Tensor Probabilistic Model Checking of Finite-Horizon Markov Chains

Jianlin Li, Nick Guo, Peter Ye, and Yizhou Zhang(✉)

David R. Cheriton School of Computer Science
University of Waterloo, Canada
{jianlin.li, nick.guo, p2ye, yizhou}@uwaterloo.ca

Abstract. We reexamine the problem of verifying Markov chains with respect to step-bounded reachability probabilities. Prevailing approaches rely on encoding the state-transition matrix using either explicit or symbolic representations. While these approaches are effective for sparse transition dynamics, they scale less favorably in the dense regime.

Our insight is to cast probabilistic model checking of Markov chains as computations over dense tensors. This methodology enables the use of off-the-shelf compiler toolchains for optimized execution of these tensor computations on hardware accelerators. We prove the soundness of the methodology of mapping probabilistic model checking to tensor computations. We implement our approach in a tool called Tessa. Empirical evaluation shows that Tessa unlocks massive speedups over state-of-the-art methods on selected benchmarks from the literature.

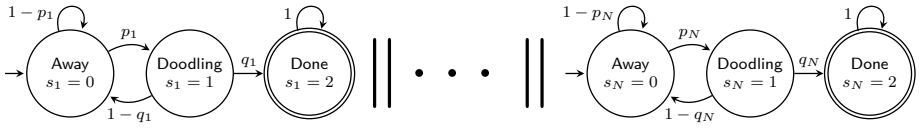
1 Introduction

Probabilistic model checking is a formal verification technique for verifying quantitative properties of systems exhibiting stochastic behavior. Given a mathematical model (such as a Markov chain) representing the behavior of a system over time, an algorithmic procedure determines whether the system satisfies properties of interest (such as step-bounded reachability).

As an example, consider the problem of scheduling an urgent faculty meeting (Figure 1) involving N professors by collecting their availability via a Doodle poll. At any time step, each professor i is in one of three states: Away, Doodling, or Done. A professor Away from their inbox may, with probability p_i, notice the Doodle poll email and transition to Doodling to answer the Doodle poll. However, the Doodling state is fragile: with probability q_i, they successfully complete the poll and reach Done; but with probability $1 - q_i$, they are interrupted by a new email or a knock on their door, forcing them back to be Away without hitting submit. The dynamics capture the intermittent nature of professors' attention. Each Markov chain in Figure 1(a) depicts the behavior of a professor.

The quantitative property to verify is the probability that all N professors have reached Done within a horizon of H time steps.

The program in Figure 1(b) models this system of stochastic processes, similarly to how one would structure it in the PRISM [1] and JANI [10] modeling



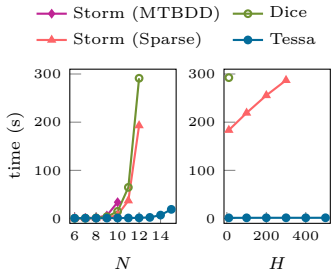
(a) N parallel Markov chains capturing the stochastic behavior of N professors.

module P_1
  var s_1 : [0 .. 2] **init** 0;
  [a] $(s_1 = 0) \rightarrow p_1 : (s_1 := 1) + (1 - p_1) : (s_1 := 0)$;
  [a] $(s_1 = 1) \rightarrow q_1 : (s_1 := 2) + (1 - q_1) : (s_1 := 0)$;
  [a] $(s_1 = 2) \rightarrow 1.0 : (s_1 := 2)$;

module $\cdots$ // *modules* $P_2 \cdots P_N$

goal $s_1 = 2 \wedge \cdots \wedge s_N = 2$

(b) A model of N professors (in PRISM-like syntax).



(c) Scaling plots.

Figure 1. Running example: collecting availability from N professors. The target property is the probability that all N professors reach Done within horizon H.

languages. Each module P_i models the behavior of professor i: it declares a state variable s_i with domain $\{0, 1, 2\}$ (indicating Away, Doodling, and Done, respectively), and three *commands* that update s_i according to the transition probabilities p_i and q_i. Each command is *guarded* by a Boolean expression (e.g., $s_i = 0$ for the first command); a command is enabled only if its guard is satisfied in the current state. The **goal** clause of the global model specifies the target property that all professors are in state Done.

Despite the semantic simplicity of this model, the verification problem is inherently computationally expensive due to state explosion. Since each process has 3 states, the global state-space size is 3^N. It is unlikely that any model checking tool on a classical computer can avoid this exponential blowup. Nevertheless, we would like to push the boundary of tractable instances as far as possible, especially given the raw speed of modern hardware accelerators at our disposal.

State-of-the-art methods largely fall into two categories:

– Probabilistic model checkers, such as PRISM [29] and Storm [16], are the prevailing tools for such verification tasks. They represent the state-transition dynamics using either explicit (i.e., sparse matrices of size $3^N \times 3^N$ in the N-professor model) or symbolic (i.e., multi-terminal binary decision diagrams, or MTBDDs) representations. Both representations are highly optimized for CPU execution in the situation of sparse transition dynamics. However, they are not as effective in the dense regime, and their irregular memory access patterns make it challenging to map them efficiently to modern hardware accelerators.
– A recent development is the use of probabilistic inference for probabilistic model checking. Rubicon [19] compiles a DTMC into the Dice [18] probabilistic programming language; running the Dice program using Dice's inference engine—

weighted model counting on binary decision diagrams (BDDs)—computes the step-bounded reachability probability. Still, BDD operations are pointer-rich, so they involve indirection that does not map well to devices like GPUs.

We present Tessa, a new methodology for verifying step-bounded reachability properties of DTMCs, by compiling DTMC models to dense tensor computations (i.e., array programs in an array programming language like JAX [9]).

As Figure 1(c) shows, Tessa exhibits markedly better scalability than existing tools on the N-professor model. The two plots show running times for different values of N (fixing the horizon $H = 10$) and different values of H (fixing $N = 12$), respectively. The first plot shows that, while Tessa does not change the asymptotic complexity of the problem (the state-space size still grows exponentially with N), it aggressively pushes down the constant factor of the exponential growth compared to Storm and Dice. The second plot shows that while all tools appear to scale linearly with H, Tessa has a significantly smaller slope—the curve is nearly flat.[1]

A key reason for Tessa's scalability is its representation of state distributions as dense tensors, and state-transition dynamics as transformations on these tensors. The name Tessa evokes a tesseract—a high-dimensional cube—reflecting our insight: rather than eagerly materializing the state-transition dynamics as a sparse matrix or a decision diagram, we treat the joint distribution of N state variables as an order-N tensor, and the transition dynamics as a tensor program.

A key advantage of this tensor representation is its amenability to optimization (by modern tensor compilers) and acceleration (by hardware such as GPUs). Because Tessa maps the verification problem to tensor operations, we can use an off-the-shelf tensor compiler (XLA [2], in this case) to optimize the generated tensor code[2] and offload them to GPU devices for massively parallel execution. In contrast, established methodologies target representations that induce irregular memory accesses, which limit both the degree of parallelism and the ease of parallelization achievable.

Targeting tensors also naturally supports parameter search for DTMCs with unknown transition probabilities. As Tessa generates differentiable tensor programs, we can leverage JAX's automatic differentiation to compute gradients of a rich selection of optimization objectives. Furthermore, the computationally intensive gradient calculation can be optimized by XLA and accelerated by GPUs.

We believe the methodology embodied by Tessa usefully complements the toolbox of probabilistic model checkers. It is not a silver bullet; existing methods are already effective for sparse models. However, Tessa opens up a new regime of tractable models that were previously out of reach.

Contributions. Our contributions are both theoretical and practical:

– We base our development on a core language for specifying DTMCs (Section 3). We show how to interpret programs in this core language as computations over

[1] The comparison may make state-of-the-art tools look inefficient, but they are actually highly optimized for CPUs. The point is that a methodological shift unlocks massive speedups.

[2] Tessa running times in Figure 1(c) include compilation times of JAX and XLA.

tensors (Section 4). This interpretation captures the essence of the implementation of Tessa. We prove that this tensor interpretation is sound with respect to the verification of step-bounded reachability properties (Section 5).

– We implement Tessa and evaluate it on selected benchmarks from the literature. Tessa employs a domain-specific compiler stack that automatically optimizes and parallelizes dense tensor computations for GPU execution (Section 6). Experimental results highlight substantial performance gains unlocked by Tessa's tensor-based approach (Section 7).

2 Preliminaries

We review discrete-time Markov chains (DTMCs) and the verification problem of step-bounded reachability properties. A *Markov chain* is a tuple $\mathcal{M} = (\mathcal{S}, \iota, \eta, \mathcal{G})$:

– $\mathcal{S}$ is a finite set of states.
– $\iota \in \mathsf{D}(\mathcal{S})$ is the initial-state distribution.
– $\eta : \mathcal{S} \to \mathsf{D}(\mathcal{S})$ is the transition function.
– $\mathcal{G} \subseteq \mathcal{S}$ is the set of goal states.

We use $\mathsf{D}(\mathcal{S})$ to denote the set of probability distributions over $\mathcal{S}$. We write $\iota(s)$ to denote the probability of starting in state s. We write $\eta(s, s')$ to denote the probability of transitioning to state s' from state s. Thus, we have that $\sum_{s \in \mathcal{S}} \iota(s) = 1$ and also that for any state $s \in \mathcal{S}$, $\sum_{s' \in \mathcal{S}} \eta(s, s') = 1$.

In this paper, we are interested in the verification of step-bounded reachability properties of the form $\Pr_{\mathcal{M}}\big(\Diamond^{\leq n}\mathcal{G}\big)$, which denotes the probability of reaching a goal state within n steps.

The probability of reaching a goal state within n steps starting from state s is denoted $\Pr_{\mathcal{M}}\big(s \vDash \Diamond^{\leq n}\mathcal{G}\big)$ and is defined inductively as follows:

$$\Pr_{\mathcal{M}}\big(s \vDash \Diamond^{\leq 0}\mathcal{G}\big) \coloneqq \begin{cases} 1 & \text{if } s \in \mathcal{G} \\ 0 & \text{otherwise} \end{cases} \tag{2.1}$$

$$\Pr_{\mathcal{M}}\big(s \vDash \Diamond^{\leq n+1}\mathcal{G}\big) \coloneqq \begin{cases} 1 & \text{if } s \in \mathcal{G} \\ \sum_{s' \in \mathcal{S}} \eta(s, s') \cdot \Pr_{\mathcal{M}}\big(s' \vDash \Diamond^{\leq n}\mathcal{G}\big) & \text{otherwise} \end{cases} \tag{2.2}$$

The inductive case (2.2) is intuitive: the probability of reaching a goal state within $n + 1$ steps is 1 if we are already in a goal state; otherwise, it is the weighted sum of the probabilities of reaching a goal state within n steps from each possible next state.

Then $\Pr_{\mathcal{M}}\big(\Diamond^{\leq n}\mathcal{G}\big)$, the probability of reaching a goal state within n steps when starting from a state drawn from the initial-state distribution, is defined as

$$\Pr_{\mathcal{M}}\big(\Diamond^{\leq n}\mathcal{G}\big) \coloneqq \sum_{s \in \mathcal{S}} \iota(s) \cdot \Pr_{\mathcal{M}}\big(s \vDash \Diamond^{\leq n}\mathcal{G}\big). \tag{2.3}$$

We introduce another (perhaps less common) way to calculate step-bounded reachability probabilities. We write $\Pr_{\mathcal{M}}\big(\mu \blacktriangleright \Diamond^{\leq n}\mathcal{G}\big)$ to denote the probability

of reaching a goal state within n steps when starting from a state drawn from distribution $\mu \in \mathsf{D}(\mathcal{S})$. It is defined inductively as follows:

$$\Pr_{\mathcal{M}}\left(\mu \blacktriangleright \Diamond^{\leq 0}\mathcal{G}\right) \coloneqq \sum_{s\in\mathcal{G}} \mu(s) \tag{2.4}$$

$$\Pr_{\mathcal{M}}\left(\mu \blacktriangleright \Diamond^{\leq n+1}\mathcal{G}\right) \coloneqq \left(\sum_{s\in\mathcal{G}} \mu(s)\right) + \Pr_{\mathcal{M}}\left(\sum_{s\notin\mathcal{G}} \mu(s)\eta(s) \blacktriangleright \Diamond^{\leq n}\mathcal{G}\right) \tag{2.5}$$

The inductive case (2.5) is intuitive: the probability of reaching a goal state within $n+1$ steps is the sum of (1) the probability of being in a goal state at the start, and (2) the probability of reaching a goal state within n steps after taking one step from a non-goal state. Notice that in the second term, the next-state distribution is given by $\sum_{s\notin\mathcal{G}} \mu(s)\eta(s)$, which is the weighted sum of the transition distributions from all non-goal states.

The correctness of this alternative way to calculate step-bounded reachability probabilities is established by Theorem 2.1 (and proven in the accompanying technical report [32]):

Theorem 2.1. $\Pr_{\mathcal{M}}\left(\Diamond^{\leq n}\mathcal{G}\right) = \Pr_{\mathcal{M}}\left(\iota \blacktriangleright \Diamond^{\leq n}\mathcal{G}\right)$.

Its connection to tensor probabilistic model checking will become clear later, but for now we simply note that we use $\Pr_{\mathcal{M}}\left(\mu \blacktriangleright \Diamond^{\leq n}\mathcal{G}\right)$ as a bridge to connect tensor-based reachability to the standard definition of reachability probabilities.

3 A Core Model-Specification Language

Directly defining a DTMC via the four-tuple formalism in Section 2 is intractable for complex systems. The state space typically grows exponentially with the number of variables, making such brute-force specification error-prone and difficult to maintain. As a result, probabilistic model checkers such as PRISM [29] and Storm [16] adopt high-level modeling languages that allow users to specify the system in a modular fashion.

In this section, we define a high-level, modular specification language that captures the core aspects of a DTMC modeling language (e.g., the `dtmc` dialect of the PRISM language [1]). This allows describing a system concisely as a collection of interacting modules rather than a monolithic state-transition matrix.

We call this core language PML. To place its tensor interpretation on a formal footing (Sections 4 and 5), we first define PML's syntax in Section 3.1 and its Markov-chain semantics in Section 3.2.

3.1 Syntax of PML

Figure 2 presents the syntax of PML.

A model M contains K modules. Each module m_k $(1 \leq k \leq K)$ declares a disjoint set of state variables X_k via a set of declarations D_k. The module further specifies its behavior via a set of commands C_k that update the state variables

$$
\begin{array}{rrcl}
\text{Model} & M & ::= & \textbf{model } m_1; \cdots; m_K \textbf{ goal } e \\
\text{Module} & m & ::= & \textbf{module } D\ C \\
\text{Declarations} & D & ::= & d_1; \cdots; d_n \\
\text{Declaration} & d & ::= & \textbf{var } x : [0 \textbf{ .. } n_{\max}] \textbf{ init } n_{\text{init}} \\
\text{Commands} & C & ::= & c_1; \cdots; c_n \\
\text{Command} & c & ::= & [a]\ g \to U \\
\text{Guard} & g & ::= & e \\
\text{Mixture of updates} & U & ::= & \theta_1 : u_1 + \cdots + \theta_n : u_n \\
\text{Update} & u & ::= & x_1 \coloneqq e_1\ \cdots\ x_n \coloneqq e_n \\
\text{Expression} & e & ::= & n \mid x \mid op(e_1, \ldots, e_n) \\
n \in \mathbb{N} & x \in \text{Variables} & & \theta \in [0, 1]
\end{array}
$$

Figure 2. PML syntax.

in X_k. While each module declares and updates only its own state variables, it can read the variables of all the modules in M.

A declaration d defines a state variable x, its domain (a bounded range of non-negative integers), and its initial value.

A command c consists of an action label a, a guard g, and a mixture U of updates. The action label allows commands from different modules to be synchronized. The guard g is a Boolean expression specifying a necessary (though not sufficient) condition for the command to be enabled.

A command may also be unlabeled (written [] in PRISM), in which case it is assigned a globally unique action label. As a simplification, we assume that all unlabeled commands have already been assigned globally unique action labels.

Each update u_i in U is associated with a probability θ_i, where $\sum_i \theta_i = 1$. Each update further consists of deterministic assignments to a subset of the module's declared state variables.

An expression e is either a constant n, a state variable x, or an n-ary operation over sub-expressions $e_1, \ldots, e_n$. As a simplification, we treat Boolean expressions as integer expressions where 0 represents false and non-zero values represent true.

The global model M also specifies the goal states via a Boolean expression e over the state variables of all modules.

3.2 Semantics of PML

We now define the standard semantics of PML by interpreting a model M as a DTMC $\mathcal{M} = (\mathit{State}[\![M]\!], \mathit{Init}[\![M]\!], \mathit{Step}[\![M]\!], \mathit{Goal}[\![M]\!])$.

Let X_k be the set of state variables declared in module m_k ($1 \le k \le K$). Let $\mathsf{Vars}(M) \coloneqq \biguplus_{k=1}^{K} X_k$ be the set of all state variables in M. Let $\mathsf{dom}(x)$ denote the domain of variable x. For $X \subseteq \mathsf{Vars}(M)$, the state space formed by the variables in X is

$$\mathsf{State}(X) \coloneqq (x \in X) \to \mathsf{dom}(x). \tag{3.1}$$

That is, a state is a mapping from each variable in X to a value in its domain. The state space of the entire model M is then defined as

$$\mathit{State}[\![M]\!] \coloneqq \mathsf{State}(\mathsf{Vars}(M)). \tag{3.2}$$

We will write State as a shorthand for $\mathit{State}[\![M]\!]$ when M is clear from context.

Notice that $\mathsf{State}(X)$ is measurable since it is a finite set. The size of $\mathsf{State}(X)$ is $|\mathsf{State}(X)| = \prod_{x \in X} |\mathsf{dom}(x)|$, which grows exponentially with the number of state variables in X. We write $s(x)$ to denote the value of variable x in state s.

Let $s_0 \in \mathsf{State}$ be the initial state where each variable $x \in \mathsf{Vars}(M)$ is initialized to the value specified in its declaration. Then the initial-state distribution is

$$\mathit{Init}[\![M]\!] \coloneqq \delta_{s_0} \tag{3.3}$$

where δ_s is the delta distribution (point mass) at s.

Let e be the goal expression of M. Then the set of goal states is defined as

$$\mathit{Goal}[\![M]\!] \coloneqq \left\{ s \in \mathsf{State} \;\middle|\; [\![e]\!](s) \neq 0 \right\} \tag{3.4}$$

where $[\![e]\!]\, s$ is the interpretation of expression e under state s.

All that remains is to define $[\![e]\!]$ and $\mathit{Step}[\![M]\!]$. To define $\mathit{Step}[\![M]\!]$, we need to first define the semantics of each module m_k in M, which further requires defining the semantics of each command in m_k and each update in a command.

Interpreting expressions. $[\![e]\!] : \mathsf{State}(X) \to \mathbb{N}$, where X is a superset of the variables appearing in e, is defined inductively as follows:

$$[\![n]\!]\, s \coloneqq n \qquad [\![x]\!]\, s \coloneqq s(x) \qquad \frac{[\![e_i]\!]\, s = n_i \text{ for } i = 1, \ldots, l \quad [\![op]\!] = f}{[\![op(e_1, \cdots, e_l)]\!]\, s \coloneqq f(n_1, \ldots, n_l)}$$

Interpreting updates. Recall that X_k denotes the set of state variables declared in m_k. Let Y_k be the set of variables each of which is either declared in m_k or read in m_k's commands. So we have that $X_k \subseteq Y_k \subseteq \mathsf{Vars}(M)$.

An update u in module m_k is interpreted as a function $[\![u]\!] : \mathsf{State}(Y_k) \to \mathsf{State}(X_k)$. That is, for $s \in \mathsf{State}(Y_k)$,

$$[\![x_1 \coloneqq e_1 \;\cdots\; x_n \coloneqq e_n]\!]\, s \coloneqq s|_{X_k}[x_1 \mapsto [\![e_1]\!]s,\; \cdots,\; x_n \mapsto [\![e_n]\!]s]. \tag{3.5}$$

In (3.5), the notation $s|_{X_k}$ denotes the restriction of s to the variables in X_k, and $s|_{X_k}[x_1 \mapsto [\![e_1]\!]s,\; \cdots,\; x_n \mapsto [\![e_n]\!]s]$ denotes the state obtained by updating $s|_{X_k}$ with the assignments in u.

Interpreting mixtures of updates. A mixture of updates U in module m_k is interpreted as a function $[\![U]\!] : \mathsf{State}(Y_k) \to \mathsf{D}(\mathsf{State}(X_k))$. That is,

$$[\![\theta_1 : u_1 + \cdots + \theta_n : u_n]\!]\, s \coloneqq \sum_{i=1}^{n} \theta_i \cdot \delta_{[\![u_i]\!]s}. \tag{3.6}$$

Interpreting modules. Let $\mathsf{Action}(M)$ be the model M's alphabet of action labels. Let $\mathsf{Action}(m_k)$ be the set of action labels appearing in m_k's commands. The module m_k is interpreted as a function $[\![m_k]\!] : \mathsf{Action}(M) \to \mathsf{State}(Y_k) \to \mathsf{D}_{<\infty}(\mathsf{State}(X_k))$. Here, $\mathsf{D}_{<\infty}(\mathsf{State}(X_k))$ is the set of unnormalized distributions over $\mathsf{State}(X_k)$; these distributions have finite total mass, but the total mass is not necessarily 1. Specifically,

$$[\![m_k]\!]\, a\, s \coloneqq \begin{cases} \delta_{s|_{X_k}} & \text{if } a \notin \mathsf{Action}(m_k), \\ \sum_{[a]\, g \to U \,\in\, \mathsf{Com}(m_k, a)} [[\![g]\!]\, s \neq 0] \cdot [\![U]\!]\, s & \text{otherwise.} \end{cases} \tag{3.7}$$

In (3.7), $\mathsf{Com}(m_k, a)$ is the set of commands in module m_k with action label a, and $[P]$ is the indicator function that evaluates to 1 if predicate P is true and 0 otherwise.

Intuitively, $[\![m_k]\!]\, a\, s$ describes the aggregate transition behavior of module m_k when the current state is s and the chosen action is a. There are two cases. If a is not in m_k's alphabet, then m_k stays put. Otherwise, a sum ranges over all commands with action label a whose guard is satisfied by s. When exactly one such command $[a]\, g \to U$ exists, $[\![m_k]\!]\, a\, s$ is the distribution denoted by $[\![U]\!]\, s$. When multiple commands with the action label a have overlapping guards, $[\![m_k]\!]\, a\, s$ is an unnormalized distribution whose total mass equals the number of enabled commands; PRISM allows this ambiguity and resolves it by a random choice among all the enabled command combinations (see (3.10)). If no command with action label a is enabled in s, $[\![m_k]\!]\, a\, s$ has total mass 0.

Interpreting models. In a specification language like PRISM [1], the global behavior of a DTMC model is defined by the parallel execution of modules synchronized by action labels. A global transition step corresponds to the simultaneous transitioning of all modules according to a common action label.

Let $\mathit{En}[\![m_k]\!]\, a\, s$, where $s \in \mathsf{State}(Y_k)$, count the number of commands with action label a that are locally enabled in module m_k in state s:

$$\mathit{En}[\![m_k]\!]\, a\, s \coloneqq \begin{cases} 1 & \text{if } a \notin \mathsf{Action}(m_k), \\ \sum_{[a]\, g \to U \,\in\, \mathsf{Com}(m_k, a)} [[\![g]\!]\, s \neq 0] & \text{otherwise,} \end{cases} \tag{3.8}$$

In case $a \notin \mathsf{Action}(m_k)$, it is considered that there is one enabled command—the implicit self-loop—so $\mathit{En}[\![m_k]\!]\, a\, s$ is defined to be 1.

Let $\mathit{En}[\![M]\!]\, a\, s$, where $s \in \mathsf{State}$, count the total number of enabled *command combinations* for action a across all modules:

$$\mathit{En}[\![M]\!]\, a\, s \coloneqq \prod_{k=1}^{K} \mathit{En}[\![m_k]\!]\, a\, s|_{Y_k}. \tag{3.9}$$

Now we can define the DTMC transition function $\mathit{Step}[\![M]\!] : \mathsf{State} \to \mathsf{D}(\mathsf{State})$:

$$\mathit{Step}[\![M]\!]\, s \coloneqq \begin{cases} \dfrac{\sum_a \bigotimes_{k=1}^{K} [\![m_k]\!]\, a\, s|_{Y_k}}{\sum_a \mathit{En}[\![M]\!]\, a\, s} & \text{if } \displaystyle\sum_a \mathit{En}[\![M]\!]\, a\, s > 0, \\ \delta_s & \text{otherwise.} \end{cases} \tag{3.10}$$

The cases in (3.10) depend on whether there is at least one enabled command combination in state s.

– In PRISM, multiple commands—possibly with different action labels, or with the same label but overlapping guards—may be simultaneously enabled in a state for a module. This ambiguity is resolved by a uniformly random choice over all enabled command combinations across all modules.

  A command combination for action a consists of one enabled command from each module; the number of such command combinations is $\mathit{En}[\![M]\!]\, a\, s = \prod_k \mathit{En}[\![m_k]\!]\, a\, s|_{Y_k}$. The denominator $\sum_a \mathit{En}[\![M]\!]\, a\, s$ in (3.10) is the total number of enabled command combinations across all actions.

  The operator $\bigotimes$ in (3.10) denotes the product of (unnormalized) distributions over disjoint sets of variables. Since the K disjoint sets of variables $X_1, \ldots, X_K$ aggregate to $\mathsf{Vars}(M)$, $\bigotimes_{k=1}^{K} [\![m_k]\!]\, a\, s|_{Y_k}$ is an (unnormalized) distribution over the global state space State. Moreover, its total mass equals $\mathit{En}[\![M]\!]\, a\, s$.

– If no action is enabled in s, the DTMC loops back to s.

While $\bigotimes_{k=1}^{K} [\![m_k]\!]\, a\, s|_{Y_k}$ is in general an unnormalized probability distribution, $\mathit{Step}[\![M]\!]\, s$ is a probability distribution. We prove this fact (Theorem 3.1) in the technical report [32].

Theorem 3.1. *For any* $s \in \mathsf{State}$, $\mathit{Step}[\![M]\!]\, s \in \mathsf{D}(\mathsf{State})$.

4 Probabilistic Model Checking as Tensor Computations

In this section, we show how to cast the verification of PML models as computations over tensors. Section 4.1 maps state distributions to tensors. Section 4.2 maps PML models to tensor transformers. Section 4.3 maps the verification of step-bounded reachability properties to tensor computations.

4.1 Representing State Distributions as Tensors

We use tensors to represent discrete probability distributions over DTMC states. A tensor is a multidimensional array generalizing scalars (order-0), vectors (order-1), matrices (order-2), and so on. In this work, we use tensors $\boldsymbol{T} \in \mathbb{R}^{a_1 \times \cdots \times a_N}$ over the field of real numbers $\mathbb{R}$ and tensors $\boldsymbol{T} \in \mathbb{N}^{a_1 \times \cdots \times a_N}$ over the field of natural numbers $\mathbb{N}$, where N is the *order* (a.k.a. *rank*) of the tensor and a_k is the size of the k-th dimension.

The state space as an index space for tensors. A state space $\mathsf{State}(X) = (x \in X) \to \mathsf{dom}(x)$ serves as the index space for order-$|X|$ tensors:

– each dimension corresponds to a state variable $x \in X$,
– and the size of each dimension is $|\mathsf{dom}(x)|$.

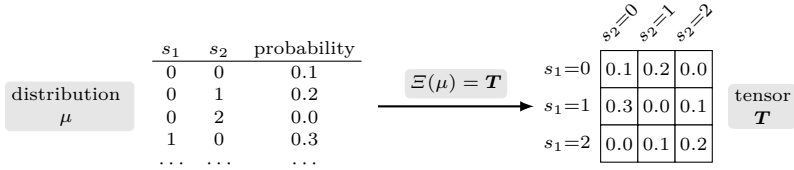
We write $\mathsf{Tensor}_{\mathbb{R}}(\mathsf{State}(X))$ to denote the set of $\mathbb{R}$-valued order-$|X|$ tensors with the index space $\mathsf{State}(X)$. That is, $\mathsf{Tensor}_{\mathbb{R}}(\mathsf{State}(X)) := \mathbb{R}^{\prod_{x \in X} |\mathsf{dom}(x)|}$. For $\boldsymbol{T} \in \mathsf{Tensor}_{\mathbb{R}}(\mathsf{State}(X))$, $\boldsymbol{T}[s]$ denotes the entry of $\boldsymbol{T}$ at index $s \in \mathsf{State}(X)$.

$\mathsf{Tensor}_{\mathbb{R}}(\mathsf{State}(X))$ is a *vector space* (a.k.a. *linear space*) under element-wise addition and scalar multiplication. This $|X|$-dimensional vector space has a natural set of basis vectors $\left\{ \boldsymbol{B}_s \;\middle|\; s \in \mathsf{State}(X) \right\}$, where each basis vector $\boldsymbol{B}_s \in \mathsf{Tensor}_{\mathbb{R}}(\mathsf{State}(X))$ is a tensor defined as $\boldsymbol{B}_s[s'] \coloneqq [s' = s]$.

State distributions as tensors. We define a function $\Xi : \mathsf{D}_{<\infty}(\mathsf{State}(X)) \to \mathsf{Tensor}_{\mathbb{R}}(\mathsf{State}(X))$ mapping a distribution $\mu \in \mathsf{D}_{<\infty}(\mathsf{State}(X))$ to a tensor:

$$\Xi(\mu) \coloneqq \sum_{s \in \mathsf{State}(X)} \mu(s) \cdot \boldsymbol{B}_s.$$

That is, $\Xi(\mu) = \boldsymbol{T}$ iff $\forall s \in \mathsf{State}(X)$, $\boldsymbol{T}[s] = \mu(s)$. For example, given a joint distribution μ of two state variables (say, the states of professors in Figure 1 when $N = 2$), Ξ maps μ to an order-2 tensor $\boldsymbol{T} \in \mathbb{R}^{3\times 3}$:



Each dimension corresponds to a professor's state. Both dimensions have size 3 (corresponding to states Away, Doodling, Done). This tensor should not be confused with the state-transition matrix used by a probabilistic model checker like Storm [16], which would be of size 9×9 (since there are 9 joint states).

Index tensors. For each variable $x \in X$, we define its *index tensor* $\boldsymbol{I}_{x \in X} \in \mathsf{Tensor}_{\mathbb{N}}(\mathsf{State}(X))$ as a $\mathbb{N}$-valued order-$|X|$ tensor. For any index $s \in \mathsf{State}(X)$, $\boldsymbol{I}_{x \in X}[s] = s(x)$ is simply the value of that variable in state s.

4.2 Tensor Semantics of PML Models

Interpreting expressions. Section 3.2 interprets an expression e as a function $[\![e]\!] : \mathsf{State}(X) \to \mathbb{N}$. We now define $(\!|e|\!)_X \in \mathsf{Tensor}_{\mathbb{N}}(\mathsf{State}(X))$ as follows, where the subscript X indicates the shape of the tensor:

$$(\!|n|\!)_X \coloneqq n \cdot \mathbf{1}_X \qquad (\!|x|\!)_X \coloneqq \boldsymbol{I}_{x \in X} \qquad \frac{(\!|e_i|\!)_X = \boldsymbol{T_i} \text{ for } i = 1, \ldots, l \quad (\!|op|\!)_X = f}{(\!|op(e_1, \cdots, e_l)|\!)_X \coloneqq f(\boldsymbol{T_1}, \ldots, \boldsymbol{T_l})}$$

where $\mathbf{1}_X \in \mathsf{Tensor}_{\mathbb{N}}(\mathsf{State}(X))$ is the all-ones tensor, and $(\!|op|\!)_X$ is $[\![op]\!]$ lifted to operate element-wise over tensors.

Interpreting updates. Section 3.2 interprets an update u in module m_k as a function $[\![u]\!] : \mathsf{State}(Y_k) \to \mathsf{State}(X_k)$. So given $s \in \mathsf{State}(Y_k)$, $[\![u]\!]\, s$ gives the value of $x \in X_k$ after executing u in state s. This reading suggests that we can define the tensor interpretation of u as a function $(\!|u|\!)_{Y_k} : X_k \to \mathsf{Tensor}_{\mathbb{N}}(\mathsf{State}(Y_k))$:

$$(\!|x_1 \coloneqq e_1 \;\cdots\; x_n \coloneqq e_n|\!)_{Y_k}\, x \coloneqq \begin{cases} (\!|e_i|\!)_{Y_k} & \text{if } x = x_i \text{ for some } i, \\ \boldsymbol{I}_{x \in Y_k} & \text{otherwise.} \end{cases} \tag{4.1}$$

Interpreting mixtures of updates. Section 3.2 interprets a mixture of updates U in module m_k as a function $[\![U]\!] : \mathsf{State}(Y_k) \to \mathsf{D}(\mathsf{State}(X_k))$. We now interpret U as a tensor $(\!|U|\!) \in \mathsf{Tensor}_{\mathbb{R}}(\mathsf{State}(Y_k \uplus X_k))$, where $\uplus$ is the disjoint-union operator. The tensor $(\!|U|\!)$ is of order $|Y_k| + |X_k|$. Specifically,

$$(\!|\theta_1 : u_1 + \cdots + \theta_n : u_n|\!) \coloneqq \sum_{i=1}^{n} \theta_i \cdot \bigodot_{x \in X_k} \Big[\Big((\!|u_i|\!)_{Y_k} \, x \Big) \otimes \mathbf{1}_{X_k} = \mathbf{1}_{Y_k} \otimes \boldsymbol{I}_{x \in X_k} \Big], \tag{4.2}$$

where $\bigodot$ is the Hadamard (i.e., element-wise) product, $\otimes$ is the outer product, and $=$ is overloaded to denote element-wise equality. Observe the correspondence between (3.6) and (4.2). The outer products in (4.2) are used to align the tensor dimensions. Intuitively, if we write (s, s') as an index, where $s \in \mathsf{State}(Y_k)$ and $s' \in \mathsf{State}(X_k)$, then we should have $(\!|\theta_1 : u_1 + \cdots + \theta_n : u_n|\!)[(s, s')] = \sum_{i=1}^{n} \theta_i \cdot [\![u_i]\!] s = s']$. That is, the tensor's value at index (s, s') is the probability of transitioning to state s' from state s when the mixture of updates U is executed.

Interpreting modules. Section 3.2 interprets a module m_k as a function $[\![m_k]\!] : \mathsf{Action}(M) \to \mathsf{State}(Y_k) \to \mathsf{D}_{<\infty}(\mathsf{State}(X_k))$. Accordingly, we define the tensor interpretation $(\!|m_k|\!) : \mathsf{Action}(M) \to \mathsf{Tensor}_{\mathbb{R}}(\mathsf{State}(Y_k \uplus X_k))$ as follows:

$$(\!|m_k|\!)\, a \coloneqq \begin{cases} \bigodot_{x \in X_k} \big[\boldsymbol{I}_{x \in Y_k} \otimes \mathbf{1}_{X_k} = \mathbf{1}_{Y_k} \otimes \boldsymbol{I}_{x \in X_k} \big] & \text{if } a \notin \mathsf{Action}(m_k), \\ \sum_{[a] g \to U \, \in \, \mathsf{Com}(m_k, a)} \Big(\Big[(\!|g|\!)_{Y_k} \neq 0 \Big] \otimes \mathbf{1}_{X_k} \Big) \odot (\!|U|\!) & \text{otherwise.} \end{cases} \tag{4.3}$$

Observe the correspondence between (3.7) and (4.3). Intuitively, $(\!|m_k|\!)\, a$ is a tensor whose value at index (s, s') is the unnormalized probability of transitioning to state s' from state s when m_k is executed with action a. When multiple commands have overlapping guards, the sum $\sum_{s'} (\!|m_k|\!)\, a[(s, s')]$ equals the number of enabled commands in state s; normalization to a probability distribution occurs at the model level rather than at the module level (see (4.6)).

Interpreting models. Section 3.2 interprets a model M as a function $\mathit{Step}[\![M]\!] : \mathsf{State} \to \mathsf{D}(\mathsf{State})$. Following the way that modules are interpreted as tensors, we could interpret M as a tensor $(\!|M|\!) : \mathsf{Tensor}_{\mathbb{R}}(\mathsf{State}(\mathsf{Vars}(M) \uplus \mathsf{Vars}(M)))$. However, this tensor would be of order $2|\mathsf{Vars}(M)|$, effectively materializing the full state-transition matrix. For space efficiency, we instead interpret M as a function $(\!|M|\!) : \mathsf{Tensor}_{\mathbb{R}}(\mathsf{State}) \to \mathsf{Tensor}_{\mathbb{R}}(\mathsf{State})$, representing the transition dynamics as a tensor transformer (i.e., code) rather than as a tensor (i.e., data).

Corresponding to $\mathit{En}[\![m_k]\!]\, a\, s$ and $\mathit{En}[\![M]\!]\, a\, s$ in (3.8) and (3.9), we define tensors $\mathit{En}(\!|m_k|\!)\, a \in \mathsf{Tensor}_{\mathbb{N}}(\mathsf{State}(Y_k))$ and $\mathit{En}(\!|M|\!)\, a \in \mathsf{Tensor}_{\mathbb{N}}(\mathsf{State})$:

$$\mathit{En}(\!|m_k|\!)\, a \coloneqq \begin{cases} \mathbf{1}_{Y_k} & \text{if } a \notin \mathsf{Action}(m_k), \\ \sum_{[a]\, g \to U \, \in \, \mathsf{Com}(m_k, a)} \Big[(\!|g|\!)_{Y_k} \neq 0 \Big] & \text{otherwise,} \end{cases} \tag{4.4}$$

$$\mathit{En}(\!|M|\!)\, a[s] \coloneqq \prod_{k=1}^{K} \mathit{En}(\!|m_k|\!)\, a\big[s|_{Y_k}\big] \tag{4.5}$$

Define $\boldsymbol{L} \coloneqq \sum_a En(\!|M|\!)\, a$ as the tensor counting the total number of enabled command combinations in each state.

We define $(\!|M|\!)$ by specifying how it transforms a tensor $\boldsymbol{T} \in \mathsf{Tensor}_{\mathbb{R}}(\mathsf{State})$ representing the current-state distribution into a tensor representing the next-state distribution. Specifically, for any $s' \in \mathsf{State}$, $(\!|M|\!)\,\boldsymbol{T}$ weights the probability of transitioning to s' from each state s by the probability $\boldsymbol{T}[s]$ of being in s:

$$(\!|M|\!)\,\boldsymbol{T})[s'] \coloneqq \sum_{s \in \mathsf{State}} \boldsymbol{T}[s] \cdot \begin{cases} \frac{\sum_a \prod_{k=1}^{K} (\!|m_k|\!)\, a\big[(s|_{Y_k}, s'|_{X_k})\big]}{\boldsymbol{L}[s]} & \text{if } \boldsymbol{L}[s] > 0, \\ [s = s'] & \text{otherwise.} \end{cases} \tag{4.6}$$

The transition probability is given by a case analysis, similarly to (3.10). The branching control flow in (4.6) hinders parallelization over the index space State, however. Fortunately, we can encode the branching logic as pure tensor flow, through a sum of two terms. We redefine $(\!|M|\!)$ as follows:

$$(\!|M|\!)\,\boldsymbol{T} \coloneqq \left(\sum_a \boldsymbol{P}_a\right) + \big(\boldsymbol{T} \odot \big[\boldsymbol{L} = \boldsymbol{0}_{\mathsf{Vars}(M)}\big]\big) \tag{4.7}$$

$$\boldsymbol{P}_a[s'] \coloneqq \sum_{s \in \mathsf{State}} \left(\frac{\boldsymbol{T}}{\max\big(\boldsymbol{L}, \boldsymbol{1}_{\mathsf{Vars}(M)}\big)}\right)[s] \cdot \prod_{k=1}^{K} (\!|m_k|\!)\, a\big[(s|_{Y_k}, s'|_{X_k})\big] \tag{4.8}$$

The two terms in (4.7) correspond to the two branches of (4.6).

- The first term $\sum_a \boldsymbol{P}_a$ rearranges the first branch of (4.6): it pushes $\sum_s$ inside $\sum_a$ and weights the input tensor $\boldsymbol{T}$ by $\boldsymbol{1} \oslash \max(\boldsymbol{L}, \boldsymbol{1})$, as required by the averaging semantics. In particular, when $\boldsymbol{L}[s] = 0$, the normalization factor $\max(\boldsymbol{L}[s], 1) = 1$, while $(\!|m_k|\!)\, a\big[(s|_{Y_k}, \cdot)\big]$ is zero for some module k for every action a, so the term vanishes.

  Notice that $\boldsymbol{P}_a$ as defined in (4.8) is a tensor contraction over dimensions s corresponding to the current state. The output dimensions s' correspond to the next state. The tensors in this contraction are the masked input tensor and the K module tensors $(\!|m_k|\!)\, a$. This tensor contraction is the main work performed by the tensor transformer $(\!|M|\!)$.
- The second term $\boldsymbol{T} \odot [\boldsymbol{L} = \boldsymbol{0}]$ weights the input tensor $\boldsymbol{T}$ by the mask $[\boldsymbol{L} = \boldsymbol{0}]$. This mask is 1 exactly where no actions are enabled, leaving the probability mass in those states unchanged and zeroing out the mass in all other states.

4.3 Casting Probabilistic Model Checking as Tensor Computations

With the tensor-transformer interpretation defined, we can now cast probabilistic model checking as tensor computations. In words, step-bounded reachability probabilities can be computed via repeated applications of the tensor transformer $(\!|M|\!)$. Specifically, for model M with goal expression e, we write $\Pr_M\big(\boldsymbol{T} \boxplus \Diamond^{\leq n} e\big)$ to denote the probability of reaching a state satisfying expression e within n steps when starting from a state drawn from the distribution represented by

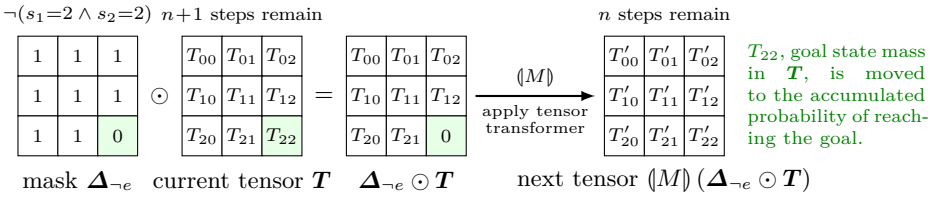


Figure 3. Visualizing the computation in (4.10) of the reachability probability. The probability mass T_{22} at the goal state $(s_1 = 2, s_2 = 2)$ is extracted and accumulated to the reachability probability. The remaining mass is transformed by the model's tensor-transformer interpretation to produce the next-state tensor.

tensor $\boldsymbol{T} \in \mathsf{Tensor}_{\mathbb{R}}(\mathsf{State})$. It is defined inductively as follows, where $\boldsymbol{\Delta}_e := \left[\!\left[e\right]\!\right]_{\mathsf{Vars}(M)} \neq \mathbf{0}_{\mathsf{Vars}(M)}\right]$ and $\boldsymbol{\Delta}_{\neg e} := \left[\!\left[e\right]\!\right]_{\mathsf{Vars}(M)} = \mathbf{0}_{\mathsf{Vars}(M)}\right]$ are mask tensors:

$$\Pr_M\left(\boldsymbol{T} \boxplus \Diamond^{\leq 0} e\right) := \langle \boldsymbol{\Delta}_e, \boldsymbol{T} \rangle \tag{4.9}$$

$$\Pr_M\left(\boldsymbol{T} \boxplus \Diamond^{\leq n+1} e\right) := \langle \boldsymbol{\Delta}_e, \boldsymbol{T} \rangle + \Pr_M\left(\left(\!\left|M\right|\!\right)\left(\boldsymbol{\Delta}_{\neg e} \odot \boldsymbol{T}\right) \boxplus \Diamond^{\leq n} e\right) \tag{4.10}$$

The definition mirrors that of $\Pr_{\mathcal{M}}\left(\mu \blacktriangleright \Diamond^{\leq n} \mathcal{G}\right)$ in (2.4) and (2.5). In the base case (4.9), the probability of being in a goal state is given by the Frobenius inner product $\langle \boldsymbol{\Delta}_e, \boldsymbol{T} \rangle = \sum_{s \in \mathsf{State}} \boldsymbol{\Delta}_e[s] \cdot \boldsymbol{T}[s]$. In the inductive case (4.10), the next-state tensor is given by $\left(\!\left|M\right|\!\right)\left(\boldsymbol{\Delta}_{\neg e} \odot \boldsymbol{T}\right)$, which is the result of applying the tensor transformer $\left(\!\left|M\right|\!\right)$ to the current-state tensor masked by $\boldsymbol{\Delta}_{\neg e}$. The mask tensor $\boldsymbol{\Delta}_{\neg e}$ ensures that only the non-goal states in the current-state tensor contribute to the next-state tensor. Figure 3 illustrates (4.10) for the 2-professor example.

4.4 Discussion

The correctness of $\Pr_M\left(\boldsymbol{T} \boxplus \Diamond^{\leq n} e\right)$ with respect to the verification problem stated in Section 2 will be established in Section 5. The close correspondence between the definitions of $\Pr_{\mathcal{M}}\left(\mu \blacktriangleright \Diamond^{\leq n} \mathcal{G}\right)$ and $\Pr_M\left(\boldsymbol{T} \boxplus \Diamond^{\leq n} e\right)$ makes short work of proving the correctness of tensor probabilistic model checking.

Two factors contribute to the efficiency of tensor probabilistic model checking. One factor is that the computations in (4.7), (4.8), (4.9), and (4.10) avoid materializing the full $3^N \times 3^N$ transition matrix of the DTMC (using the N-professor example for concreteness). Instead, the transition matrix is implicitly encoded as tensor computations (i.e., code rather than data) that transform the order-N tensors.

A more important factor contributing to the efficiency is that the tensor computations in (4.7), (4.8), (4.9), and (4.10) are composed of standard operations over dense tensors and, therefore, can be implemented as first-order array programs in an array programming language such as JAX [9]. In other words, we have essentially compiled the probabilistic model checking problem for DTMCs

into array programs of the kind that are otherwise ubiquitous in machine learning. These array programs can then be optimized by machine-learning compilers and exploit the massive parallelism offered by hardware accelerators such as GPUs.

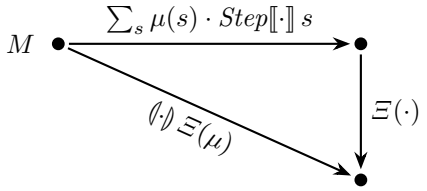
Finally, compiling to tensor computations makes it possible to search for model parameters satisfying reachability properties, via gradient-based optimization. Since the generated array programs are differentiable, we can compute gradients of distributional properties with respect to model parameters using automatic differentiation, which is readily supported by JAX.

5 Correctness of Tensor Probabilistic Model Checking

In this section, we establish the correctness of tensor probabilistic model checking.

In the following, let M be a PML model, and let $\mu \in \mathsf{D}(\mathit{State}[\![M]\!])$. Theorem 5.1 establishes the correctness of the tensor-transformer interpretation $(\!|M|\!)$ with respect to the standard interpretation $\mathit{Step}[\![M]\!]$.

Theorem 5.1. $(\!|M|\!)\,\Xi(\mu) = \Xi(\sum_s \mu(s) \cdot \mathit{Step}[\![M]\!]\, s)$. *That is, the diagram below commutes.*



Let $\mathcal{M} = (\mathit{State}[\![M]\!], \mathit{Init}[\![M]\!], \mathit{Step}[\![M]\!], \mathit{Goal}[\![M]\!])$ be the DTMC represented by M per Section 3.2. Let e be the goal expression specified in M. Let $\mathcal{G} := \mathit{Goal}[\![M]\!] = \left\{ s \in \mathit{State}[\![M]\!] \;\middle|\; [\![e]\!](s) \neq 0 \right\}$ be the set of goal states in $\mathcal{M}$. Theorem 5.2 follows from Theorem 5.1.

Theorem 5.2. $\Pr_M\big(\Xi(\mu) \boxplus \Diamond^{\leq n} e\big) = \Pr_{\mathcal{M}}\big(\mu \blacktriangleright \Diamond^{\leq n} \mathcal{G}\big)$ *for any* $n \in \mathbb{N}$.

Proofs of Theorem 5.1 and Theorem 5.2 are available in the technical report [32].

Theorem 5.3 establishes the ultimate correctness of tensor probabilistic model checking with respect to step-bounded reachability probabilities.

Theorem 5.3. $\Pr_M\big(\Xi(\mathit{Init}[\![M]\!]) \boxplus \Diamond^{\leq n} e\big) = \Pr_{\mathcal{M}}\big(\Diamond^{\leq n} \mathcal{G}\big)$ *for any* $n \in \mathbb{N}$.

Proof. Let $\iota = \mathit{Init}[\![M]\!]$ be the initial-state distribution of the DTMC $\mathcal{M}$. Starting from the left-hand side:

$$\begin{aligned} \Pr_M\big(\Xi(\iota) \boxplus \Diamond^{\leq n} e\big) &= \Pr_{\mathcal{M}}\big(\iota \blacktriangleright \Diamond^{\leq n} \mathcal{G}\big) && \text{(by Theorem 5.2)} \\ &= \Pr_{\mathcal{M}}\big(\Diamond^{\leq n} \mathcal{G}\big) && \text{(by Theorem 2.1)} \end{aligned}$$

□

The proof of Theorem 5.3 reveals that $\Pr_{\mathcal{M}}\big(\iota \blacktriangleright \Diamond^{\leq n} \mathcal{G}\big)$ bridges tensor-based reachability and the standard definition of reachability probability.

6 Accelerating Tensor Computations with JAX and XLA

At this point, we have reduced the verification of DTMCs (with respect to step-bounded reachability properties) to dense tensor computations. But that alone does not guarantee efficiency.

Efficiency hinges on how fast the tensor computations can run. In the last decade, training and inference in machine learning (ML) have driven significant advances in compiler optimizations for tensor computations, as well as hardware acceleration for them. It is thus natural to leverage off-the-shelf ML compilers and hardware accelerators to speed up tensor probabilistic model checking.

Specifically, Tessa compiles DTMC models to array programs in JAX [9]. JAX, embedded in Python, is a popular array programming language for high-performance numerical computing and machine learning. A just-in-time compiler traces the JAX program. The resulting intermediate representation is then handed off to XLA (Accelerated Linear Algebra) [2], a domain-specific compiler designed to optimize tensor computations.

XLA performs whole-program optimizations that are critical for performance on modern hardware accelerators. Importantly, it applies kernel fusion, merging multiple element-wise operations (such as those in Sections 4.2 and 4.3) into a single GPU kernel. Fusion significantly reduces memory footprint and the pressure on memory bandwidth, as it avoids materializing intermediate results between operations to the GPU memory. Such optimizations enable our tensor-based verification algorithm to fully saturate the massive parallelism offered by GPUs.

Moreover, by compiling to JAX, Tessa leverages ML compiler optimizations *transparently*. The user does not write any GPU kernel code or manage GPU memory explicitly. Future improvements in ML compiler technology and hardware accelerators will directly benefit Tessa without changes to its implementation.

7 Evaluation

We have implemented Tessa in Python. In this section, we evaluate Tessa on two fronts: model checking and parameter synthesis.

7.1 Model Checking

Evaluation methodology. We adopt benchmarks directly from Rubicon [19]. These benchmarks consist of DTMC models with dense transition dynamics. Future work could expand the scope of the evaluation to include a more comprehensive set of benchmarks. Nevertheless, the current selection is already representative of challenging Markov chain verification tasks in the dense regime.

All experiments ran on a machine with an Intel Core i7-7820X CPU, 128 GB RAM, and an NVIDIA GeForce RTX 2080 Ti GPU (11 GB VRAM). At first glance, comparing CPU-bound methods against a GPU-accelerated tool might seem like comparing apples to oranges. However, this hardware distinction is precisely the point. Prevailing methods rely on representations that induce irregular memory access. Consequently, they do not map naturally to GPUs.

In contrast, Tessa generates dense tensor workloads at which GPUs excel. We are therefore comparing *methodologies* on the hardware they naturally map to—rather than comparing the hardware per se.

We note that the GPU used in our experiments is an older consumer-grade model in the hardware vendor's lineup. While we already observe substantial speedups over state-of-the-art tools, access to frontline hardware is expected to yield even greater gains.

Baseline methods. For Rubicon [19], the authors use Storm (Sparse) and Storm (MTBDD) as baselines in their evaluation. We include both, as well as Dice (via the Rubicon transpiler [19]).

- Storm (Sparse). In this engine of the Storm model checker, the transition dynamics of a DTMC model is represented as a sparse matrix in the standard *compressed sparse row* (CSR) format. The engine uses efficient sparse matrix–vector multiplication kernels from off-the-shelf libraries such as Eigen and Gmm++.
- Storm (MTBDD). This engine of Storm represents the DTMC transition dynamics using multi-terminal binary decision diagrams. Random access to entries of the transition matrix is not as efficient as in the sparse-matrix engine, but the MTBDD engine can be more memory-efficient for certain models.
- Dice. In this approach, the DTMC model specification is first lowered to the Dice probabilistic programming language [18] using the Rubicon transpiler [19]. The Dice compiler then represents the set of paths from the initial state to the goal states effectively as a BDD. This approach is shown to excel on models with certain structures.

All methods are configured to use the double-precision floating-point format.

Results. We now report results for each of the benchmarks on which the Rubicon/Dice method is evaluated [19]: Queues, Weather Factories, and Herman.

Queues. The Queues model consists of K queues, each with capacity 3. Tasks arrive probabilistically at every step. Three queues are designated type 1, while the remainder are type 2. The goal states are those in which all type 1 queues and at least one type 2 queue are full. We compute the probability of reaching a goal state within horizon H. Figure 4 shows how each tool scales with K (left two plots) and with H (right two plots).

The first plot compares all methods as K varies, with $H = 10$ fixed. All methods scale exponentially with K. On the hardest instance that any baseline method can solve ($K = 10$), Tessa shows over 100× speedup over the next fastest method.

The second plot zooms in on the performance of Tessa. Since JAX and XLA perform just-in-time (JIT) compilation, we measure two runs to show the effect of JIT compilation: in the second plot, the difference between the two Tessa curves indicates the JIT compilation overhead.[3]

[3] Technically, the Tessa (1st) curve also includes the time taken to compile the model specification into JAX. Since this compilation happens entirely in Python, we measure it after warm-up runs and add it to the 1st-run time.

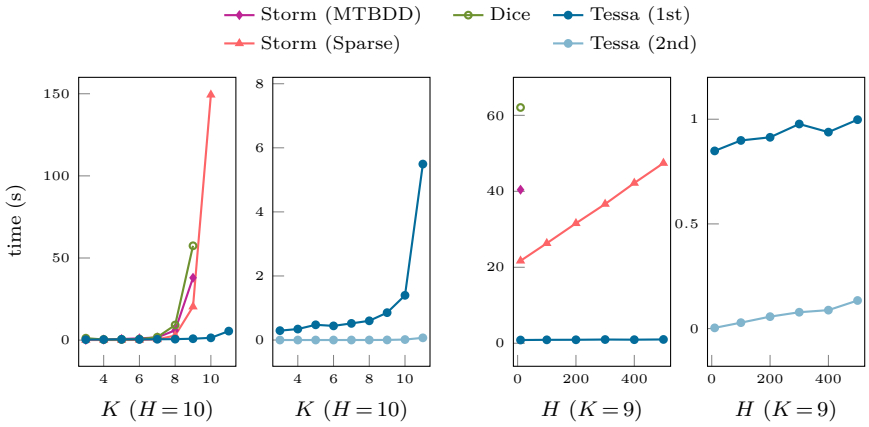


Figure 4. Scaling on the Queues benchmark.

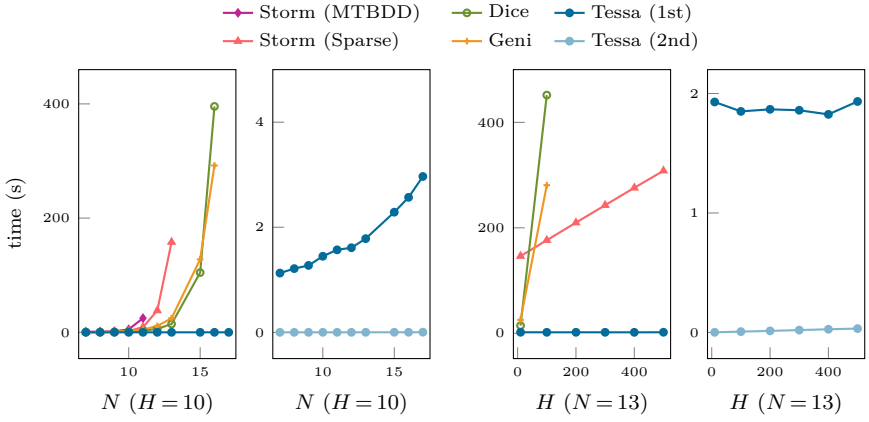


Figure 5. Scaling on the Weather Factories benchmark.

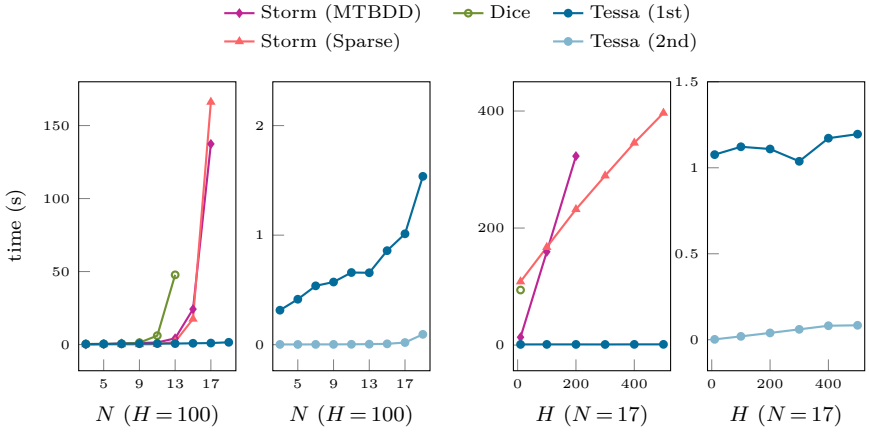


Figure 6. Scaling on the Herman benchmark.

The right two plots show how the methods scale as H varies, with $K = 9$ fixed. Storm (MTBDD) and Dice reach time limits at lower K values. Storm (Sparse) and Tessa scale linearly with H, but Tessa is ~40× faster at $H = 500$.

Weather Factories. This model consists of N factories, each in a binary state: striking or operational. Transition probabilities are local but conditioned on a Markov process, weather. We verify the reachability of the state where all factories are simultaneously striking within a given horizon H.

Figure 5 shows how each method scales on this model. We additionally include the Geni probabilistic programming language (PPL) as a baseline; the benchmark is part of its evaluation suite [33]. The methodology of using Geni for DTMC model checking is similar to that of Dice: both repurpose a PPL for DTMC model checking. The difference is that Geni's compiler targets generating functions, while Dice's compiler targets BDDs.

All methods scale exponentially with N. Unlike in the Queues benchmark, here Dice scales better than Storm (Sparse) as N increases. On the hardest instance that any baseline method can solve ($N = 16$), Tessa shows over 100× speedup over the next fastest method, Geni.

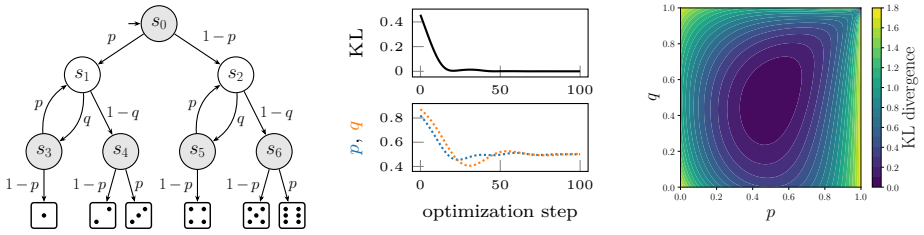
All methods (that run at $N = 13$) scale linearly with H. At $H = 500$, Tessa shows over 100× speedup over the next fastest method, Storm (Sparse).

Herman. Herman's protocol [17] is a well-known example in the literature of probabilistic model checking [30]. It is a randomized self-stabilization algorithm for leader election in a distributed ring of processors. We verify the probability that a system of N processors stabilizes within a horizon of H steps. Figure 6 shows how each method scales on this model.

Due to state explosion, all methods scale exponentially with N, but the effect is not felt by Tessa until a larger N. On the hardest instance that any baseline method can solve ($N = 17$), Tessa shows over 100× speedup over the next fastest method, Storm (MTBDD).

The right two plots in Figure 6 show that Tessa scales effectively with H as well. At $H = 500$, Tessa demonstrates over 300× speedup over the next fastest method, Storm (Sparse). While the speedup inherently includes the hardware advantage of a GPU, it underscores the value of mapping the verification problem to an accelerator-friendly representation.

Discussion. We caveat that Tessa outpaces these baseline methods for models that *fit* within the VRAM limit. For sparse models, Storm (Sparse) and Storm (MTBDD) are in general more space-efficient than Tessa. Nevertheless, that Tessa achieves these speedups under the 11 GB VRAM constraint indicates that the method is reasonably space-efficient for the class of models it targets (thanks to the Tessa implementation and XLA optimizations exploiting model structure to reduce memory footprint), whereas baseline methods may run out of memory or time out on the same dense models.



(a) (b) (c)

Figure 7. (a) A parametric DTMC encoding a Knuth–Yao die roller. In gray (resp. white) states, a coin of bias p (resp. q) is flipped. (b) KL divergence and parameter values as gradient descent progresses. (c) Optimization landscape over parameters p and q as a contour plot. As Figure 7(c) indicates, optimal values are $p = 0.5$ and $q = 0.5$, which are found by gradient descent as shown in Figure 7(b).

7.2 Parameter Search

We further evaluate Tessa on parameter synthesis using the Knuth–Yao algorithm [24]. This model generates a target distribution using coin flips. Figure 7(a) depicts the Markov chain [20, 21]. The task is to find the coin biases (p and q) that produce this target distribution. We formulate this search as an optimization problem. The objective is to minimize the Kullback–Leibler (KL) divergence [26] between the model's output and the target.

Tessa compiles the model into a differentiable tensor program, whose result can be programmatically composed with distributional properties including but not limited to state reachability—in this case, the KL divergence. The resulting objective can then be directly composed with JAX's automatic differentiation framework and gradient-based optimizers [11]. Figure 7(b) shows that, starting from random initialization, gradient descent successfully converges to the optimal parameter values well within 100 steps in a few seconds. This experiment highlights the *flexibility* with which Tessa can be used to optimize for distributional properties beyond state reachability.

8 Related Work

GPU-accelerated probabilistic model checking. Bošnački et al. [7, 8] study CUDA-accelerated sparse matrix–vector multiplication for DTMCs. Our approach differs fundamentally from Bošnački et al. in data representation, memory access patterns, engineering simplicity, and the class of models targeted. Bošnački et al.'s approach is best for DTMCs with sparse transition dynamics, representing the transition dynamics as a flattened sparse matrix. To mitigate the overhead of indirect memory access inherent of this storage format, they build on the modified CSR format [27] and develop custom CUDA kernels for sparse matrix–vector multiplication (SpMV). Češka et al. [34] adapt similar sparse-matrix techniques to parameter synthesis for continuous-time Markov chains through custom GPU

kernels. Also using sparse matrices, Heemstra and Wijs [15] perform explicit state space exploration as well as model checking entirely on the GPU.

In contrast, our approach is less suitable for sparse models but effective for dense ones. It compiles to JAX's dense tensor operations and does not require GPU programming. Rather than materializing the model into a sparse matrix (i.e., data), we map the model to tensor transformations (i.e., code), which XLA can fuse and optimize. This allows our system to make good use of the high-throughput dense linear algebra units on modern accelerators, which are often underutilized in sparse regimes.

Bak et al. [6] use GPUs to accelerate statistical model checking (SMC) of extended timed automata. SMC is fundamentally different from probabilistic model checking (PMC): while PMC computes exact probabilities by exhaustively exploring the state space, SMC estimates probabilities via sampling, thus trading accuracy for feasibility. Achieving low error margins in SMC is at the cost of high demands on computational resources for Monte Carlo simulations.

State explosion. Techniques for mitigating state explosion have been heavily studied. They either compress, abstract, or prune the state space [5, 4, 28, 23, 13, 14, 22, 12]. State-of-the-art probabilistic model checkers (e.g., Storm [16] and PRISM [29]) integrate such sophisticated state-space reduction techniques.

Compared to these techniques, Tessa takes an orthogonal approach. The translation to tensor computations is largely oblivious to the state-explosion issue. Rather, we rely on a tensor compiler for fusion and optimization. In a sense, we cast state-space reduction as compiler optimizations, offloading much of the complexity of extracting high performance to a mature compiler stack.

Distribution transformers. Our tensor transformer semantics is akin to the distribution transformer semantics of Kozen [25], which has found many uses in the analysis of probabilistic models (e.g., [3, 33]). We recast it and establish the formal ties between the tensor transformer semantics and DTMC model checking.

9 Conclusion

We cast model checking of finite-horizon Markov chains as dense tensor computations, for which compiler and hardware support is readily available. This new perspective delivers sizable performance gains for models with dense transition dynamics while maintaining mathematical soundness. We hope our approach makes a useful addition to the toolbox of probabilistic model checkers, extends their practical reach, and inspires future work on tensor-based verification methods.

Acknowledgments. We thank the anonymous reviewers for their valuable feedback. This work was supported in part by the Natural Sciences and Engineering Research Council of Canada. The views and opinions expressed are those of the authors and do not necessarily reflect the position of any funding agency.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

Data-Availability Statement. The artifact accompanying this paper is available at https://doi.org/10.5281/zenodo.19802567 [31]. The artifact includes the Tessa implementation, the benchmarks used in the evaluation, and instructions to reproduce the experimental results.

References

1. PRISM manual (2025), URL https://prismmodelchecker.org/manual, version 4.9
2. XLA: A machine learning compiler for GPUs, CPUs, and ML accelerators. https://github.com/openxla/xla (nd), accessed: 2026-01-10
3. S. Akshay, Krishnendu Chatterjee, Tobias Meggendorfer, Đorđe Žikelić: MDPs as distribution transformers: Affine invariant synthesis for safety objectives. In: Int'l Conf. on Computer Aided Verification (CAV) (2023), https://doi.org/10.1007/978-3-031-37709-9_5
4. Luca de Alfaro, Marta Z. Kwiatkowska, Gethin Norman, David Parker, Roberto Segala: Symbolic model checking of probabilistic processes using MTBDDs and the Kronecker representation. In: Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (2000), https://doi.org/10.1007/3-540-46419-0_27
5. Christel Baier, Edmund M. Clarke, Vasiliki Hartonas-Garmhausen, Marta Kwiatkowska, Mark Ryan: Symbolic model checking for probabilistic processes. In: Int'l Colloquium on Automata, Languages and Programming (ICALP) (1997), https://doi.org/10.1007/3-540-63165-8_199
6. Oliver S. Bak, Mathias W. B. Christiansen, Oliver V. Eriksen, Sergio Feo-Arenis, Peter G. Jensen, Marcus D. Jensen, Simas Juozapaitis, Kim G. Larsen, Marius Mikučionis, Marco Muñiz, Andreas Windfeld: GPU accelerating statistical model checking for extended timed automata. In: Principles of Verification: Cycling the Probabilistic Landscape (2025), https://doi.org/10.1007/978-3-031-75775-4_12
7. Dragan Bošnački, Stefan Edelkamp, Damian Sulewski, Anton Wijs: GPU-PRISM: An extension of PRISM for general purpose graphics processing units. In: Proc. of the Ninth Int'l Workshop on Parallel and Distributed Methods in Verification, and Second Int'l Workshop on High Performance Computational Systems Biology (2010), https://doi.org/10.1109/PDMC-HiBi.2010.11
8. Dragan Bošnački, Stefan Edelkamp, Damian Sulewski, Anton Wijs: Parallel probabilistic model checking on general purpose graphics processors. Int'l Journal on Software Tools for Technology Transfer (STTT) **13** (2011), https://doi.org/10.1007/s10009-010-0176-4
9. James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, Qiao Zhang: JAX: composable transformations of Python+NumPy programs (2018), URL http://github.com/jax-ml/jax

10. Carlos E. Budde, Christian Dehnert, Ernst Moritz Hahn, Arnd Hartmanns, Sebastian Junges, Andrea Turrini: JANI: Quantitative model and tool interaction. In: Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (2017), https://doi.org/10.1007/978-3-662-54580-5_9
11. DeepMind, Igor Babuschkin, Kate Baumli, Alison Bell, Surya Bhupatiraju, Jake Bruce, Peter Buchlovsky, David Budden, Trevor Cai, Aidan Clark, Ivo Danihelka, Antoine Dedieu, Claudio Fantacci, Jonathan Godwin, Chris Jones, Ross Hemsley, Tom Hennigan, Matteo Hessel, Shaobo Hou, Steven Kapturowski, Thomas Keck, Iurii Kemaev, Michael King, Markus Kunesch, Lena Martens, Hamza Merzic, Vladimir Mikulik, Tamara Norman, George Papamakarios, John Quan, Roman Ring, Francisco Ruiz, Alvaro Sanchez, Laurent Sartran, Rosalia Schneider, Eren Sezener, Stephen Spencer, Srivatsan Srinivasan, Miloš Stanojević, Wojciech Stokowiec, Luyu Wang, Guangyao Zhou, Fabio Viola: The DeepMind JAX Ecosystem (2020), URL http://github.com/google-deepmind
12. Tom Dijk, Jaco Pol: Multi-core symbolic bisimulation minimisation. In: Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (2016), https://doi.org/10.1007/978-3-662-49674-9_19
13. Ernst Moritz Hahn, Holger Hermanns, Björn Wachter, Lijun Zhang: PASS: Abstraction refinement for infinite probabilistic models. In: Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (2010), https://doi.org/10.1007/978-3-642-12002-2_30
14. Ernst Moritz Hahn, Holger Hermanns, Lijun Zhang: Probabilistic reachability for parametric Markov models. Int'l Journal on Software Tools for Technology Transfer (STTT) **13**(1) (2011), https://doi.org/10.1007/s10009-010-0146-x
15. Jan Heemstra, Anton Wijs: GPUexplore$^{\text{prob}}$: Markov chain state space construction and verification with GPUs. In: Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (2025), https://doi.org/10.1007/978-3-031-90660-2_8
16. Christian Hensel, Sebastian Junges, Joost-Pieter Katoen, Tim Quatmann, Matthias Volk: The probabilistic model checker Storm. Int'l Journal on Software Tools for Technology Transfer (STTT) (2022), https://doi.org/10.1007/s10009-021-00633-z
17. Ted Herman: Probabilistic self-stabilization. Information Processing Letters **35**(2) (1990), https://doi.org/10.1016/0020-0190(90)90107-9
18. Steven Holtzen, Guy Van den Broeck, Todd Millstein: Scaling exact inference for discrete probabilistic programs. Proc. of the ACM on Programming Languages (PACMPL) **4**(OOPSLA) (Nov 2020), https://doi.org/10.1145/3428208
19. Steven Holtzen, Sebastian Junges, Marcell Vazquez-Chanlatte, Todd Millstein, Sanjit A. Seshia, Guy Van den Broeck: Model checking finite-horizon Markov chains with probabilistic inference. In: Int'l Conf. on Computer Aided Verification (CAV) (2021), https://doi.org/10.1007/978-3-030-81688-9_27

20. Nils Jansen, Sebastian Junges, Joost-Pieter Katoen: Parameter synthesis in Markov models: A gentle survey. In: Principles of Systems Design: Essays Dedicated to Thomas A. Henzinger on the Occasion of His 60th Birthday (2022), https://doi.org/10.1007/978-3-031-22337-2_20
21. Sebastian Junges, Erika Ábrahám, Christian Hensel, Nils Jansen, Joost-Pieter Katoen, Tim Quatmann, Matthias Volk: Parameter synthesis for Markov models: covering the parameter space. Formal Methods in System Design **62** (Feb 2024), https://doi.org/10.1007/s10703-023-00442-x
22. Nishanthan Kamaleson, David Parker, Jonathan E. Rowe: Finite-horizon bisimulation minimisation for probabilistic systems. In: Model Checking Software (SPIN'16) (2016), https://doi.org/10.1007/978-3-319-32582-8_10
23. Joost-Pieter Katoen, Tim Kemna, Ivan Zapreev, David N. Jansen: Bisimulation minimisation mostly speeds up probabilistic model checking. In: Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (2007), https://doi.org/10.1007/978-3-540-71209-1_9
24. Donald E. Knuth, Andrew C. Yao: The complexity of nonuniform random number generation. In: Algorithms and Complexity: New Directions and Recent Results, Academic Press (1976)
25. Dexter Kozen: Semantics of probabilistic programs. Journal of Computer and System Sciences **22**(3) (1981), https://doi.org/10.1016/0022-0000(81)90036-2
26. Solomon Kullback, Richard A. Leibler: On information and sufficiency. The Annals of Mathematical Statistics **22**(1) (1951), https://doi.org/10.1214/aoms/1177729694
27. Marta Kwiatkowska, Rashid Mehmood: Out-of-core solution of large linear systems of equations arising from stochastic modelling. In: Proc. of the Second Joint Int'l Workshop on Process Algebra and Probabilistic Methods, Performance Modeling and Verification (2002), https://doi.org/10.1007/3-540-45605-8_9
28. Marta Kwiatkowska, Gethin Norman, David Parker: Symmetry reduction for probabilistic model checking. In: Int'l Conf. on Computer Aided Verification (CAV) (2006), https://doi.org/10.1007/11817963_23
29. Marta Kwiatkowska, Gethin Norman, David Parker: PRISM 4.0: Verification of probabilistic real-time systems. In: Int'l Conf. on Computer Aided Verification (CAV) (2011), https://doi.org/10.1007/978-3-642-22110-1_47
30. Marta Kwiatkowska, Gethin Norman, David Parker: Probabilistic verification of Herman's self-stabilisation algorithm. Formal Aspects of Computing **24** (Jul 2012), https://doi.org/10.1007/s00165-012-0227-6
31. Jianlin Li, Nick Guo, Peter Ye, Yizhou Zhang: Tensor probabilistic model checking of finite-horizon Markov chains (artifact) (2026), https://doi.org/10.5281/zenodo.19802567, also available at https://github.com/tessa-cav26-ae/tessa-cav26-ae

32. Jianlin Li, Nick Guo, Peter Ye, Yizhou Zhang: Tensor probabilistic model checking of finite-horizon Markov chains (extended version). Technical Report CS-2026-03, School of Computer Science, University of Waterloo (2026)
33. Jianlin Li, Yizhou Zhang: Compiling with generating functions. Proc. of the ACM on Programming Languages (PACMPL) **9**(ICFP) (Aug 2025), https://doi.org/10.1145/3747534
34. Milan Češka, Petr Pilař, Nicola Paoletti, Luboš Brim, Marta Kwiatkowska: PRISM-PSY: Precise GPU-accelerated parameter synthesis for stochastic systems. In: Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (2016), https://doi.org/10.1007/978-3-662-49674-9_21