.D. thesis, Massachusetts Institute of Technology, 1995.

[45] Stoelinga M.I.A.: *verification of probabilistic, real-time and parametric systems*, Ph.D. thesis, Radboud University, Nijmegen, 2002.

Affiliations

MohammadSadegh Mohagheghi
Vali-e-Asr University of Rafsanjan, Department of Computer Science, Rafsanjan, Iran, mohagheghi@vru.ac.ir

Khayyam Salehi
Shahrekord University, Department of Computer Science, Shahrekord, Iran, kh.salehi@sku.ac.ir

Received: 8.07.2024
Revised: 6.09.2024
Accepted: 21.09.2024