Symblicit Exploration and Elimination for Probabilistic Model Checking

Ernst Moritz Hahn
University of Twente
Enschede, The Netherlands
e.m.hahn@utwente.nl

Arnd Hartmanns
University of Twente
Enschede, The Netherlands
a.hartmanns@utwente.nl

ABSTRACT

Binary decision diagrams can compactly represent vast sets of states, mitigating the state space explosion problem in model checking. Probabilistic systems, however, require multi-terminal diagrams storing rational numbers. They are inefficient for models with many distinct probabilities and for iterative numeric algorithms like value iteration. In this paper, we present a new "symblicit" approach to checking Markov chains and related probabilistic models: We first generate a decision diagram that *symb*olically collects all reachable states and their predecessors. We then concretise states one-by-one into an exp*licit* partial state space representation. Whenever all predecessors of a state have been concretised, we *eliminate* it from the explicit state space in a way that preserves all relevant probabilities and rewards. We thus keep few explicit states in memory at any time. Experiments show that very large models can be model-checked in this way with very low memory consumption.

ACM Reference Format:
Ernst Moritz Hahn and Arnd Hartmanns. 2021. Symblicit Exploration and Elimination for Probabilistic Model Checking. In *The 36th ACM/SIGAPP Symposium on Applied Computing (SAC '21), March 22–26, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3412841.3442052

1 INTRODUCTION

Many of the complex systems that we are surrounded by, rely on, and use every day are inherently probabilistic: The Internet is built on randomised algorithms such as the collision avoidance schemes in Ethernet and wireless protocols, with the latter additionally being subject to random message loss. Hard- and software in cars, trains, and aeroplanes is designed to be fault-tolerant based on mean-time-to-failure statistics and stochastic wear models. Machine learning algorithms give recommendations based on estimates of the likelihoods of possible outcomes, which in turn may be learned from randomly sampled data.

Given a formal mathematical model of such a system, e.g. in the form of (a high-level description of) a discrete- or continuous-time Markov chain (DTMC or CTMC), probabilistic model checking can automatically compute (an approximation of) the value of a quantity of interest. Such quantities include the probability to finally reach an unsafe state (a measure of reliability), the steady-state probability to be in a failure state (determining availability), the long-run average reward (measuring e.g. throughput or energy consumption), or the accumulated cost up to a certain set of states (e.g. until a job is complete). The standard approach is to proceed in two phases: First, *explore* the state space, building a representation of the set of reachable states and the transitions connecting them. Transitions are annotated with rational values for probabilities and rewards, which are usually represented as floating-point numbers. Second, use an iterative numeric algorithm such as value iteration [36] or one of its sound variants [5, 21, 30, 37] to *compute* the quantity of interest. These algorithms in fact compute a value for every state that approximates the quantity starting from that state up to a prescribed error ϵ. In contrast to classic functional model checking, which admits on-the-fly algorithms for e.g. reachability or LTL properties, probabilistic model checking is thus doubly affected by the state space explosion problem: First, the entire state space must be stored in memory, including many numeric values. Second, the numeric computation requires multiple values to be stored, and updates to be performed on them, for all states.

Current approaches to mitigate the state space explosion problem in probabilistic model checking include the use of partial exploration and learning algorithms, bisimulation minimisation, and compact representations of the state space or value vectors by binary decision diagrams (BDDs). They exploit different structural properties that only sometimes overlap. The learning-based approaches [1, 8] for reachability probabilities work well for models where a small initial subset of the state space determines most of the probability mass. In such cases, which are not abundant among existing case studies [24], they complete in a few seconds while exhaustive approaches run out of time or memory [11, Table 1]. Bisimulation minimisation reduces the state space to a quotient according to a probabilistic bisimulation relation; see [3, Sect. 5.1] for an overview. It has been implemented in Storm [15] and allows certain very large models to be checked efficiently; in general, its impact depends on the amount of bisimilar states in the given system. Finally, BDDs [9, 34] have a long history of use in (discrete-state) model checking [12] to compactly represent state spaces, in good cases reducing memory usage by orders of magnitude. They work well when the state space is structured and exhibits symmetry, which is often the case for real-life case studies modelled by humans (as opposed to randomly generated examples). In probabilistic model checking, however, numeric values from continuous domains are part of state spaces and must be encoded in the decision diagrams. A binary encoding of their floating-point representation does not usually result in compact diagrams; instead,

SAC '21, March 22–26, 2021, Virtual Event, Republic of Korea

ACM ISBN 978-1-4503-8104-8/21/03…$15.00
https://doi.org/10.1145/3412841.3442052

multi-terminal BDDs (MTBDDs), where each of the (unbounded number of) leaves represents one number, have been applied with some success, notably in the probabilistic model checker Prism [33]. They however do not provide much compaction for models with many distinct probabilities or reward values due to the large number of leaves. They also do not work well to represent the large vectors of values used in iterative numeric algorithms such as value iteration, which progress through many very different intermediate values for each state before converging to, but often not reaching, a fixpoint. For this reason, Prism defaults to its *hybrid* engine, which uses MTBDDs for the state space but arrays of double-precision values for iteration. Its fully symbolic *mtbdd* engine only solves specific large structured models in reasonable runtime.

Our contribution is a new approach that combines (MT)BDDs, explicit state representations, and state elimination to tackle the problem of model-checking large probabilistic specifications. Its novelty lies in (1) using BDDs precisely for those tasks that they work well for, and (2) using state elimination instead of the standard iterative algorithms in the computation phase. We work with discrete-state probabilistic systems; in this paper, we use DTMC to explain our approach, but the same techniques apply directly to CTMC, too, and may be extended to Markov decision processes (MDP) [36] and Markov automata (MA) [16] using the extension of state elimination to MDP presented in [22].

Our state space exploration performs a standard breadth-first search, but we use a decision diagram instead of the standard hash set to store the set of visited states. We do not store transitions, thus no continuous numeric values blow up the diagram. However, we count the number of predecessors of each state—thus we use an MTBDD. Since this number is a discrete quantity with low variation in most models, the diagrams usually remain compact. For the computation phase, we explore the state space again, this time creating a representation that includes transitions, but that is explicit. During this exploration, we keep track of the number of explored predecessors of each state. Whenever, for some fully-explored state s, this number reaches the predecessor count given by the MTBDD, we apply *state elimination*: we remove s from the (yet incomplete) explicit state space representation, and replace all of its incoming and outgoing transitions by direct transitions between the predecessors and successors of s. By redistributing the original transitions' probabilities and rewards in the right way, the quantity of interest remains unaffected. This method of computation simultaneously avoids the iterative algorithms' convergence and precision issues [20] and keeps memory usage due to the explicit representation low: on many models, most states are eliminated soon after they have been fully explored, thus only few need to be kept in memory at any time. Upon termination, only the initial state and goal state(s) remain, and the value for the quantity of interest can be read off the transitions connecting them.

Two technical insights make our computation phase work well: First, we use an explicit representation not only to avoid storing probabilities and rewards in an MTBDD, but also because state elimination tends to create unstructured intermediate state spaces that would blow up any BDD representation. Second, the precomputed predecessor count allows us to eliminate a state at precisely the moment after which we will not encounter it again in our search, avoiding costly re-explorations and re-eliminations.

Related work. State elimination stems from the classic algorithm to convert a finite automaton into a regular expression [10]. It was introduced to probabilistic model checking to solve parametric Markov chains [13] and forms the core of the Param [26] and Prophesy [14] tools. For non-parametric models, it enables efficient computation of reward-bounded reachability probabilities [22]. In this paper, we use it for non-parametric Markov chains and unbounded (infinite-horizon) properties. In all of these settings, its effectiveness crucially depends on the order in which states are eliminated, which is determined by (configurable) heuristics. Symblicit techniques have previously been used for long-run average properties [39], based on bisimulation minimisation, and later expanded in related settings [7]. A different form of elimination on strongly-connected components was used by Gui et al. [19] to accelerate (explicit-state) value iteration.

2 BACKGROUND

$\mathbb{R}_0^+$ and $\mathbb{R}^+$ are the sets of all non-negative and positive real numbers, respectively. We write $\{ x_1 \mapsto y_1, \dots \}$ to denote the function that maps all x_i to y_i, and if necessary in the respective context, implicitly maps to 0 all x for which no explicit mapping is specified. A (discrete) *probability distribution* over S is a function

$$\mu \colon S \to [0,1]$$

such that $\sum_{s \in spt(\mu)} \mu(s) = 1$ where the *support* $spt(\mu)$ is defined as

$$spt(\mu) \stackrel{\text{def}}{=} \{ s \in S \mid \mu(s) > 0 \}.$$

We write $Dist(S)$ for the set of all probability distributions over S.

2.1 Discrete-Time Markov Chains

Definition 2.1. A *discrete-time Markov chain* (DTMC) is a tuple

$$M = \langle S, s_I, P, R \rangle$$

of a finite set of *states* S, an *initial state* $s_I \in S$, a *transition* function $P \colon S \to Dist(S)$, and a *reward* function $R \colon S \to \mathbb{R}_0^+$.

We also write a transition as $s \xrightarrow{p} s'$ if $p = P(s)(s') > 0$. A transition is uniquely identified by the two states it connects. When in state s of a DTMC, we delay for one time unit before jumping to the next state. *Continuous-time Markov chains* (CTMC) extend DTMC by additionally assigning a *rate* $Q(s) \in \mathbb{R}^+$ to every state. Then the probability to delay for at most t time units is

$$1 - \mathrm{e}^{-Q(s) \cdot t},$$

i.e. the residence time follows the exponential distribution with rate $Q(s)$. In both models, the probability to then move to state s' is given by $P(s)$. When staying for t time units in state s, we incur a reward of $R(s) \cdot t$. To simplify the presentation, we use DTMC throughout this paper, but mention the changes needed in definitions or algorithms to use CTMC where appropriate.

Example 2.2. As a running example, we use a very abstract model of the Zeroconf protocol [6], shown as DTMC M_z in Fig. 1 (adapted from [18]). We draw transitions as arrows labelled with their probability. Non-zero rewards are given next to the states. M_z has 7 states and 12 transitions. The protocol is used by hosts joining a network to auto-configure a unique IP address. A new host joining the network of $h = 32$ hosts starts in state `1`. It selects an address

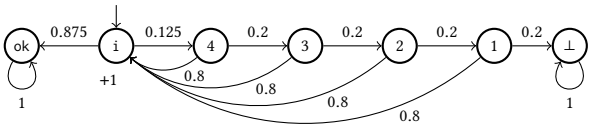

Figure 1: DTMC M_z for Zeroconf ($h = 32, a = 2^8, p = 0.2, n = 4$)

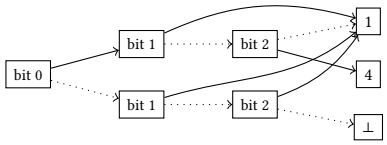

Figure 2: MTBDD for the predecessor count of M_z's states

uniformly at random from the space of $a = 2^8$ addresses. The probability that the address is already in use is $\frac{h}{a} = \frac{1}{8}$. The host checks $n = 4$ times whether its address is already in use. If it is not, all checks will succeed, modelled by state ok, to which we moved with probability $1 - \frac{h}{a}$. If it is, then states n down to 1 model the checks. Each check can fail to give the correct negative result due to message loss with probability $p = 0.2$. If all tests do so, then the host incorrectly believes that it has a unique address, in state $\bot$. Otherwise, it retries with a newly chosen address from state i. We incur a reward of 1 in state i, i.e. for every IP address we try. The size of the DTMC can be blown up arbitrarily via parameter n.

In practice, higher-level modelling languages like Modest [25] or the Prism language [33] are used to specify larger DTMC. The semantics of a DTMC is formally captured by its *paths*. A path represents a concrete resolution of both nondeterministic and probabilistic choices:

Definition 2.3. Given a DTMC M as above, a *finite path* is a sequence

$$\pi_{\text{fin}} = s_0\ t_0\ s_1\ t_1 \ldots s_n$$

of states $s_i \in S$ and delays $t_i \in \mathbb{R}^+$ where $P(s_i)(s_{i+1}) > 0$ and $t_i = 1$ for all $i \in \{0, \ldots, n-1\}$. Let $|\pi_{\text{fin}}| \stackrel{\text{def}}{=} n$, $\text{last}(\pi_{\text{fin}}) \stackrel{\text{def}}{=} s_n$,

$$\text{dur}(\pi_{\text{fin}}) \stackrel{\text{def}}{=} \sum_{i=0}^{n-1} t_i, \quad \text{and} \quad \text{rew}(\pi_{\text{fin}}) \stackrel{\text{def}}{=} \sum_{i=0}^{n-1} t_i \cdot R(s_i).$$

Π_{fin} is the set of all finite paths starting in s_I. A *path* is an analogous infinite sequence π, and Π are all paths starting in s_I. We define

$$s \in \pi \ \Leftrightarrow\ \exists\, i \colon s = s_i.$$

Let $\pi_{\to m}$ for $m \in \mathbb{N}$ be the prefix of π of length m, i.e. $|\pi_{\to m}| = m$, and let $\pi_{\to G}$ be the shortest prefix of π that contains a state in $G \subseteq S$, or $\bot$ if π contains no such state. Let $\text{rew}(\bot) \stackrel{\text{def}}{=} \infty$.

In CTMC, the t_i can be arbitrary numbers in $\mathbb{R}_0^+$. For M as above, following the rules described below Definition 2.1 and the standard cylinder set construction [4], we obtain a probability measure $\mathbb{P}_M$ on measurable sets of paths starting in s_I. We can then define the following properties of interest:

Definition 2.4. Given a set of goal states $G \subseteq S$, the *reachability probability* with respect to G is

$$\mathbb{P}(\diamond\, G) \stackrel{\text{def}}{=} \mathbb{P}_M(\pi \in \Pi \mid \exists\, g \in G \colon g \in \pi).$$

Let $r_G \colon \Pi \to \mathbb{R}_0^+$ be the random variable defined by

$$r_G(\pi) = \text{rew}(\pi_{\to G}).$$

Then the *expected reward* to reach G is the expected value of r_G under $\mathbb{P}_M$, written as $\mathbb{E}(\blacklozenge\, G)$. Let $r_{lra} \colon \Pi \to \mathbb{R}_0^+$ be defined by

$$r_{lra}(\pi) = \liminf_{i \to \infty} \frac{\text{rew}(\pi_{\to i})}{\text{dur}(\pi_{\to i})}.$$

Then the *long-run average reward* is the expectation of r_{lra} under $\mathbb{P}_M$, written as $\mathbb{L}$.

The steady-state probability $\mathbb{S}(S')$ of residing in a state in $S' \subseteq S$ is a special case of the long-run average reward where $R(s) = 1$ if $s \in S'$ and 0 otherwise. Whenever we consider a DTMC with a set of goal states G, we assume that they have been made absorbing, i.e. that for all $g \in G$ we have $P(g)(g) = 1$. Given a CTMC, reachability probabilities and expected rewards can be computed on its *embedded DTMC*, obtained by dividing all rewards by $Q(s)$; only for long-run averages do we need a dedicated treatment of the rates resp. residence times.

Example 2.5. For our Zeroconf example DTMC M_z from Fig. 1, we may want to compute the probability to eventually pick a unique address $\mathbb{P}(\diamond\, \{\text{ok}\})$, which will be just below 1, and the expected number of addresses that we ever try $\mathbb{E}(\blacklozenge\, \{\text{ok}, \bot\})$. Note that $\mathbb{E}(\blacklozenge\, \{\text{ok}\})$ is ∞ by definition since the set of paths that never reach state ok has positive probability.

2.2 Binary Decision Diagrams

Binary decision diagrams (BDDs) [9, 34] represent Boolean functions as rooted directed acyclic graphs. They have two leaf nodes, *true* and *false*. Every inner node is associated to one input bit, and has two children: the high (solid line) and low (dotted line) child. On a path from the root to a leaf, every bit must occur at most once. Such a path corresponds to the inputs in which bit b_i is assigned to *true* (*false*) if we go from a node for b_i to its high (low) child. We typically order the bits on all paths, merge isomorphic subgraphs, and remove redundant nodes. Then a BDD can represent many functions with few nodes.

In model checking, BDDs are used to represent sets of states (by assigning *true* to the binary encoding of a state[1] iff it is in the set) as well as the transition relation (by assigning *true* to the binary encoding of a pair of states if they are connected by a transition). In probabilistic model checking, however, we need to encode functions that map to rational numbers to encode transition probabilities, rewards, and the value vectors in value iteration. Most tools represent them as 64-bit floating-point values, but the corresponding binary representation does not typically allow good compression with BDDs. Symbolic probabilistic model checkers such as Prism thus use *multi-terminal* BDDs (MTBDDs) with one leaf node per number. Since a finite model only contains finitely many probabilities, or values for states, this approach is effective, but often not efficient: for example, when performing value iteration on our example DTMC M_z for $\mathbb{P}(\diamond\, \{\text{ok}\})$, we have to encode the following function after 5 iterations:

$$\{\ \text{i} \mapsto 0.99225, 4 \mapsto 0.9716, 3 \mapsto 0.966,\\ 2 \mapsto 0.938, 1 \mapsto 0.784, \text{ok} \mapsto 1, \bot \mapsto 0\ \}$$

[1] For models given in higher-level modelling languages, a state's binary encoding results from concatenating the binary values of all (finite-domain) variables in the model; thus the model's structure and symmetry can be exploited by the BDD encoding.

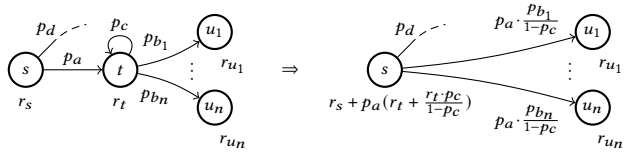

Figure 3: DTMC state elimination

Observe that every state has a distinct value, thus the MTBDD offers no compression. In practice, they only work well for very specific models with few distinct transition probabilities and rewards, and where the iterative numeric algorithms assign the same (intermediate) values to many states.

Example 2.6. Fig. 2 shows an MTBDD mapping every state of M_z to its number of predecessor states. We have 7 states, thus use 3 bits for their encoding. States 1 through 4 are encoded as that number, `i` is 5 (101_2), `ok` is 6 (110_2), and $\bot$ is 7 (111_2). There is no (reachable) state encoded as 0, thus we map 0 to the extra $\bot$ leaf node—in this way, such an MTBDD can indicate that certain states are unreachable, or have not been explored yet. Observe that the MTBDD representation achieves some compression by excluding two redundant nodes for bit 2. If we scale the model up by increasing n, the compression increases.

2.3 State Elimination

State elimination is a process by which a state of a DTMC is removed, i.e. transitions are modified such that it is no longer reachable from the initial state and it is removed from the state set S, in a way that preserves the values of all properties of interest. We show the schematic of state elimination in Fig. 3: we eliminate a state t by redistributing the probability to enter a self-loop onto its other outgoing transitions, then combine its incoming and outgoing transitions. It is easy to see that this preserves the probabilities of all measurable sets of paths that pass through t when projecting t out from every path. In particular, the paths that forever take the self-loop have probability mass zero, which is why we could eliminate the loop. For rewards, the transformation only preserves the *expected* reward values of sets of paths:

(1) In t, the expected number of times we take the self-loop is $p_c/(1-p_c)$, so the expected reward from passing through t is
$$\frac{r_t \cdot p_c}{1-p_c}$$
(for the loop) plus r_t (incurred when taking one of the other outgoing transitions).
(2) Out of s, the probability to enter t next is p_a, thus we multiply the expected reward of passing through t by p_a and add this to the reward of s.

Alg. 1 shows the pseudocode to perform state elimination on a DTMC stored in explicit data structures (i.e. hash sets for states, lists of transitions, etc.).

3 A SYMBLICIT APPROACH

As we explained in Sect. 1 and illustrated in Sect. 2.2, many probabilistic models do not give rise to a compact BDD-based representation if the numeric values—probabilities, rewards, rates for CTMC—are included. Furthermore, the standard iterative numeric algorithms like value iteration usually produce data that is hardly BDD-compressible. In this section, we present a combined symbolic-explicit approach that uses MTBDDs in a way that usually avoids such problems, and that uses state elimination to calculate $\mathbb{P}$, $\mathbb{E}$, and $\mathbb{L}$ values without having to keep (values for all states of) the entire state space in memory.

```
1  function Elim(⟨S, s_I, P, R⟩, s ∈ S, S_keep ⊆ S)        // all explicit
2      if s ∈ spt(P(s)) ∧ P(s)(s) < 1 then
3          foreach s′ ∈ spt(P(s)) \ { s } do                // redistribute the
4              P(s)(s′) := P(s)(s′)/(1 − P(s)(s))           // self-loop prob.
5          R(s) := R(s) + R(s)·P(s)(s)/(1 − P(s)(s))        // add reward
6          P(s)(s) := 0                                     // remove self-loop
7      foreach s_pre ∈ { s″ | s ∈ spt(P(s″)) } \ { s } do
8          p := P(s_pre)(s), P(s_pre)(s) := 0               // remove s_pre --p--> s
9          foreach s′ ∈ spt(P(s)) do                        // merge trans. of s into
10             P(s_pre)(s′) := P(s_pre)(s′) + p · P(s)(s′)   // tr. of s_pre
11         R(s_pre) := R(s_pre) + p · R(s)                  // add reward of s to s_pre
12     if s ∉ S_keep ∧ P(s)(s) = 0 then                     // remove s if not needed
13         P := P \ { s ↦ P(s) }, R := R \ { s ↦ R(s) }
14         S := S \ { s }
```

Alg. 1: State elimination for probabilities and exp. rewards

The pseudocode of our approach is shown as function `ExplElim` in Alg. 2. It uses functions `Explore` of Alg. 3 and `Elim` of Alg. 1. We typeset values that represent executable code in `monospace` font: compact specifications in high-level modelling languages are typically compiled to or interpreted as functions that, given an explicit (bit string) representation of a state, enumerate its transitions (`P`), compute its reward (`R`), and return *true* iff it is a goal state (`G`). We mark variables storing symbolic data (i.e. BDDs or MTBDDs) with a $\widehat{hat}$. All other values typeset in *italics* use explicit data structures such as bit strings for states, hash sets or queues of such bit strings, lists of transitions, etc.

Our first step, in line 2, is to symbolically explore the set of reachable states by calling function `Explore`. This function performs a standard breadth-first search, using a BDD for the set of visited states, and additionally constructs an MTBDD that counts the number of predecessors of each state like the one shown in Fig. 2 for M_z. In our implementation, *seen* and *pre* are actually managed in a single MTBDD as explained in Example 2.6.

We then, starting from line 3, perform another exploration of the state space. This time, however, we use explicit data structures, and we track the number of *fully explored* predecessors for every state in hash table *pre′*. A state is fully explored if its reward, all of its transitions, and all successor states, have been added to the explicit representations for S, R, and P. We track the set of fully explored states in hash set *done*. Whenever we are done visiting a state s in this second exploration (i.e. in line 13 and below), it has just become fully explored, and the fully-explored-predecessor count of its successors has changed. We then check which of these changed states fulfils the criteria for being eliminated: It must be

```
1  function ExplElim(s_I, P, R, G)  // explicit s_I, executable P, R, G
2      pre^ := Expl(s_I, P)          // get predecessor count MTBDD
3      done := ∅, agenda := { s_I }  // done: hash set, agenda: queue
4      S := { s_I }, P := ∅, R := ∅, pre' := { s_I ↦ 0 }   // all explicit
5      while agenda ≠ ∅ do
6          s := next element of agenda, agenda := agenda \ { s }
7          foreach s' ∈ spt(P(s)) do              // explore state s
8              P(s)(s') := P(s)(s'), R(s) := R(s)
9              if s' ∉ S then
10                 S := S ∪ { s }, agenda := agenda ∪ { s }
11                 pre' := pre' ∪ { s' ↦ 0 }
12             if s' ≠ s then pre'(s') := pre'(s') + 1
13         done := done ∪ { s }                   // s is now fully explored
14         E := { s_e | s_e ∈ {s} ∪ spt(P(s)) ∩ done }  // just modified,
15         E := { s_e | s_e ∈ E ∧ pre^(s_e) = pre'(s) }  // pred. explored:
16         foreach s_elim ∈ E do                  // eliminate these states
17             Elim(⟨S, s_I, P, R⟩, s_elim, { s_I })
18             if s_elim ∉ S then                 // cleanup
19                 pre' := pre' \ { s_elim ↦ pre'(s_elim) }
20                 done := done \ { s_elim }
21     if compute reachability prob. then return Σ_{g∈G} P(s_I)(g)
22     else if compute expected reward then
23         if spt(P(s_I)) \ G ≠ ∅ then return ∞   // case ℙ(◇G)<1
24         else return R(s_I) + Σ_{g∈G} P(s_I)(g) · R(g)
```

Alg. 2: Symblicit explore-eliminate algorithm

fully explored (which only s is for certain), and all of its predecessors must be fully explored (which we determine by comparing pre' and $\widehat{pre}$). We call `Elim` on these states in line 17. In this way, if we indeed manage to eliminate most states soon after they have been explored, the explicit data structures—S, P, R, pre', *done*, etc.—only track few states at any time and thus consume little memory. The predecessor count in $\widehat{pre}$ is crucial for being able to perform efficient elimination; without it, we would have to apply heuristics that could lead to states being eliminated that would later be explored as successors of other states again, leading to costly re-exploration and re-eliminations.

In `Elim`, if a state is part of the set S_{keep}, we still modify and "redirect" the transitions of its predecessors to go around this state, but we do not remove it from the state space. We use this to avoid eliminating the initial state s_I. We also do not eliminate states whose only transition is a self-loop: they do not have successors to which transitions could be redirected. Elimination will thus eventually reduce each bottom strongly connected component (BSCC) of the DTMC to one such self-loop state. Since we assume all goal states to only have a self-loop, each of them is a BSCC. Once the outer loop of line 5 in `ExplElim` finishes, `Elim` has been called for all states. Every surviving state at this point is thus the result of eliminating a number of transient states plus a non-goal BSCC or a goal state, and has become a direct successor of the initial state. We can then directly read the value of $\mathbb{P}(\diamond\,\mathsf{G})$ from the transitions to the goal

```
1  function Expl(s_I, P)              // explicit s_I, executable P
2      seen^ := { s_I }, agenda := { s_I }   // seen: BDD, agenda: queue
3      pre^ := { s_I ↦ 0 }            // predecessor count MTBDD
4      while agenda ≠ ∅ do
5          s := next element of agenda, agenda := agenda \ { s }
6          foreach s' ∈ spt(P(s)) \ { s } do
7              if s' ∉ seen^ then
8                  seen^ := seen^ ∪ { s' }, agenda := agenda ∪ { s' }
9                  pre^ := pre^ ∪ { s' ↦ 0 }
10             pre^(s') := pre^(s') + 1        // s is a new predecessor
11     return pre^
```

Alg. 3: Exploration with symbolic predecessor counting

states (line 21). Similarly, the value of $\mathbb{E}(\bullet\,\mathsf{G})$ can be derived directly from the remaining rewards, if it is not ∞ by definition (lines 23-24).

Example 3.1. For our example DTMC M_z of Fig. 1, we have already shown the predecessor count MTBDD computed by `Expl` in Fig. 2. Let us now step through the rest of `ExplElim` on this model. The partial state spaces that we consider in each step are shown in Fig. 4. Fully explored states are drawn with solid outlines, all other states (i.e. those in S but not in *done*) with dashed outlines. In step (1), we have just fully explored state `i`, i.e. we executed line 13 in the first iteration of the outer loop. Since $\widehat{pre}$ tells us that `i` still has unexplored predecessors, we cannot eliminate, and next explore `ok` in step (2). We then eliminate `ok`—its only predecessor `i` is fully explored—but since `ok` has just a single self-loop, the elimination has no effect. In step (3), we have just explored state 4, which can now be eliminated. The result is shown as step (4). We proceed in the same pattern in steps (5) through (8). Then, in step (9), we fully explore state 1. Now all predecessors of `i` are fully explored, and we can eliminate both 1 and `i`. For the sake of illustration, let us pick the more complicated ordering and eliminate `i` first. The result is shown as step (10). Since `i` is the initial state, we keep it, but redirect all incoming transitions. We also merge its rewards, which is why 1 now has a non-zero reward. Note that we show rationals in Fig. 4, but our implementation uses floating-point numbers. Remember that, without $\widehat{pre}$, we might have eliminated `i` too early; after any subsequent exploration of a state in $\{1, \ldots, n\}$, we would then have to re-eliminate `i`. We finally eliminate 1 in step (11) and explore state $\bot$ in step (12). At this point, the outer loop terminates; we read

$$\mathbb{P}(\diamond\,\{\,\mathsf{ok}\,\}) = \frac{4375}{4376} \approx 0.999771$$

$$\text{and} \quad \mathbb{E}(\bullet\,\{\,\mathsf{ok}, \bot\,\}) = 1 + \frac{119918}{119793} \approx 2.001043.$$

Observe that, at any time, we kept at most 4 explicit states in memory. We can arbitrarily increase the size of this model by increasing n, but will only ever need at most 4 explicit states in memory.

Long-run average rewards. The algorithm we presented so far computed reachability probabilities and expected rewards. For long-run average reward properties, there are no goal states. In such a case, our state elimination procedure computes the *recurrence*

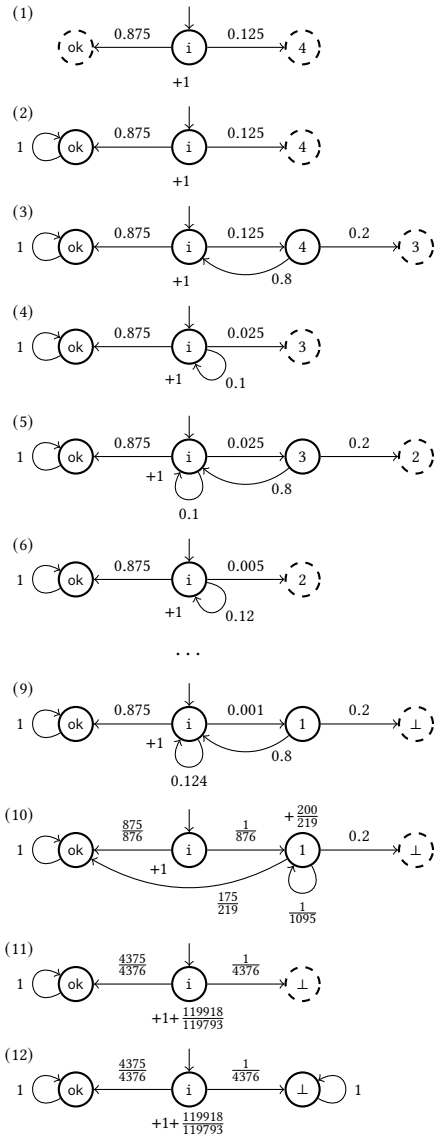

Figure 4: Exploration-elimination applied to M_z

reward for each BSCC [17]. To obtain the long-run average reward, we need to divide the recurrence reward for the rewards as given in the DTMC by the recurrence reward that we would obtain if all states had reward 1. We can do so by extending Algs. 1 and 2 to work on two reward structures r_u and r_l in parallel. Upon termination of the outer loop in `ExplElim`, we then have one of the two situations described at the end of Sect. 4 in [17], and can again directly read off the value for our ($\mathbb{L}$-)property. Consider Fig. 5: In the simpler case, the remaining model consists of the initial state $\bar{s}$ with a self-loop with probability one and $r_u = u_{\bar{s}}$, $r_l = l_{\bar{s}}$. In this case, the average value is

$$\frac{u_{\bar{s}}}{l_{\bar{s}}}.$$

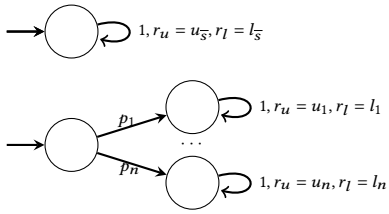

Figure 5: Computation of long-run averages

In the other case, the remaining model consists of the initial state $\bar{s}$ which has a probability of p_i to move to one of the other n remaining states s_i , $i = 1, \ldots, n$, which all have a self-loop with probability one and $r_u(s_i) = u_i$, $r_l(s_i) = l_i$. In this case, the average value is

$$\sum_{i=1,\ldots,n} p_i \frac{u_i}{l_i}.$$

When computing long-run average rewards, the final value for $\mathbb{L}$ may be small, but the two recurrence rewards that we need to divide are often extremely large numbers beyond what can usefully be represented as 64-bit (i.e. double-precision) floating point numbers. We thus implemented a variant of our algorithm that uses the GNU MPFR library (see mpfr.org) for arbitrary-precision floating-point arithmetic, allowing us to use more than 64 bits. We did not find this to significantly affect the performance of the overall approach.

Alternatives and optimisations. So far, we have assumed that we compute successors for each state explicitly and individually. For the state elimination phase, doing so is indeed necessary. However, for just exploring the states we could also compute the *transition relation* as a BDD, and then use the transition relation to symbolically explore the set of reachable states. This is the standard approach in model checkers such as Prism and potentially faster than the semi-symbolic approach we have discussed. Also, using the transition relation and according MTBDD operations (in particular sum abstraction), the number of predecessors of each state can be computed symbolically as well.

We also assumed that the reachable states and the number of predecessors are stored as (MT)BDDs. An alternative is to store these numbers on secondary storage (e.g. on a hard disk) similar to [29]. Such an approach would be useful for models with state spaces unsuitable to be stored as BDDs. This might be the case because of lack of implicit symmetries or because the size of the representation of each state is not constant.

4 EXPERIMENTAL EVALUATION

We have implemented a preliminary version of our method as a plugin for the probabilistic model checker ePMC [27]. For the analysis, we transform the model and property into C++ code to quickly compute successors of states, similar to the approach used in SPIN [32]. This C++ file is then extended with code to achieve the following: In the first phase, we explore the state space breadth-first, exploring each state explicitly but storing sets of states as BDDs, using the BDD package CUDD [2]. In the second phase, we generate an MTBDD mapping all states to value 0. Then, we iterate over all reachable states, recompute their successors, and increment the value of these successors in the MTBDD by 1 each

Table 1: Simple Molecular Reactions performance results

params	$\|S\|$	result	time	states	trans	mem
10^1	11	2.2623e+01	8	5	5	22
10^2	101	2.3894e+01	8	5	5	22
10^3	1000	2.4012e+01	7	5	5	22
10^4	10.0 k	2.4024e+01	7	5	5	25
10^5	100 k	2.4025e+01	10	5	5	28
10^6	1.00 M	2.4025e+01	32	5	5	27
10^7	10.0 M	2.4025e+01	258	5	5	31
10^8	100 M	2.4025e+01	2609	5	5	29
10^9	1000 M	2.4025e+01	18807	5	5	25

time. In the third phase, we execute the state elimination algorithm as discussed. This C++ code is then compiled and run in a process separate from ePMC, of which we measure the memory usage. (The memory usage of ePMC itself is not of much interest, because it is rather small and about the same for any analysis.)

In the following, we apply our tool to several case studies from the Prism website. All experiments were performed on a MacBook Pro with a 2.7 GHz Quad-Core Intel Core i7 processor and 16 GB 2133 MHz LPDDR3 RAM. In the tables, $|S|$ is the total number of states the model has for the given parameters, "result" is the value of the property computed, "time" is the total runtime of the analysis in seconds, and "states" ("trans") is the maximum number of states (transitions) stored explicitly at any time. By "mem" we denote the peak memory usage of the analysis process in megabytes.

Simple Molecular Reactions[2] is a CTMC model of the chemical reaction $\mathrm{Na}+\mathrm{Cl} \leftrightarrow \mathrm{Na}^+ + \mathrm{Cl}^-$. The parameters of this case study are *N1* and *N2*, the initial numbers of *Na* and *Cl* molecules, respectively. In Table 1, we show the performance figures for the analysis of `R=? [ S ]`, i.e. the expected long-run average number of *Na* molecules. We consider a starting configuration in which the number of *Na* and *Cl* is the same, that is, *N1* = *N2*.

We see that our method scales well for large numbers of molecules and accordingly large state spaces. The memory usage grows only slowly with increasing model parameters, and the number of states and transitions that need to be stored explicitly is constant. The runtime does increase significantly, though, since the symbolic exploration phase as well as the state elimination need to process exponentially more states as the parameter values increase.

The *Bounded Retransmission Protocol*[3] [31] transmits files divided into *N* packets. Data and acknowledgement packets are sent over unreliable channels; each can be retransmitted at most *MAX* times. Table 2, provides performance figures for the analysis of `P=?[ F s=5 ]`, i.e. the probability that the sender does not eventually report a successful transmission, for different values of parameters *N* and *MAX*. Compared to the instances on the Prism website, we user higher values for *N* and *MAX* because our focus is on the scalability of our method. Our first table entry uses the same values as the largest instance considered on the Prism website.

Our method handles instances with several million states with low memory usage. Even for higher parameter values for which

[2]https://www.prismmodelchecker.org/casestudies/molecules.php
[3]https://www.prismmodelchecker.org/casestudies/brp.php

Table 2: Bounded Retransmission Protocol performance

params	$\|S\|$	result	time	states	trans	mem
2^6-5	4936	4.48e-08	9	12	29	25
2^6-10	9101	1.05e-15	9	20	53	25
2^6-100	84.1 k	5.03e-153	9	140	517	25
2^6-1000	834 k	3.14e-1526	25	258	986	33
2^7-10	18.2 k	2.11e-15	9	20	53	25
2^7-100	168 k	1.01e-152	10	140	517	32
2^7-1000	1.67 M	6.28e-1526	38	514	1978	36
2^8-10	36.4 k	4.21e-15	9	20	53	27
2^8-100	336 k	2.01e-152	15	150	517	28
2^8-1000	3.33 M	1.26e-1525	72	1026	3962	39
2^9-10	72.7 k	8.42e-15	9	20	53	29
2^9-100	672 k	4.03e-152	18	140	517	32
2^9-1000	6.66 M	2.51e-1525	140	1340	5167	42
2^{10}-10	145 k	1.69e-14	10	20	53	29
2^{10}-100	1.26 M	8.06e-152	29	140	517	30
2^{10}-1000	13.3 M	5.02e-1525	280	1340	5167	43
2^{11}-10	291 k	3.37e-14	12	20	53	27
2^{11}-100	2.69 M	1.61e-151	50	140	517	30
2^{11}-1000	26.6 M	1.00e-1524	552	1340	5167	41
2^{12}-10	582 k	6.74e-14	17	20	53	27
2^{12}-100	5.37 M	3.22e-151	95	140	517	28
2^{12}-1000	53.3 M	2.01e-1524	1151	1340	5004	39
2^{13}-10	1.16 M	1.35e-13	26	20	53	28
2^{13}-100	10.7 M	6.45e-151	187	140	517	27
2^{13}-1000	107 M	4.02e-1524	2385	1340	5004	40
2^{14}-10	2.33 M	2.67e-13	48	20	53	27
2^{14}-100	21.5 M	1.29e-150	392	140	517	28
2^{14}-1000	213 M	8.03e-1524	4534	1340	5004	40
2^{15}-10	4.65 M	5.39e-13	91	20	53	26
2^{15}-100	43.0 M	2.58e-150	781	140	517	27
2^{15}-1000	426 M	1.61e-1523	9039	1340	5004	41
2^{16}-10	9.31 M	1.08e-12	20	54	156	25
2^{16}-100	86.0 M	5.16e-150	140	503	1362	29
2^{17}-10	18.6 M	2.1572e-12	312	20	54	26
2^{17}-100	172 M	1.03e-149	2847	140	503	29

the number of total states $|S|$ is in the millions, we never use more than a few thousand explicit states and transitions and less than 100 MB of memory.

The *Wireless Communication Cell*[4] case study is a performance model of wireless communication cells [28]. Parameter *N* describes the number of channels in a cell. We compute `R{"calls"}=? [ S ]`, i.e. the average number of calls in the cell on the long run. We provide performance figures in Table 3. Also for this case study, our approach works fine in that the number of states and transitions to be stored is constant and peak memory usage remains very low.

The *Crowds Protocol*[5] [38] allows anonymous web browsing. To do so, messages are not directly sent, but forwarded to other users, who might either forward them again or send them to the

[4]https://www.prismmodelchecker.org/casestudies/cell.php
[5]https://www.prismmodelchecker.org/casestudies/crowds.php

Table 3: Wireless Communication Cell performance results

params	$\|S\|$	result	time	states	trans	mem
10000	10.0 k	7.00e+01	7	5	5	24
100000	100 k	7.00e+01	9	5	5	24
1000000	1.00 M	7.00e+01	24	5	5	25
10000000	10.0 M	7.00e+01	183	5	5	26
100000000	100 M	7.00e+01	1731	5	5	25

Table 4: Crowds Protocol performance results

params	$\|S\|$	result	time	states	trans	mem
5-5	8653	2.7884e-01	9	289	1278	77
5-6	18.8 k	2.9791e-01	9	510	2236	77
5-7	37.3 k	3.1812e-01	9	823	3898	164
10-5	111 k	2.1662e-01	19	6604	33513	2487
10-6	353 k	2.3162e-01	106	17824	89120	10698
15-5	592 k	1.9674e-01	676	44104	356143	9240

destination. By doing so, it is hard for attackers to decide whether the sender of a message is the original sender or is just forwarding the message. The model we consider has two parameters: *TotalRuns* is the number of routing paths of the model instance, and *CrowdSize* is the number of honest participants of the protocol. We consider the property `Pmax=? [ F (new & runCount=0 & observe0 > observe1 & ... & observe0 > observe19) ]`, i.e. the probability that an attacker can eventually observe the true sender of a message more often than participants just forwarding the message and is thus able to guess the original sender. In Table 4, we provide performance figures. For this model the current implementation of our method does not perform well. The reason is that too many states and transitions have to be stored explicitly at the same time, leading to a large memory overhead.

The *Embedded Control System*[6] [35] features a cyclic polling process. If a certain component detects that more than a given number *MAX_COUNT* of cycles have been skipped due to issues, the system is shut down for safety reasons. We consider the property `R{"danger"}=? [ F "down" ]`, i.e. the expected time the system is in an endangered state before it eventually has to be shut down. We provide performance figures in Table 5. Here, our method works fine again as the number of states and transitions stored explicitly is limited.

5 CONCLUSION AND FUTURE WORK

In this paper, we have presented a new memory-efficient analysis method for properties of stochastic models. For conciseness of presentation, we have focused on DTMCs (with some of the case studies being CTMCs), but the method should in principle also work for more expressive formalisms like MDP (using the MDP state elimination idea of [22]). Our experiments show that it can analyse models with millions of states in just a few megabytes of memory. Its efficiency depends on how amenable a model is to BDD compression (like any symbolic method), and also on how well-suited the graph structure of the state space is for state elimination. The Crowds Protocol case study shows that some models do not allow states to be eliminated soon enough when exploring in a breadth-first order for the memory savings to materialise. As we have seen, runtime also still increases with state space size, although part of that effect is due to the simple symbolic exploration engine of our prototype tool. All in all, our method thus complements existing techniques such as Prism's *mtbdd* engine, Storm's bisimulation minimisation, and the learning-based approaches [1, 8]. Where it works, however, our method can offer unprecedented scalability in probabilistic model checking.

An incidental advantage of our method is that it can directly obtain an *exact* result: Our current implementation uses (variable-precision) floating-point numbers, with computation precision limited by the properties of this representation. Yet we could as well use rational arithmetic to obtain exact values without any change in the core algorithms, since all intermediate and final values are rational—though at an increase in computation time and memory usage. Alternatively, we could use interval arithmetic to obtain precise upper and lower bounds, again without any change in the algorithms, and with only moderate overhead. This is in contrast to methods based on value iteration or state space abstraction, for which special precaution is required to obtain sound results.

As future work, we plan to turn our prototype implementation into a full-fledged tool, compare its performance to the various approaches implemented in other tools, add support for MDP, and provide exact or interval arithmetic-based computations. Motivated by the problematic performance of the Crowds protocol case study, we also want to consider different search orders so as to improve the behaviour for models for which the currently implemented strict breadth-first search order does not perform well.

Table 5: Embedded Control System performance results

params	$\|S\|$	result	time	states	trans	mem
512	320 k	3.3454e-01	56	267	11938	80
1024	639 k	3.3731e-01	107	267	11938	80
2048	1.28 M	3.4283e-01	201	267	11938	80
4096	2,56 M	3.5371e-01	400	267	11938	80
8192	5,11 M	3.7485e-01	794	267	11938	80
16384	10.2 M	4.1469e-01	1579	267	11938	80
32768	20.4 M	7.6563e-01	2914	284	10513	72

DATA AVAILABILITY

The tools used and data generated in our experimental evaluation as well as instructions to replicate the experiments are archived and available at DOI 10.4121/13379135 [23].

ACKNOWLEDGMENTS

This work was supported by the EU under project 864075 CAESAR, and by NWO VENI grant no. 639.021.754.

[6] https://www.prismmodelchecker.org/casestudies/embedded.php

REFERENCES

[1] Pranav Ashok, Yuliya Butkova, Holger Hermanns, and Jan Kretínský. 2018. Continuous-Time Markov Decisions Based on Partial Exploration. In *ATVA (Lecture Notes in Computer Science)*, Vol. 11138. Springer, 317–334. https://doi.org/10.1007/978-3-030-01090-4_19

[2] R. Iris Bahar, Erica A. Frohm, Charles M. Gaona, Gary D. Hachtel, Enrico Macii, Abelardo Pardo, and Fabio Somenzi. 1997. Algebraic Decision Diagrams and Their Applications. *Formal Methods in System Design* 10, 2/3 (1997), 171–206. https://doi.org/10.1023/A:1008699807402

[3] Christel Baier, Holger Hermanns, and Joost-Pieter Katoen. 2019. The 10,000 Facets of MDP Model Checking. In *Computing and Software Science - State of the Art and Perspectives*. Lecture Notes in Computer Science, Vol. 10000. Springer, 420–451. https://doi.org/10.1007/978-3-319-91908-9_21

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

[5] Christel Baier, Joachim Klein, Linda Leuschner, David Parker, and Sascha Wunderlich. 2017. Ensuring the Reliability of Your Model Checker: Interval Iteration for Markov Decision Processes. In *CAV (Lecture Notes in Computer Science)*, Vol. 10426. Springer, 160–180. https://doi.org/10.1007/978-3-319-63387-9_8

[6] Henrik C. Bohnenkamp, Peter van der Stok, Holger Hermanns, and Frits W. Vaandrager. 2003. Cost-Optimization of the IPv4 Zeroconf Protocol. In *DSN*. IEEE Computer Society, 531–540. https://doi.org/10.1109/DSN.2003.1209963

[7] Aaron Bohy, Véronique Bruyère, Jean-François Raskin, and Nathalie Bertrand. 2017. Symblicit algorithms for mean-payoff and shortest path in monotonic Markov decision processes. *Acta Inf.* 54, 6 (2017), 545–587. https://doi.org/10.1007/s00236-016-0255-4

[8] Tomás Brázdil, Krishnendu Chatterjee, Martin Chmelik, Vojtech Forejt, Jan Kretínský, Marta Z. Kwiatkowska, David Parker, and Mateusz Ujma. 2014. Verification of Markov Decision Processes Using Learning Algorithms. In *ATVA (Lecture Notes in Computer Science)*, Vol. 8837. Springer, 98–114. https://doi.org/10.1007/978-3-319-11936-6_8

[9] Randal E. Bryant. 2018. Binary Decision Diagrams. In *Handbook of Model Checking*. Springer, 191–217. https://doi.org/10.1007/978-3-319-10575-8_7

[10] Janusz A. Brzozowski and Edward J. McCluskey. 1963. Signal Flow Graph Techniques for Sequential Circuit State Diagrams. *IEEE Trans. Electronic Computers* 12, 2 (1963), 67–76. https://doi.org/10.1109/PGEC.1963.263416

[11] Yuliya Butkova, Arnd Hartmanns, and Holger Hermanns. 2019. A Modest Approach to Modelling and Checking Markov Automata. In *QEST (Lecture Notes in Computer Science)*, Vol. 11785. Springer, 52–69. https://doi.org/10.1007/978-3-030-30281-8_4

[12] Sagar Chaki and Arie Gurfinkel. 2018. BDD-Based Symbolic Model Checking. In *Handbook of Model Checking*. Springer, 219–245. https://doi.org/10.1007/978-3-319-10575-8_8

[13] Conrado Daws. 2004. Symbolic and Parametric Model Checking of Discrete-Time Markov Chains. In *ICTAC (Lecture Notes in Computer Science)*, Vol. 3407. Springer, 280–294. https://doi.org/10.1007/978-3-540-31862-0_21

[14] Christian Dehnert, Sebastian Junges, Nils Jansen, Florian Corzilius, Matthias Volk, Harold Bruintjes, Joost-Pieter Katoen, and Erika Ábrahám. 2015. PROPhESY: A PRObabilistic ParamEter SYnthesis Tool. In *CAV (Lecture Notes in Computer Science)*, Vol. 9206. Springer, 214–231. https://doi.org/10.1007/978-3-319-21690-4_13

[15] Christian Dehnert, Sebastian Junges, Joost-Pieter Katoen, and Matthias Volk. 2017. A Storm is Coming: A Modern Probabilistic Model Checker. In *CAV (Lecture Notes in Computer Science)*, Vol. 10427. Springer, 592–600. https://doi.org/10.1007/978-3-319-63390-9_31

[16] Christian Eisentraut, Holger Hermanns, and Lijun Zhang. 2010. On Probabilistic Automata in Continuous Time. In *LICS*. IEEE Computer Society, 342–351. https://doi.org/10.1109/LICS.2010.41

[17] Paul Gainer, Ernst Moritz Hahn, and Sven Schewe. 2018. Accelerated Model Checking of Parametric Markov Chains. In *ATVA (Lecture Notes in Computer Science)*, Vol. 11138. Springer, 300–316. https://doi.org/10.1007/978-3-030-01090-4_18

[18] Paul Gainer, Ernst Moritz Hahn, and Sven Schewe. 2018. Incremental Verification of Parametric and Reconfigurable Markov Chains. In *QEST (Lecture Notes in Computer Science)*, Vol. 11024. Springer, 140–156. https://doi.org/10.1007/978-3-319-99154-2_9

[19] Lin Gui, Jun Sun, Songzheng Song, Yang Liu, and Jin Song Dong. 2014. SCC-Based Improved Reachability Analysis for Markov Decision Processes. In *ICFEM (Lecture Notes in Computer Science)*, Vol. 8829. Springer, 171–186. https://doi.org/10.1007/978-3-319-11737-9_12

[20] Serge Haddad and Benjamin Monmege. 2014. Reachability in MDPs: Refining Convergence of Value Iteration. In *RP (Lecture Notes in Computer Science)*, Vol. 8762. Springer, 125–137. https://doi.org/10.1007/978-3-319-11439-2_10

[21] Serge Haddad and Benjamin Monmege. 2018. Interval iteration algorithm for MDPs and IMDPs. *Theor. Comput. Sci.* 735 (2018), 111–131. https://doi.org/10.1016/j.tcs.2016.12.003

[22] Ernst Moritz Hahn and Arnd Hartmanns. 2016. A Comparison of Time- and Reward-Bounded Probabilistic Model Checking Techniques. In *SETTA (Lecture Notes in Computer Science)*, Vol. 9984. 85–100. https://doi.org/10.1007/978-3-319-47677-3_6

[23] Ernst Moritz Hahn and Arnd Hartmanns. 2021. Symblicit Exploration and Elimination for Probabilistic Model Checking (Artifact). 4TU.ResearchData. https://doi.org/10.4121/13379135

[24] Ernst Moritz Hahn, Arnd Hartmanns, Christian Hensel, Michaela Klauck, Joachim Klein, Jan Kretínský, David Parker, Tim Quatmann, Enno Ruijters, and Marcel Steinmetz. 2019. The 2019 Comparison of Tools for the Analysis of Quantitative Formal Models (QComp 2019 Competition Report). In *25 Years of TACAS: TOOLympics (Lecture Notes in Computer Science)*, Vol. 11429. Springer, 69–92. https://doi.org/10.1007/978-3-030-17502-3_5

[25] Ernst Moritz Hahn, Arnd Hartmanns, Holger Hermanns, and Joost-Pieter Katoen. 2013. A compositional modelling and analysis framework for stochastic hybrid systems. *Formal Methods in System Design* 43, 2 (2013), 191–232. https://doi.org/10.1007/s10703-012-0167-z

[26] Ernst Moritz Hahn, Holger Hermanns, Björn Wachter, and Lijun Zhang. 2010. PARAM: A Model Checker for Parametric Markov Models. In *CAV (Lecture Notes in Computer Science)*, Vol. 6174. Springer, 660–664. https://doi.org/10.1007/978-3-642-14295-6_56

[27] Ernst Moritz Hahn, Yi Li, Sven Schewe, Andrea Turrini, and Lijun Zhang. 2014. iscasMc: A Web-Based Probabilistic Model Checker. In *FM*. 312–317. https://doi.org/10.1007/978-3-319-06410-9_22

[28] Guenter Harine, Raymond A. Marie, Ramón Puigjaner, and Kishor S. Trivedi. 2001. Loss formulas and their application to optimization for cellular networks. *IEEE Trans. Veh. Technol.* 50, 3 (2001), 664–673. https://doi.org/10.1109/25.933303

[29] Arnd Hartmanns and Holger Hermanns. 2015. Explicit Model Checking of Very Large MDP Using Partitioning and Secondary Storage. In *ATVA (Lecture Notes in Computer Science)*. 131–147. https://doi.org/10.1007/978-3-319-24953-7_10

[30] Arnd Hartmanns and Benjamin Lucien Kaminski. 2020. Optimistic Value Iteration. In *CAV (Lecture Notes in Computer Science)*, Vol. 12225. Springer, 488–511. https://doi.org/10.1007/978-3-030-53291-8_26

[31] Leen Helmink, M. P. A. Sellink, and Frits W. Vaandrager. 1993. Proof-Checking a Data Link Protocol. In *TYPES (Lecture Notes in Computer Science)*, Vol. 806. Springer, 127–165. https://doi.org/10.1007/3-540-58085-9_75

[32] Gerard J. Holzmann. 1997. The Model Checker SPIN. *IEEE Trans. Software Eng.* 23, 5 (1997), 279–295. https://doi.org/10.1109/32.588521

[33] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. 2011. PRISM 4.0: Verification of Probabilistic Real-Time Systems. In *CAV (Lecture Notes in Computer Science)*, Vol. 6806. Springer, 585–591. https://doi.org/10.1007/978-3-642-22110-1_47

[34] C. Y. Lee. 1959. Representation of switching circuits by binary-decision programs. *The Bell System Technical Journal* 38, 4 (1959), 985–999. https://doi.org/10.1002/j.1538-7305.1959.tb01585.x

[35] J. Muppala, G. Ciardo, and K. Trivedi. 1994. Stochastic reward nets for reliability prediction. *Communications in Reliability, Maintainability and Serviceability* 1, 2 (1994), 9–20.

[36] M. L. Puterman. 1994. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons Inc., New York.

[37] Tim Quatmann and Joost-Pieter Katoen. 2018. Sound Value Iteration. In *CAV (Lecture Notes in Computer Science)*, Vol. 10981. Springer, 643–661. https://doi.org/10.1007/978-3-319-96145-3_37

[38] Michael K. Reiter and Aviel D. Rubin. 1998. Crowds: Anonymity for Web Transactions. *ACM Trans. Inf. Syst. Secur.* 1, 1 (1998), 66–92. https://doi.org/10.1145/290163.290168

[39] Ralf Wimmer, Bettina Braitling, Bernd Becker, Ernst Moritz Hahn, Pepijn Crouzen, Holger Hermanns, Abhishek Dhama, and Oliver E. Theel. 2010. Symblicit Calculation of Long-Run Averages for Concurrent Probabilistic Systems. In *QEST*. IEEE Computer Society, 27–36. https://doi.org/10.1109/QEST.2010.12