# Approximate steady–state analysis of large Markov models based on the structure of their decision diagram encoding ⋆

Min Wan [a] Gianfranco Ciardo [a] Andrew S. Miner [b]

[a] *Dept. of Computer Science and Engineering, University of California, Riverside*

[b] *Dept. of Computer Science, Iowa State University*

**Abstract**

We propose a new *approximate numerical algorithm for the steady–state solution of general structured ergodic Markov models*. The approximation uses a state–space encoding based on multiway decision diagrams and a transition rate encoding based on a new class of *edge–valued decision diagrams*. The new method retains the favorable properties of a previously proposed Kronecker–based approximation, while eliminating the need for a Kronecker–consistent model decomposition. Removing this restriction allows for a greater utilization of event *locality*, which facilitates the generation of both the state–space and the transition rate matrix, thus extends the applicability of this algorithm to larger and more complex models.

*Key words:* Markov chains, decision diagrams, steady–state analysis, approximation, aggregation.

## 1 Introduction

Markov models are extensively used for performance and reliability analysis of discrete–state systems. Exact numerical methods for the stationary or transient solution of continuous–time Markov chains (CTMCs) with a finite state space have been extensively studied in the literature [24]. However, while most practical models are compactly described using some high–level formalism, their underlying CTMC may be so large that storing the transition rate

⋆ Work supported in part by the National Science Foundation under Grants CSR-0546041, CCF-0848463, and CCF-1018057.

*Email addresses:* mwan@cs.ucr.edu (Min Wan), ciardo@cs.ucr.edu (Gianfranco Ciardo), asminer@cs.iastate.edu (Andrew S. Miner).

matrix, as well as storing and computing the stationary or transient probability vector, may overwhelm even the largest computers. To deal with this *state explosion* problem, approximate techniques have been proposed, where some smaller–scale CTMCs are solved, and the results somehow combined.

In this paper, we focus on the approximate stationary solution of finite ergodic CTMCs, i.e., the computation of the steady–state probability vector, extending the approach of [20]. We decompose the overall model, use a *multiway decision diagram* (MDD) [23] to encode its state space, define *aggregate* CTMCs based on this MDD representation, iteratively solve them until we reach a fixpoint, and finally compute approximate stationary measures. As the aggregated CTMCs are relatively small, the storage and computational cost are enormously reduced.

The method of [20] was empirically shown to be efficient and have fast convergence, but it has a major limitation: it is applicable only to *Kronecker–consistent* models, i.e., the transition rate matrix must be expressed as the sum (over all events) of the Kronecker product (over all state variables) of *local* matrices [2, 5, 15]. While in principle this Kronecker form always exists, in practice it might be achieved only by splitting model events, resulting in an excessive number of events, or by merging state variables, resulting in excessively large *local* state spaces. This restriction may render the state–space *generation* inefficient or even infeasible, while a finer non-Kronecker decomposition would greatly improve the time and memory performance when used in conjunction with state–of–the–art symbolic state–space generation algorithms [13, 26].

To overcome this restriction, we propose to abandon the Kronecker approach in favor of an encoding of the transition rate matrix based on a new class of *edge–valued decision diagrams* [11]. Specifically, we adopt our new *multiplicative edge–valued multiway decision diagrams* (EV$^*$MDDs), which can compactly encode the transition rate matrix under an arbitrary model decomposition.

The remainder of the paper is organized as follows. Sect. 2 reviews the required background on MDDs and CTMC aggregation. Sect. 3 introduces EV$^*$MDDs and algorithms for their efficient manipulation, then shows how to encode the transition rate matrices with them. Sect. 4 details our approximation algorithm. Sect. 5 reports experimental results on a set of benchmarks. Finally, Sect. 6 concludes the paper. For reference, Figure 1 summarizes the symbols we use throughout the paper.

| Model description | | | |
|---|---|---|---|
| $\mathcal{X}, \mathcal{S}$ | set of states | $\mathcal{Z}, \mathcal{T}, \mathcal{T}_e$ | transition relation |
| $\mathbf{Q}$ | generator matrix | $\mathbf{R}, \mathbf{R}_e$ | transition rate matrix |
| $\boldsymbol{\pi}$ | probability vector | $f, f_e$ | rate function |
| $spt(f)$ | variables in $f$'s support | $\varphi, \varphi_e$ | probability function |
| $\mathbf{x}$ | tuple of state variables | $\mathbf{i}, \mathbf{i}', \mathbf{j}, \mathbf{j}'$ | a state |
| $x_k$ | $k^{th}$ state variable | $i_k, j_k$ | $k^{th}$ state component |
| $\mathcal{V}_k$ | domain of variable $x_k$ | $\mathcal{V}_p$ | domain of node $p$ |
| Decision diagrams | | | |
| $r^*$ | root node | $\rho^*$ | dangling edge's value |
| $\mathbf{0}, \mathbf{1}$ | MDD terminal nodes | $p, q, r$ | non–terminal node |
| $\Omega$ | EV$^*$MDD terminal node | $\langle \rho, r \rangle$ | edge with value $\rho$, destination $r$ |
| $r^*_{\mathcal{X}}$ | MDD encoding set $\mathcal{X}$ | $\langle \rho^*_f, r^*_f \rangle$ | EV$^*$MDD encoding function $f$ |
| $p[i_k]$ | $i^{th}_k$ edge of node $p$ | $\alpha, \beta, \gamma$ | path or tuple |
| $p[\alpha]$ | extends edge notation | $p[\alpha, \alpha']$ | path from $p$ interleaving $\alpha$, $\alpha'$ |
| $p[\alpha].v$ | edge/path value | $\mathcal{N}(p)$ | set of nodes at or under $p$ |
| $p[\alpha].d$ | edge/path destination | $\mathcal{N}(p, x_h)$ | nodes under $p$ associated to $x_h$ |
| Decision–diagram–based aggregation | | | |
| $\mathcal{A}(p)$ | $\{\alpha : r^*[\alpha] = p\}$ | $\mathcal{B}(p)$ | $\{\beta : p[\beta] = \mathbf{1}\}$ |
| $[\![p, i_k]\!]$ | $\mathcal{A}(p) \times \{i_k\} \times \mathcal{B}(p[i_k])$ | $\mathbf{R}_k$ | $\mathbf{R}$ aggregation w.r.t. $x_k$ |
| $\mathbf{Q}_k$ | $\mathbf{Q}$ aggregation w.r.t. $x_k$ | $\boldsymbol{\pi}_k$ | $\boldsymbol{\pi}$ aggregation w.r.t. $x_k$ |
| $\boldsymbol{\pi}[\mathcal{X}]$ | $\sum_{\mathbf{i} \in \mathcal{X}} \boldsymbol{\pi}[\mathbf{i}]$ | $\boldsymbol{\pi}[p]$ | $\boldsymbol{\pi}[\mathcal{A}(p) \times \mathcal{B}(p)]$ |
| $\boldsymbol{\pi}[p,\gamma]$ | $\boldsymbol{\pi}[\mathcal{A}(p) \times \{\gamma\} \times \mathcal{B}(p[\gamma])]$ | $\boldsymbol{\pi}[\gamma]$ | $\boldsymbol{\pi}[r^*, \gamma]$ |

Fig. 1. Symbols used in the paper.

## 2 Background

### *2.1 Model description*

Consider a *structured* CTMC with a finite state space $\mathcal{S} \subset \mathbb{N}^L$ and transition rate matrix $\mathbf{R} \in \mathbb{R}^{|\mathcal{S}| \times |\mathcal{S}|}$, whose state is a tuple $\mathbf{x} = (x_L, \ldots, x_1) \in \mathbb{N}^L$ of $L$

variables with a predefined order $x_L \succ \cdots \succ x_1$ imposed on them. We refer to the choice of these variables and their order as a *decomposition* of the CTMC. Each variable $x_k$ takes values in a finite set $\mathcal{V}_k = \{0, \ldots, n_k\} \subset \mathbb{N}$, which is also referred to as the *local state space* of $x_k$.

We seek to compute $\boldsymbol{\pi} \in \mathbb{R}^{|\mathcal{S}|}$, the solution of

$$\boldsymbol{\pi} \cdot \mathbf{Q} = 0 \quad \text{subject to} \quad \textstyle\sum_{\mathbf{i} \in \mathcal{S}} \boldsymbol{\pi}[\mathbf{i}] = 1,$$

where $\mathbf{Q} \in \mathbb{R}^{|\mathcal{S}| \times |\mathcal{S}|}$ is the generator matrix of the CTMC and $\boldsymbol{\pi}[\mathbf{i}]$ is the stationary probability of state $\mathbf{i}$. Matrix $\mathbf{Q}$ coincides with $\mathbf{R}$ except in its diagonal elements, which are given by $\mathbf{Q}[\mathbf{i}, \mathbf{i}] = -\sum_{\mathbf{j} \in \mathcal{S} \setminus \{\mathbf{i}\}} \mathbf{R}[\mathbf{i}, \mathbf{j}]$. We assume that the CTMC is *ergodic*, i.e., starting from any state in the CTMC, it is possible to reach (eventually) any other state in the CTMC [24]. This implies that $\boldsymbol{\pi}[\mathbf{i}]$ is non–zero for every reachable state $\mathbf{i} \in \mathcal{S}$.

We consider the important case where the model behavior is described by a set of *asynchronous* events $\mathcal{E}$. For each event $e \in \mathcal{E}$, $\mathcal{T}_e$ is a transition relation, i.e., the finite set of state pairs $(\mathbf{i}, \mathbf{i}') \in \mathcal{T}_e$ such that the model can move from state $\mathbf{i}$ to state $\mathbf{i}'$ when $e$ occurs. The overall transition relation is given by $\mathcal{T} = \bigcup_{e \in \mathcal{E}} \mathcal{T}_e$. We assume each event has a *deterministic* state–dependent rate, i.e., the rate function $f_e$ of event $e$ depends only on the *current* state, and is further broken into the product of $C$ *sub–functions*:

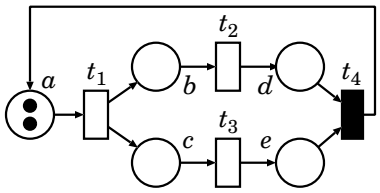
$$f_e \equiv \prod_{c=1}^{C} f_e^c, \tag{1}$$

where each $f_e^c$ depends on a set of state variables, namely variables in its *support*, denoted as $spt(f_e^c)$, and $spt(f_e) = \bigcup_{c=1}^{C} spt(f_e^c)$. The supports of these sub–functions do not have to be pairwise disjoint, and, in practice, the support of $f_e$ often contains just a few of the $L$ state variables. We stress that requiring a decomposition of $f_e$ into sub–functions only improves performance and is not restrictive in any way, since we can always use $C = 1$ if $f_e$ is cannot be easily decomposed into a product of functions.

Event $e$ may bring the model to a new state chosen from a set of states according to a probability function $\varphi_e$, which also depends on a set of state variables and can be similarly expressed as a product of sub–functions. The difference between $f$ and $\varphi$ is that $f$ depends only on the current state (or "from" state, denoted with unprimed symbols) while $\varphi$ also depends on the next state (or "to" state, denoted with primed symbols).

Then, the transition rate from state $\mathbf{i}$ to state $\mathbf{i}'$ due to event $e$ is

$$\mathbf{R}_e[\mathbf{i}, \mathbf{i}'] \equiv \begin{cases} f_e(\mathbf{i}) \cdot \varphi_e(\mathbf{i}, \mathbf{i}') & \text{if } (\mathbf{i}, \mathbf{i}') \in \mathcal{T}_e, \\ 0 & \text{otherwise,} \end{cases} \tag{2}$$




| **trans.** | **rate** |
|---|---|
| $t_1$ | $2.0 \cdot \#(a)$ |
| $t_2$ | $1.0 \cdot \#(b)$ |
| $t_3$ | $4.0 \cdot \#(c)$ |
| $t_4$ | immediate |

Fig. 2. Running example: the GSPN model of our fork–and–join system.

where, even if we write $f_e(\mathbf{i})$, only the values of variables in $spt(f_e)$ for state $\mathbf{i}$ are needed to evaluate $f_e$, similarly for $\varphi_e$. We stress that, for our encoding, we do not require $f_e(\mathbf{i})$ and $\varphi_e(\mathbf{i}, \mathbf{i}')$ to evaluate to 0 if $e$ is disabled in $\mathbf{i}$ or if it cannot lead from $\mathbf{i}$ to $\mathbf{i}'$; i.e., the values of $f_e(\mathbf{i})$ and $\varphi_e(\mathbf{i}, \mathbf{i}')$ are utilized only when $(\mathbf{i}, \mathbf{i}') \in \mathcal{T}_e$. Conversely, if $(\mathbf{i}, \mathbf{i}') \in \mathcal{T}_e$ but $f_e(\mathbf{i}) = 0$ or $\varphi_e(\mathbf{i}, \mathbf{i}') = 0$, state–space exploration could build a proper superset of the actual reachable states, i.e., some states in the obtained set $\mathcal{S}$ may in fact have a zero probability of being reached. Such cases have not arisen in our experience but, if desired, $\mathcal{T}_e$ could be modified to take into account this possibility. Finally, $\mathbf{R} = \sum_{e \in \mathcal{E}} \mathbf{R}_e$.

**Running example.** We use generalized stochastic Petri nets (GSPNs) [1], extended to allow variable–cardinality arcs, inhibitor arcs, and transition guards, as the high–level formalism to describe CTMCs. The GSPN of Fig. 2 models a simple fork–and–join queueing network with two customers (the two tokens initially in place $a$), and will be our running example. The black transition is immediate, and fires as soon as it is enabled. The white transitions have an exponentially distributed firing time, with the rates listed in the figure, where "$\#(z)$" means the number of tokens in place $z$. We consider two different decompositions for this model:

(1) $L = 3$, $\{x_3 \equiv [\#(a), \#(d), \#(e)], x_2 \equiv [\#(c)], x_1 \equiv [\#(b)]\}$.

(2) $L = 4$, $\{x_4 \equiv [\#(d), \#(e)], x_3 \equiv [\#(c)], x_2 \equiv [\#(b)], x_1 \equiv [\#(a)]\}$.

Here, $x_3 \equiv [\#(a), \#(d), \#(e)]$ means that each value of $x_3$ corresponds to a different combination for the number of tokens in $a$, $d$, and $e$.

The former is a Kronecker–consistent decomposition [7], the latter is not. This is because, in the second decomposition, the firing of $t_2$ adds a token to place $d$ (if place $e$ is empty) or place $a$ (if place $e$ contains tokens, so that transition $t_4$ fires once), but $a$ is described by $x_1$, while $d$ and $e$ are described by $x_4$.

Since the introduction of reduced ordered binary decision diagrams (BDDs) [4], further variations of decision diagrams have been proposed in the literature to compactly encode and efficiently compute functions on structured sets.

*Multiway decision diagrams* (MDDs) [16] allow integer–valued variables. Given $L$ variables $\mathbf{x} = (x_L, \ldots, x_1)$ with order $x_L \succ \cdots \succ x_1$, each $x_k$ taking value on a finite set $\mathcal{V}_k = \{0, 1, \ldots, n_k\} \subset \mathbb{N}$, a (quasi–reduced) MDD defined on $\mathbf{x}$ is a directed acyclic edge–labeled multi–graph where:

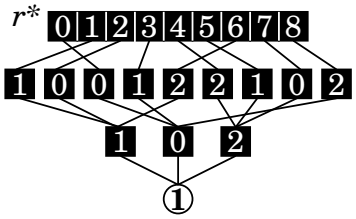
- Each nonterminal node $p$ is associated to a variable $p.var \in \{x_L, \ldots, x_1\}$, and we let $\mathcal{V}_p = \mathcal{V}_k$ if $p.var = x_k$.
- $\mathbf{0}$ and $\mathbf{1}$ are the only *terminal* nodes. Let $\mathbf{0}.var = \mathbf{1}.var = x_0$ and $x_k \succ x_0$, for $L \geq k \geq 1$.
- Each nonterminal node $p$ has $|\mathcal{V}_p|$ edges, labeled with a different index $i \in \mathcal{V}_p$ and pointing to a node $q$, written as $p[i] = q$. When $p.var = x_k$, we require that for every edge $p[i]$, either $p[i].var = x_{k-1}$ or $p[i] = \mathbf{0}$. Also, we require that $p[i] \neq \mathbf{0}$ for some $i$.
- *Duplicate* nodes are not allowed, i.e., given two distinct nonterminal nodes $p$ and $q$ with $p.var = q.var$, there must be an index $i \in \mathcal{V}_p$ such that $p[i] \neq q[i]$.
- There is a single root node $r^*$, with $r^*.var = x_L$ and no incoming edges, so that we can identify the MDD with its unique root.

Given a node $p$ with $p.var = x_k$, let $\mathcal{N}(p)$ be the set of nodes reachable from $p$, including $p$ itself, and, for $x_k \succeq x_h$, let $\mathcal{N}(p, x_h) = \{q \in \mathcal{N}(p) : q.var = x_h\}$ be the set of nodes associated to variable $x_h$ that are reachable from node $p$. We then extend the edge notation to paths, so that the node reached from $p$ through tuple $\alpha = (i_k, i_{k-1}, \ldots, i_h) \in \mathcal{V}_k \times \cdots \times \mathcal{V}_h$, for $L \geq k \geq h \geq 1$, is defined recursively as

$$p[\alpha] = \begin{cases} p[i_k][i_{k-1}, \ldots, i_h] \in \mathcal{N}(p, x_h) \cup \{\mathbf{0}\} & \text{if } p[i_k] \neq \mathbf{0}, \\ \mathbf{0} & \text{otherwise.} \end{cases}$$

Node $p$ associated to $x_k$ encodes the tuples $\mathcal{B}(p) = \{\beta \in \mathcal{V}_k \times \cdots \times \mathcal{V}_1 : p[\beta] = \mathbf{1}\}$, where $\mathcal{B}$ is mnemonic for "below". The MDDs just defined are canonical [10, 16]: given a set $\mathcal{X} \subseteq \mathcal{V}_L \times \cdots \times \mathcal{V}_1$, there exists a unique MDD $r^*_{\mathcal{X}}$ satisfying $\mathcal{X} = \mathcal{B}(r^*_{\mathcal{X}})$, where $r^*_{\mathcal{X}} = \mathbf{0}$ if $\mathcal{X} = \emptyset$, and $r^*_{\mathcal{X}}.var = x_L$ otherwise.

Fig. 3 shows the 3- and 4-variable MDDs encoding the state space for our fork–and–join model, under decomposition (1) or (2), respectively. Only paths leading to node $\mathbf{1}$ are shown. The meaning of the values for each $x_k$ is given on the right side. For example, in the upper MDD, the local states with index



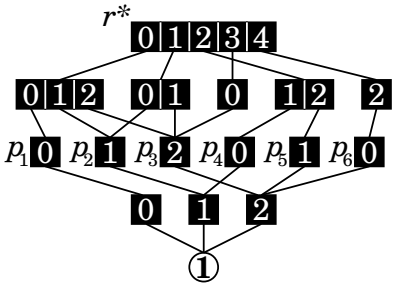

$\mathcal{V}_3$=$\{0{:}a^2, 1{:}a^1, 2{:}a^1e^1, 3{:}a^1d^1, 4{:}\lambda, 5{:}e^1, 6{:}d^1, 7{:}e^2, 8{:}d^2\}$

$\mathcal{V}_2$=$\{0{:}\lambda, 1{:}c^1, 2{:}c^2\}$

$\mathcal{V}_1$=$\{0{:}\lambda, 1{:}b^1, 2{:}b^2\}$

$$\mathcal{B}(r^*) = \{000,\ 111\ ,\ 201\ ,\ 310\ , 422,\ 512\ ,\ 621\ , 702, 820\}$$

$$\mathcal{S} = \{\ a^2\ , a^1c^1b^1, a^1e^1b^1, a^1d^1c^1, c^2b^2, e^1c^1b^2, d^1c^2b^1, e^2b^2, d^2c^2\}$$




$\mathcal{V}_4$=$\{0{:}\lambda, 1{:}e^1, 2{:}d^1, 3{:}e^2, 4{:}d^2\}$

$\mathcal{V}_3$=$\{0{:}\lambda, 1{:}c^1, 2{:}c^2\}$

$\mathcal{V}_2$=$\{0{:}\lambda, 1{:}b^1, 2{:}b^2\}$

$\mathcal{V}_1$=$\{0{:}a^2, 1{:}a^1, 2{:}\lambda\}$

$$\mathcal{B}(r^*) = \{0000,\ 0111\ , 0222,\ 1011\ ,\ 1122\ ,\ 2101\ ,\ 2212\ , 3022, 4202\}$$

$$\mathcal{S} = \{\ a^2\ \ , c^1b^1a^1,\ c^2b^2\ , e^1b^1a^1, e^1c^1b^2, d^1c^1a^1, d^1c^2b^1,\ e^2b^2\ ,\ d^2c^2\ \}$$

Fig. 3. Running example: the MDD encoding $\mathcal{S}$, under two decompositions.

0 and 4 in $\mathcal{V}_3$ are, respectively, $a^2$, meaning place $a$ contains two tokens and both place $d$ and $e$ are empty, and $\lambda$, meaning all three places are empty. Both MDDs encode set $\mathcal{S}$, which contains nine states, but the two encodings differ in the order in which the number of tokens in each place is listed when describing a state; this in turns affects the lexicographic order in which states are listed. Also, the first decomposition is essentially explicit, as the root node has nine outgoing edges, but this example is nevertheless useful for illustrative purposes.

### *2.3 Symbolic state–space generation*

We now briefly review the state–of–the–art on the symbolic state–space generation techniques to produce one of the two inputs of our algorithm, the MDD encoding the state space $\mathcal{S}$. We assume that the model is described using a high–level formalism, such as a GSPN (where the firing rates of timed transitions are ignored during state–space generation).

Let $Img(\mathcal{X}, \mathcal{Z}) = \{\mathbf{i}' : \exists \mathbf{i} \in \mathcal{X}, (\mathbf{i}, \mathbf{i}') \in \mathcal{Z}\}$ denote the *image* of the set of

states $\mathcal{X}$ under the transition relation $\mathcal{Z}$. Then, $\mathcal{S}$ is the minimal set satisfying $\mathcal{S} \supseteq \mathcal{S}_{init}$ and $\mathcal{S} \supseteq Img(\mathcal{S}, \mathcal{T})$, where $\mathcal{S}_{init}$ is the *initial* set of states. If $\mathcal{T}$ can be computed *a priori*, we can simply initialize $\mathcal{S}$ to $\mathcal{S}_{init}$ and repeatedly add to it $Img(\mathcal{S}, \mathcal{T})$, until we reach a fixpoint; otherwise, we need to generate $\mathcal{T}$ *on–the–fly* alongside $\mathcal{S}$ [9, 13, 26]. As $\mathcal{T} = \bigcup_{e \in \mathcal{E}} \mathcal{T}_e$, we can use the transition relations $\mathcal{T}_e$ instead of $\mathcal{T}$ for state–space generation, as long as each $\mathcal{T}_e$ is applied often enough to avoid "missing" any possible transition.

We already know that any set of states $\mathcal{X}$ can be encoded into an MDD $r^*_{\mathcal{X}}$ defined on $(x_L, \ldots, x_1)$ with the order $x_L \succ \cdots \succ x_1$. Similarly, any transition relation $\mathcal{Z}$ can be encoded into an MDD $r^*_{\mathcal{Z}}$ defined on $(x_L, x'_L, \ldots, x_1, x'_1)$ with the order $x_L \succ x'_L \succ \cdots \succ x_1 \succ x'_1$, where the unprimed variables refer to the "from" states and the primed variables refer to the "to" states, such that, if we let $\mathbf{i} = (i_L, \ldots, i_1)$ and $\mathbf{i}' = (i'_L, \ldots, i'_1)$, then $(\mathbf{i}, \mathbf{i}') \in \mathcal{Z} \Leftrightarrow (i_L, i'_L, \ldots, i_1, i'_1) \in \mathcal{B}(r^*_{\mathcal{Z}})$. We choose an *interleaved* order for the variables in the MDD for $\mathcal{Z}$ because it has been experimentally found to result in a compact encoding and allow an efficient computation [13, 26]; we nevertheless let $(\mathbf{i}, \mathbf{i}')$ denote the path tuple $(i_L, i'_L, \ldots, i_1, i'_1)$ and any (unprimed,primed) pair of tuples $(\alpha, \alpha')$, with $\alpha = (i_l, \ldots, i_h)$ and $\alpha' = (i'_l, \ldots, i'_h)$, denote the interleaving $(i_l, i'_l, \ldots, i_h, i'_h)$ of these two tuples.

The image computation $Img(\mathcal{X}, \mathcal{Z})$ corresponds to a *RelationalProduct* operation on MDD $r^*_{\mathcal{X}}$ and MDD $r^*_{\mathcal{Z}}$ [13, 26]. The simplest symbolic state–space generation uses a breadth–first strategy [21] and performs an image computation at each iteration, until reaching a global fixpoint. This method can be improved through chaining [22], to compound the effect of asynchronous events within a given breadth–first iteration. The saturation algorithm [8] uses instead a completely different iteration strategy to compute a fixpoint with respect to the events affecting each node in ascending variable order.

The saturation algorithm was first defined for transition relations that admit a Kronecker encoding. While this encoding always exists, it might be unwieldy, thus the approach was not fully general in practice. [19] avoids the Kronecker encoding by partitioning the transition relation of each event into groups, encoded by matrix diagrams [17], while [13] allows an even finer partition for the transition relations, where conjuncts can share state variables, which results in a more efficient computation. Our recent work [26] further improves on [13] by introducing an *extensible* version of MDDs to encode the transition relations while maximizing the reuse of nodes and caches. We believe this to be the most efficient symbolic state–space generation method to date for many realistic models exhibiting asynchronous behavior.

Among the variants of saturation mentioned above, [8] has restrictions on the decomposition of the model and is adopted by the approximate numerical solution in [20], while [13, 19, 26] allow an arbitrary decomposition of the

model. Empirically, a finer decomposition using more state variables can be much more efficient both time- and memory–wise than [8], which may be forced to use a coarser Kronecker decomposition.

In this paper, we adopt the state–space generation of [26]. A "side effect" of state–space generation, the on–the–fly generation of transition relations if they are not known or computed *a priori*, is needed in Sect. 3.3.

### *2.4 Decision–diagram–based aggregation*

An obvious way to tackle the state explosion problem is to *reduce* the number of states involved in the computation. An example of this idea is the basic aggregation we describe next.

**Basic aggregation.** We can partition $\mathcal{S}$ into $n$ sets of states $\{\mathcal{C}_1, \ldots, \mathcal{C}_n\}$. If we consider each set as a "macrostate", we can define an *aggregated* CTMC with state space $\mathcal{S}_{agg} = \{1, \ldots, n\}$ and transition rate matrix $\mathbf{R}_{agg}$ given by

$$\mathbf{R}_{agg}[i, i'] = \sum_{\mathbf{i} \in \mathcal{C}_i} \left( \boldsymbol{\pi}[\mathbf{i}|\mathcal{C}_i] \cdot \sum_{\mathbf{i}' \in \mathcal{C}_{i'}} \mathbf{R}[\mathbf{i}, \mathbf{i}'] \right), \tag{3}$$

where $\boldsymbol{\pi}[\mathbf{i}|\mathcal{C}_i]$ is the stationary conditional probability of being in state $\mathbf{i}$ given the state of CTMC is in set $\mathcal{C}_i$. It is well known that this aggregated CTMC is ergodic if the original one is ergodic, and that its stationary distribution $\boldsymbol{\pi}_{agg}$ satisfies $\boldsymbol{\pi}_{agg}[i] = \boldsymbol{\pi}[\mathcal{C}_i]$, where we let $\boldsymbol{\pi}[\mathcal{X}] = \sum_{\mathbf{i} \in \mathcal{X}} \boldsymbol{\pi}[\mathbf{i}]$ for $\mathcal{X} \subseteq \mathcal{S}$.

**MDD-based aggregation.** Following the partitioning strategy in [20], we define $L$ different aggregated CTMCs, one for each variable $x_k \in \{x_L, \ldots, x_1\}$, based on the reachable state paths from node $r^*_{\mathcal{S}}$ to node $\mathbf{1}$ in the MDD $r^*_{\mathcal{S}}$. A macrostate of an aggregated CTMC captures all the states that, in the MDD encoding, correspond to paths traversing a particular edge; i.e., for each $p \in \mathcal{N}(r^*_{\mathcal{S}}, x_k)$ and $i_k \in \mathcal{V}_k$, a macrostate of the aggregated CTMC for $x_k$, written as $[\![p, i_k]\!]$ corresponds to the set of states

$$[\![p, i_k]\!] \equiv \mathcal{A}(p) \times \{i_k\} \times \mathcal{B}(p[i_k]),$$

where $\mathcal{A}(p) \equiv \{\alpha : r^*_{\mathcal{S}}[\alpha] = p\}$ denotes all tuples leading from $r^*_{\mathcal{S}}$ to $p$ ($\mathcal{A}$ is mnemonic for "above"). Since $[\![p, i_k]\!] = \emptyset$ if and only if $p[i_k] = \mathbf{0}$, the state space of the aggregated CTMC for $x_k$ is $\{[\![p, i_k]\!] : p \in \mathcal{N}(r^*_{\mathcal{S}}, x_k), p[i_k] \neq \mathbf{0}\}$, and we use $\boldsymbol{\pi}_k$ and $\mathbf{R}_k$ to denote the steady–state probability vector and the transition rate matrix of this aggregated CTMC, respectively.

Considering the lower MDD in Fig. 3, $\mathcal{A}(p_2) = \{01, 10\}$ and $\mathcal{B}(p_2[1]) = \{1\}$, thus $[\![p_2, 1]\!] = \{01, 10\} \times \{1\} \times \{1\} = \{0111, 1011\}$. The state space of the

aggregated CTMC for $x_2$ is $\{[\![p_1, 0]\!], [\![p_2, 1]\!], [\![p_3, 2]\!], [\![p_4, 0]\!], [\![p_5, 1]\!], [\![p_6, 0]\!]\}$.

**Node and path probability.** We define the probability of an MDD node $p$ as $\boldsymbol{\pi}[p] = \boldsymbol{\pi}[\mathcal{A}(p) \times \mathcal{B}(p)]$, which can be viewed as the probability that a path of the MDD corresponding to a "random" state (chosen according to the stationary distribution $\boldsymbol{\pi}$) passes through node $p$, with special cases $\boldsymbol{\pi}[r_{\mathcal{S}}^*] = 1$, $\boldsymbol{\pi}[\mathbf{0}] = 0$, and $\boldsymbol{\pi}[\mathbf{1}] = 1$. Similarly, the probability of reaching node $p$ and then following edges according to the tuple $\gamma$ is $\boldsymbol{\pi}[p, \gamma] = \boldsymbol{\pi}[\mathcal{A}(p) \times \{\gamma\} \times \mathcal{B}(p[\gamma])]$, where $\boldsymbol{\pi}[p, i_k] = \boldsymbol{\pi}_k[[\![p, i_k]\!]]$ is a special case, and we write $\boldsymbol{\pi}[\gamma]$ instead of $\boldsymbol{\pi}[r_{\mathcal{S}}^*, \gamma]$. We can then condition on tuples or nodes using the classical definition of conditional probability, e.g., $\boldsymbol{\pi}[\gamma|p] = \boldsymbol{\pi}[p, \gamma]/\boldsymbol{\pi}[p]$. In particular, we have that, for a state $\mathbf{i} = (\alpha, i_k, \beta) \in [\![p, i_k]\!]$ where $\alpha \in \mathcal{A}(p)$ and $\beta \in \mathcal{B}(p[i_k])$,

$$\boldsymbol{\pi}[\mathbf{i}|p, i_k] = \boldsymbol{\pi}[\alpha, i_k, \beta|p, i_k] = \boldsymbol{\pi}[\alpha|p, i_k] \cdot \boldsymbol{\pi}[\beta|\alpha, p, i_k] = \boldsymbol{\pi}[\alpha|p, i_k] \cdot \boldsymbol{\pi}[\beta|\alpha, i_k], \quad (4)$$

since $\alpha$ uniquely determines $p$ (while the converse is not true in general).

## 3 Symbolic encoding of transition rate matrices

Kronecker–based encodings of transition rate matrices have been proposed to compute the *exact* or *approximate* stationary solution of ergodic CTMCs. These encodings are compact and effective but only applicable to models with a Kronecker decomposition, i.e., when we can express the transition rate matrix as the sum (over all events) of the Kronecker product (over all state variables) of *local* matrices [2, 5, 15]. While in principle this Kronecker form always exists, in practice it might be achieved only by splitting model events, resulting in an excessive number of events, or by merging state variables, resulting in excessively large local state spaces. For example, in Fig. 3, from decomposition (2) to decomposition (1), the largest local state space size changes from 5 to 9, and it is easy to see that, when there are $N$ initial tokens in place $a$, the largest local state space has size $2N + 1$ for decomposition (2) and $(N + 1)^2$ for decomposition (1). To overcome this restriction and make our approach completely general, we propose a new class of edge–valued decision diagrams as an alternative to Kronecker matrices, to retain the memory efficiency of the Kronecker approach while working with arbitrary model decompositions.

### *3.1 A new class of edge–valued decision diagrams*

We now introduce the multiplicative edge–valued multiway decision diagrams (EV*MDDs). Given $L$ variables $\mathbf{x} = (x_L, \ldots, x_1)$ with an order $x_L \succ \cdots \succ x_1$, each $x_k$ taking value in a finite set $\mathcal{V}_k = \{0, 1, \ldots, n_k\} \subset \mathbb{N}$, an EV*MDD

defined on $\mathbf{x}$ encodes a function of the form $f : \mathcal{V}_L \times \cdots \times \mathcal{V}_1 \to \mathbb{R}^{\geq 0}$ by associating values to the edges of an MDD. Formally, a (quasi–reduced) EV$^*$MDD defined on $\mathbf{x}$ is a directed acyclic edge–labeled multi–graph where:

- Each nonterminal node $p$ is associated to a variable $p.var \in \{x_L, \ldots, x_1\}$, and we let $\mathcal{V}_p = \mathcal{V}_k$ if $p.var = x_k$.
- $\Omega$ is the only *terminal* node. Let $\Omega.var = x_0$ and $x_k \succ x_0$, for $L \geq k \geq 1$.
- Each nonterminal node $p$ has $|\mathcal{V}_p|$ edges, each labeled with a different index $i \in \mathcal{V}_p$ and associated with a real value in $[0, 1]$. We write $p[i] = \langle \rho, q \rangle = \langle p[i].v, p[i].d \rangle$ if the edge labeled by $i$ points to node $q$ and is associated with value $\rho$. When $p.var = x_k$, we require that $q = \Omega$ if $\rho = 0$, $q.var = x_{k-1}$ otherwise. We also require that $\max_{i \in \mathcal{V}_p} p[i].v = 1$.
- There is a single root node $r^*$, with $r^*.var = x_L$ and an incoming *dangling* edge having associated value $\rho^* \in (0, +\infty)$, so that the EV$^*$MDD can be identified by $\langle \rho^*, r^* \rangle$.
- *Duplicate* nodes are not allowed: given two distinct nonterminal nodes $p$ and $q$ with $p.var = q.var$, there must be an index $i \in \mathcal{V}_p$ such that $p[i] \neq q[i]$.

Again, we can extend the edge notation to paths so that the node reached from $p$ through a tuple $\alpha = (i_k, i_{k-1}, \ldots, i_h) \in \mathcal{V}_k \times \cdots \times \mathcal{V}_h$, for $L \geq k \geq h \geq 1$, is defined recursively as

$$p[\alpha].d = \begin{cases} (p[i_k].d)[i_{k-1}, \ldots, i_h].d & \text{if } p[i_k].d \neq \Omega, \\ \Omega & \text{otherwise,} \end{cases}$$

and the value associated to this path is

$$p[\alpha].v = \begin{cases} p[i_k].v \cdot (p[i_k].d)[i_{k-1}, \ldots, i_h].v & \text{if } p[i_k].d \neq \Omega, \\ p[i_k].v & \text{otherwise.} \end{cases} \tag{5}$$

Thus, edge $\langle \rho, r \rangle$ with $r.var = x_k \succ x_0$ encodes function $f(\alpha) = \rho \cdot r[\alpha].v$, for $\alpha \in \mathcal{V}_k \times \cdots \times \mathcal{V}_1$, while edge $\langle \rho, \Omega \rangle$ encodes the constant $\rho$. Also, since $\max_{i \in \mathcal{V}_p} p[i].v = 1$ for each node $p$, we have that $\rho = \max(f)$. The next section proves that EV$^*$MDDs provide a canonical encoding for functions of the form $f : \mathcal{V}_L \times \cdots \times \mathcal{V}_1 \to \mathbb{R}^{\geq 0}$. We let $\langle \rho_f^*, r_f^* \rangle$ be the EV$^*$MDD encoding $f$, where the root $r_f^*$ is associated to $x_L$, unless $f$ is constant function identically equal to 0, for which we simply allow the special case $\langle \rho_f^*, r_f^* \rangle = \langle 0, \Omega \rangle$. EV$^*$MDDs can be considered both a generalization of an encoding based on Kronecker products (retaining its advantages while overcoming some of its disadvantages) and a multiplicative counterpart of EV$^+$MDDs [11] (thus exponentially more compact than non-edge-valued decision diagram representations such as MTBDDs [14], for some functions, e.g., $f(i_L, ..., i_1) = \prod_{L \geq k \geq 1} p_k^{i_k}$, where $p_k$ is the $k^{\text{th}}$ prime number).

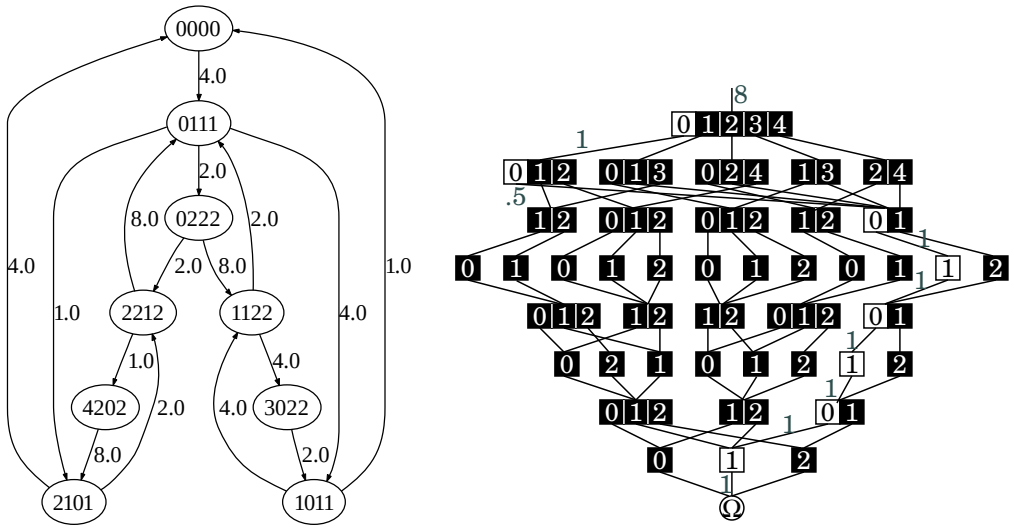



Fig. 4. Running example: exact CTMC and the EV*MDD encoding it.

The left of Fig. 4 shows the CTMC for decomposition (2) of our running example, while the right shows its EV*MDD encoding (only edges with non–zero value are displayed, and we omit all edge values except along one path, highlighted in white, corresponding to $\mathbf{R}[0000, 0111] = 4.0$). This EV*MDD contains some "fictitious" rates, as it in fact encodes a *potential* transition rate matrix, i.e., $\mathbf{R}[\mathbf{i}, \mathbf{i}'] > 0$ does not necessarily imply $\mathbf{i} \in \mathcal{S}$, more details about this can be found in Sect. 4.

Fig. 5 shows two representative operations for EV*MDDs, *Multiply* and *Add*, which take as input two EV*MDDs $\langle \rho_f^*, r_f^* \rangle$ and $\langle \rho_g^*, r_g^* \rangle$ and output $\langle \rho_{f \cdot g}^*, r_{f \cdot g}^* \rangle$ or $\langle \rho_{f+g}^*, r_{f+g}^* \rangle$, respectively. *Multiply* can be applied to two EV*MDDs defined on two different tuples of variables (as long as the overall set of variables has some predefined order, since the resulting EV*MDD is defined on a tuple containing all these variables). *Add* instead requires inputs defined on the same tuple of variables. Of course, functions defined on different tuple of variables can also be added, but we use this assumption to avoid details of reduction rules and missing variables for decision diagrams [26]; this framework is adequate for our needs. *UniqueTableInsert* ensures that there are no duplicate nodes and an operation cache makes both operations efficient, as always with decision–diagram manipulations.

### *3.2 Canonicity proof*

To prove that EV*MDDs are canonical representations, we must show that any non–negative function can be represented as an EV*MDD, and that this EV*MDD is unique.

The first part is proved by giving a construction of the EV*MDD encoding

```
evmdd Multiply(evmdd ⟨a,p⟩, evmdd ⟨b,q⟩)
 1  if a = 0 or b = 0 return ⟨0,Ω⟩;
 2  if p = Ω then return ⟨a · b,q⟩;
 3  if q = Ω then return ⟨a · b,p⟩;                          • base cases
 4  if q.var ≻ p.var then Swap(⟨a,p⟩, ⟨b,q⟩);                 • exploit commutativity
 5  if CacheHit(MULTIPLY, p, q, ⟨ρ,r⟩) then return ⟨a · b · ρ,r⟩;   • check cache
 6  r ← NewNode(p.var); μ ← 0;
 7  foreach i ∈ V_p do
 8    if p.var ≻ q.var then r[i] ← Multiply(p[i], ⟨b,q⟩);
 9    else r[i] ← Multiply(p[i], q[i]);                      • p.var = q.var
10    μ ← max{μ, r[i].v};
11  foreach i ∈ V_p s.t. r[i] ≠ ⟨0,Ω⟩ do r[i].v ← r[i].v/μ;     • normalize
12  r ← UniqueTableInsert(r);                                • insert into unique table
13  CacheAdd(MULTIPLY, p, q, ⟨μ,r⟩);                         • add to cache
14  return ⟨a · b · μ,r⟩;
```

```
evmdd Add(evmdd ⟨a,p⟩, evmdd ⟨b,q⟩)
 1  if a = 0 return ⟨b,q⟩;
 2  if b = 0 return ⟨a,p⟩;
 3  if p = q return ⟨a + b,p⟩;                               • base cases
 4  if a > b then Swap(⟨a,p⟩, ⟨b,q⟩);                        • exploit commutativity
 5  a ← a/b;
 6  if CacheHit(ADD, a, p, q, ⟨ρ,r⟩) then return ⟨b · ρ,r⟩;      • check cache
 7  r ← NewNode(p.var); μ ← 0;
 8  foreach i ∈ V_p do
 9    r[i] ← Add(⟨a · p[i].v,p[i].d⟩, q[i]);
10    μ ← max{μ, r[i].v};
11  foreach i ∈ V_p s.t. r[i] ≠ ⟨0,Ω⟩ do r[i].v ← r[i].v/μ;     • normalize
12  r ← UniqueTableInsert(r);                                • insert into unique table
13  CacheAdd(ADD, a, p, q, ⟨μ,r⟩);                           • add to cache
14  return ⟨b · μ,r⟩;
```

Fig. 5. Examples of EV*MDD operations.

an arbitrary non–negative function defined on a set of variables, as shown in Fig. 6. Procedure $Build(f, \mathbf{x})$ reads a function $f$ defined on $\mathbf{x}$ and outputs the canonical EV*MDD encoding it (*vtuple* and *ituple* are, respectively, sequences of variables or indices ordered consistently with "$\succ$"). It first builds an EV*MDD with duplicate nodes and arbitrary non–negative edge values, by enumerating non–zero values of $f$ and inserting them using procedure *AddEntry* (the result is a tree with edge values, not a quasi-reduced EV*MDD, but it nevertheless correctly encodes $f$ according to the interpretation of Eq. 5). Then, it calls procedure *Reduce* to produce a quasi-reduced EV*MDD described in Sect. 3.1. The *Reduce* procedure scales all edge values so that that the maximum value for each node is 1, and uses the unique table to ensure that there are no duplicate nodes, using a bottom–up strategy. If $\mathbf{x}$ is an

```
evmdd Build(expr f, vtuple x)
 1 if x = () then return ⟨f,Ω⟩;                                  • constant function
 2 ⟨ρ,r⟩ ← ⟨0,Ω⟩;
 3 foreach i ∈ V_{l_n} × ··· × V_{l_1} do                        • x = (x_{l_n}, ..., x_{l_1})
 4   if f(i) ≠ 0 then ⟨ρ,r⟩ ← AddEntry(⟨ρ,r⟩, i, f(i), x);
 5 return Reduce(⟨ρ,r⟩);
```

```
evmdd AddEntry(evmdd ⟨ρ,r⟩, ituple (i_{l_n}, ..., i_{l_1}), real σ, vtuple (x_{l_n}, ..., x_{l_1}))
 1 if ⟨ρ,r⟩ = ⟨0,Ω⟩ then ⟨ρ,r⟩ ← ⟨1,NewNode(x_{l_n})⟩;
 2 if x_{l_n} = x_{l_1} then r[i_{l_1}] ← ⟨σ,Ω⟩;                  • last variable in support
 3 else r[i_{l_n}] ← AddEntry(r[i_{l_n}], (i_{l_{n-1}}, ..., i_{l_1}), σ, (x_{l_{n-1}}, ..., x_{l_1}));
 4 return ⟨ρ,r⟩;
```

```
evmdd Reduce(evmdd ⟨ρ,r⟩)   • for nodes not yet checked into the unique table
 1 if r = Ω then return ⟨ρ,Ω⟩;
 2 if CacheHit(REDUCE, r, ⟨γ,p⟩) then return ⟨ρ · γ,p⟩;     • r already reduced
 3 μ ← 0;                                  • to compute maximum edge value in r
 4 foreach i ∈ V_r s.t. r[i] ≠ ⟨0,Ω⟩ do
 5   r[i] ← Reduce(r[i]);
 6   μ ← max{μ, r[i].v};
 7 foreach i ∈ V_r s.t. r[i] ≠ ⟨0,Ω⟩ do r[i].v ← r[i].v/μ;        • normalize
 8 p ← UniqueTableInsert(r);
 9 CachdAdd(REDUCE, r, ⟨μ,p⟩);
10 return ⟨ρ · μ,p⟩;
```

Fig. 6. Building the EV*MDD encoding $f$.

empty tuple, $f$ is a constant and *Build* simply returns $\langle f,\Omega\rangle$.

We now prove that this encoding is unique.

**Theorem 1 (Canonicity)** Given a tuple of variables $\mathbf{x} = (x_L, \ldots, x_1)$ and functions $f, g$ defined on $\mathbf{x}$, $\langle \rho_f^*, r_f^* \rangle = \langle \rho_g^*, r_g^* \rangle \iff f \equiv g$.

**Proof.** Obviously, if $\langle \rho_f^*, r_f^* \rangle = \langle \rho_g^*, r_g^* \rangle$, then they encode the same function, thus we only need to prove that, given function $f$, there cannot be two distinct $\langle \rho_f^*, r_f^* \rangle$ and $\langle \rho_g^*, r_g^* \rangle$ that encode it. Since $\langle \rho_f^*, r_f^* \rangle = \langle 0, \Omega \rangle$ is the only encoding of $f \equiv 0$ and $\langle \rho_f^*, r_f^* \rangle = \langle f, \Omega \rangle$ is the only encoding of $f$ when it is defined on an empty tuple $\mathbf{x} = ()$, we only need to consider the case $f \not\equiv 0$ and $\mathbf{x} \neq ()$.

As the definition of EV*MDD implies that $\rho_f^* = \rho_g^* = \max(f)$, we need to prove $r_f^* = r_g^*$, by induction. For the basis, when $f$ is defined on just one variable, i.e., $L = 1$ and $\mathbf{x} = (x_1)$, for any $i \in \mathcal{V}_1$, $f(i) = \rho_f^* \cdot r_f^*[i].v = \rho_g^* \cdot r_g^*[i].v$ implies $r_f^*[i].v = r_g^*[i].v$. Furthermore, of course, $r_f^*[i].d = r_g^*[i].d = \Omega$, thus $r_f^* = r_g^*$. For the inductive step, assume the theorem holds when $L = l$ for some $l \geq 1$ and show it holds when $L = l + 1$. For any tuple $\alpha = (i_{l+1}, i_l, \ldots, i_1) \equiv (i_{l+1}, \alpha') \in$

$\mathcal{V}_{l+1} \times \mathcal{V}_l \times \cdots \times \mathcal{V}_1$, let $f'$ be the function encoded by $r_f^*[i_{l+1}]$ and $g'$ be the function encoded by $r_g^*[i_{l+1}]$. Then,

$$f(\alpha) = \rho_f^* \cdot r_f^*[\alpha].v = \rho_f^* \cdot r_f^*[i_{l+1}, \alpha'].v = \rho_f^* \cdot r_f^*[i_{l+1}].v \cdot (r_f^*[i_{l+1}].d)[\alpha'].v = \rho_f^* \cdot f'(\alpha').$$

Similarly $f(\alpha) = \rho_g^* \cdot g'(\alpha')$. We already know $\rho_f^* = \rho_g^*$, so $f' \equiv g'$.

- If $f' \equiv 0$, we must have $r_f^*[i_{l+1}].v = 0$. Otherwise $(r_f^*[i_{l+1}].d).var = x_l \succ x_0$. Since each non–terminal node contains at least one edge valued 1, there exists at least one path $\beta \in \mathcal{V}_l \times \cdots \times \mathcal{V}_1$ such that $(r_f^*[i_{l+1}].d)[\beta].v = 1 \neq 0$, so $f'(\beta) = r_f^*[i_{l+1}].v > 0$; we reach a contradiction. So $r_f^*[i_{l+1}] = \langle 0, \Omega \rangle$. Similarly $r_g^*[i_{l+1}] = \langle 0, \Omega \rangle$, so $r_f^*[i_{l+1}] = r_g^*[i_{l+1}]$.
- If $f' \not\equiv 0$, since $f'$ is an $l$-variable function, the inductive assumption tells us that $f' \equiv g'$ implies $\langle \rho_{f'}^*, r_{f'}^* \rangle = \langle \rho_{g'}^*, r_{g'}^* \rangle$, thus $r_f^*[i_{l+1}] = r_g^*[i_{l+1}]$.

Since $i_{l+1}$ is arbitrarily chosen from $\mathcal{V}_{l+1}$, we can conclude that $r_f^* = r_g^*$ and the theorem is proved. □

### *3.3 Building the transition rate matrix*

We use an interleaved–order, $2L$-variable EV*MDD $\langle \rho_{\mathbf{R}}^*, r_{\mathbf{R}}^* \rangle$ to encode the transition rate matrix $\mathbf{R}$ such that

$$\forall \mathbf{i}, \mathbf{i}' \in \mathcal{S}, \mathbf{R}[\mathbf{i}, \mathbf{i}'] = \rho_{\mathbf{R}}^* \cdot r_{\mathbf{R}}^*[\mathbf{i}, \mathbf{i}'].v. \tag{6}$$

This is analogous to using $2L$-variable MDDs to encode transition relations during state–space generation. In fact, our algorithm to build $\langle \rho_{\mathbf{R}}^*, r_{\mathbf{R}}^* \rangle$ exploits this similarity, and utilizes the MDD encodings of the transition relations that are required for symbolic state–space generation. Our EV*MDD construction algorithm is shown in detail in Fig. 7, and is described below.

The main procedure, *BuildR*, calls procedure *BuildTransitionRate* to build the EV*MDD encoding of each matrix $\mathbf{R}_e$. These are summed over every event $e$ by iteratively calling procedure *Add* (shown in Fig. 5). The end result, EV*MDD $\langle \rho_{\mathbf{R}}^*, r_{\mathbf{R}}^* \rangle$, is used by our approximation algorithm in Sect. 4.

Procedure *BuildTransitionRate* builds the EV*MDD encoding $\mathbf{R}_e$ starting from the transition relation $\mathcal{T}_e$ produced during symbolic state–space generation. In principle, we can use any such algorithm that produces an $L$-variable MDD encoding for the state space $\mathcal{S}$ and an interleaved–order $2L$-variable MDD encoding relation $\mathcal{T}_e$ for each model event $e$; as already stated, we follow the method in [26]. In Fig. 7, $r_{\mathcal{T}_e}^*$ denotes the root node of the quasi–reduced MDD encoding transition relation $\mathcal{T}_e$, and it is assumed that this MDD has already been built. In practice, the state–space generation algorithm might use a slightly different representation for $\mathcal{T}_e$ (e.g., a different reduction rule or

```
BuildR()                                  • Build the EV*MDD encoding for R = Σ_{e∈E} R_e
 1 ⟨ρ*_R,r*_R⟩ ← ⟨0,Ω⟩;                   • initialize R to a matrix of zeros
 2 foreach e ∈ E do                       • add to R the contribution of event e
 3   ⟨ρ,r⟩ ← BuildTransitionRate(e);
 4   ⟨ρ*_R,r*_R⟩ ← Add(⟨ρ*_R,r*_R⟩, ⟨ρ,r⟩);
```

```
evmdd BuildTransitionRate(event e)        • Build the EV*MDD encoding for R_e
 1 ⟨ρ,r⟩ ← MakeEvmdd(1, r*_{T_e});
 2 ⟨a,p⟩ ← BuildFunction(f_e);
 3 ⟨a,p⟩ ← Multiply(⟨a,p⟩, BuildFunction(φ_e));
 4 return Multiply(⟨ρ,r⟩, ⟨a,p⟩);
```

```
evmdd BuildFunction(expr f)               • expressed as products of expressions
 1 ⟨ρ,r⟩ ← ⟨1,Ω⟩;
 2 exps ← GetProducts(f);                 • break into products
 3 foreach f^c in exps do
 4   ⟨a,p⟩ ← Build(f^c, spt(f^c));
 5   ⟨ρ,r⟩ ← Multiply(⟨ρ,r⟩, ⟨a,p⟩);
 6 return ⟨ρ,r⟩;
```

Fig. 7. Building EV*MDD encoding $\mathbf{R}$.

state variable indexing could be used to facilitate "on–the–fly" construction of $\mathcal{T}_e$, as in the case of extensible MDDs [26]). In this case, a simple pre-processing step [26] can transform the encoding of $\mathcal{T}_e$ into one satisfying the quasi–reduced MDD definition of Sect. 2.2.

From the MDD representation of $\mathcal{T}_e$, we build an EV*MDD representation of boolean matrix $\mathbf{R}_e^b$, where $\mathbf{R}_e^b(\mathbf{i}, \mathbf{i}') = 1$ iff $(\mathbf{i}, \mathbf{i}') \in \mathcal{T}_e$. This is achieved in Fig. 7 via the call to *MakeEvmdd*. Procedure *MakeEvmdd*, not shown, produces an EV*MDD from an MDD by copying nodes such that MDD edges pointing to a terminal node **0** or **1** become EV*MDD edges to $\Omega$ with an associated value 0 or 1, respectively, while MDD edges pointing to a non–terminal node become EV*MDD edges to the corresponding (EV*MDD) non–terminal node with an associated value 1. Finally, we add a dangling edge pointing to the EV*MDD root node, with an associated value 1.

Procedure *BuildTransitionRate* completes the construction of $\mathbf{R}_e$ according to Eq. 2, namely by exploiting the fact that $\mathbf{R}_e[\mathbf{i}, \mathbf{i}'] = f_e(\mathbf{i}) \cdot \varphi_e(\mathbf{i}, \mathbf{i}') \cdot \mathbf{R}_e^b[\mathbf{i}, \mathbf{i}']$ for any two states $\mathbf{i}, \mathbf{i}'$. The EV*MDDs for $f_e$ and $\varphi_e$ are built by calling procedure *BuildFunction* (described below) for the return expressions for functions $f_e$ and $\varphi_e$, respectively. Then, $\mathbf{R}_e$ is obtained as the element–wise product of $\mathbf{R}_e^b$, $f_e$, and $\varphi_e$, which is computed as the product of $\langle 1, r^*_{\mathbf{R}_e^b} \rangle$, $\langle \rho^*_{f_e}, r^*_{f_e} \rangle$ and $\langle \rho^*_{\varphi_e}, r^*_{\varphi_e} \rangle$ using procedure *Multiply* of Fig. 5.

Finally, given a function $f_e$ expressed as the product of sub-functions accord-

ing to Eq. 1, procedure *BuildFunction* in Fig. 6 builds the EV*MDD encoding $f_e$, by first calling $Build(f_e^c, spt(f_e^c))$ to build the EV*MDD for each term $f_e^c$. This requires enumerating the possible values of $(x_{s_1}, \ldots, x_{s_{l^{e,c}}}) = spt(f_e^c)$ and evaluating $f_e^c$ on them, but is usually computationally feasible, unlike the full enumeration of the possible values for the entire set of state variables $(x_L, \ldots, x_1)$. The finer decomposition allowed by our approach results in smaller local state spaces, thus tends to further reduce this computational cost. The overall encoding of $f_e$ is again computed using procedure *Multiply* (recall that the functions being multiplied do not need to have the same support).

### *3.4 Comparison with a Kronecker encoding*

When the model is Kronecker–consistent, $\mathbf{R}$ can be encoded either using a Kronecker representation [5] or an EV*MDD. The former is a *disjunctive* encoding where each $\mathbf{R}_e$ is encoded as the Kronecker product of local matrices. This encoding is compact because it only stores $|spt(\mathbf{R}_e)| = |spt(f_e) \cup spt(\varphi_e)|$ relatively small and usually very sparse matrices for event $e$. EV*MDDs also support this disjunctive encoding, and they encode each disjunct $\mathbf{R}_e$ into an $|spt(\mathbf{R}_e)|$-variable EV*MDD. Indeed, it is easy to see that the EV*MDD encoding $\mathbf{R}_e$ has exactly $|spt(\mathbf{R}_e)|$ "from" nodes, each followed by one or more "to" nodes (more precisely, as many as the distinct non–zero rows of the corresponding Kronecker matrix). In the worst case, the disjunctive EV*MDD encoding uses twice as many edges as there are non–zeros in the Kronecker matrices for the same model decomposition, because each non–zero element in a Kronecker matrix corresponds to two edges in an EV*MDD (one from a "from" node followed by one from a "to" node). On the other hand, EV*MDDs allow for node sharing, which can greatly reduce the number of edges and is the main motivation to use decision diagrams instead of decision trees.

More fundamentally, though, the EV*MDD representation allows us to use a model decomposition that is not Kronecker–consistent. In practice, the benefit of this generalization is that it allows for "finer" decompositions consisting of more state variables, each with fewer possible values.

We emphasize that our approximation algorithm can use either a single–root EV*MDD for $\mathbf{R}$, or a disjunctive encoding for $\mathbf{R}$. In this paper, we use a single–root EV*MDD to encode $\mathbf{R}$ for illustration purposes and also because this choice may avoid many additions of rates during the computation. However, the algorithm in the next section needs only a small change to use a disjunctive encoding, specifically, traversing down from the root of each EV*MDD for each event, instead of from the only root. The ability to handle a disjunctive encoding is important, as this encoding can require many fewer nodes than the single–root MDD encoding: in the worst case, a single–root EV*MDD

generated after summing $n$ EV*MDDs can be exponentially larger (in $n$) than the original EV*MDDs being summed, just as the union of $n$ MDDs can grow exponentially in $n$.

To illustrate the differences in these encodings, Table 1 shows a comparison of the size of the Kronecker and the EV*MDD encodings mentioned above for our running example fork–join model. In the table, $N$ refers to the initial number of tokens in place $a$. For the Kronecker–consistent decomposition ($L = 3$), the number of non–zero matrix elements in the Kronecker representation (column "nz") is $3N^2+6N$, the number of non–zero edges in the disjunctive EV*MDD encoding is $6N^2+11N+1$, nearly double, and the number of non–zero edges in the single–root EV*MDD encoding is $4N^2 + 14N + 5$. However, for the finer, not Kronecker–consistent, decomposition ($L = 4$), the number of non–zero edges in the disjunctive EV*MDD encoding is $24N + 2$ and the number of non–zero edges in the single–root EV*MDD encoding is $30N + 12$. Of course, in this case, the entries for Kronecker are "n/a" (not applicable). Thus, the memory requirements to store $\mathbf{R}$ are linear in $N$ under the finer decomposition, with the disjunctive encoding being more efficient, but are quadratic in $N$ under the Kronecker–consistent decomposition, with the single–root encoding being more efficient than the disjunctive encoding. The space requirements discussed so far refer to the encoding of the "unfiltered" matrix $\mathbf{R}$: some of the encoded rates are fictitious, i.e., do not correspond to legal transitions and must be filtered out using the transition relation $\mathcal{T}$ (specifically, we must set $\mathbf{R}(\mathbf{i}, \mathbf{i}')$ to zero if $(\mathbf{i}, \mathbf{i}') \notin \mathcal{T}$, as required by Eq. 2). Unlike the Kronecker approach, however, the EV*MDD encoding allows us to eliminate these fictitious transitions in the encoding itself. Column "Actual $\mathbf{R}$" in Table 1 shows the number of nodes and edges of the resulting EV*MDD. As it often happens, the memory requirements increase: the number of edges is $10N^2+12N+3$ for the Kronecker–consistent decomposition and $9N^2 + 22N + 1$ for the finer decomposition, confirming that, in practice, it is usually preferable to filter matrix $\mathbf{R}$ while using it, rather than computing the filtered version of $\mathbf{R}$ explicitly.

## 4 Our approximation algorithm

After building the MDD encoding the state space $\mathcal{S}$ and the EV*MDD encoding the transition rate matrix $\mathbf{R}$ for the CTMC under study, we are ready to carry out our approximation algorithm. First, we examine exact aggregation using EV*MDDs.

| | | | **Kronecker** | | **EV*MDD** | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | Disjunctive | | Single root | | Actual **R** | |
| $N$ | $\eta(\mathbf{R})$ | $L$ | matrices | nz | nodes | edges | nodes | edges | nodes | edges |
| 1 | 5 | 3 | 10 | 9 | 16 | 18 | 17 | 23 | 21 | 25 |
| | | 4 | n/a | n/a | 23 | 26 | 31 | 42 | 28 | 32 |
| 2 | 16 | 3 | 10 | 24 | 30 | 47 | 26 | 49 | 52 | 67 |
| | | 4 | n/a | n/a | 34 | 50 | 42 | 72 | 66 | 81 |
| 3 | 33 | 3 | 10 | 45 | 50 | 88 | 37 | 83 | 97 | 129 |
| | | 4 | n/a | n/a | 45 | 74 | 53 | 102 | 116 | 148 |
| 4 | 56 | 3 | 10 | 72 | 76 | 141 | 50 | 125 | 156 | 211 |
| | | 4 | n/a | n/a | 56 | 98 | 64 | 132 | 178 | 233 |
| 5 | 85 | 3 | 10 | 105 | 108 | 206 | 65 | 175 | 229 | 313 |
| | | 4 | n/a | n/a | 67 | 122 | 75 | 162 | 252 | 336 |
| 6 | 120 | 3 | 10 | 144 | 146 | 283 | 82 | 233 | 316 | 435 |
| | | 4 | n/a | n/a | 78 | 146 | 86 | 192 | 338 | 457 |
| 7 | 161 | 3 | 10 | 189 | 190 | 372 | 101 | 299 | 417 | 577 |
| | | 4 | n/a | n/a | 89 | 170 | 97 | 222 | 436 | 596 |
| 8 | 208 | 3 | 10 | 240 | 240 | 473 | 122 | 373 | 532 | 739 |
| | | 4 | n/a | n/a | 100 | 194 | 108 | 252 | 546 | 753 |
| 9 | 261 | 3 | 10 | 297 | 296 | 586 | 145 | 455 | 661 | 921 |
| | | 4 | n/a | n/a | 111 | 218 | 119 | 282 | 668 | 928 |
| 10 | 320 | 3 | 10 | 360 | 358 | 711 | 170 | 545 | 804 | 1123 |
| | | 4 | n/a | n/a | 122 | 242 | 130 | 312 | 802 | 1121 |

Table 1
Running example: Data structure sizes for Kronecker or EV*MDD encodings of **R**.

### *4.1 Exact aggregation*

For clarity, we recall some equations obtained in the previous sections, namely Eq. 3 (basic aggregation):

$$\mathbf{R}_{agg}[i, i'] = \sum_{\mathbf{i} \in \mathcal{C}_i} \boldsymbol{\pi}[\mathbf{i}|\mathcal{C}_i] \sum_{\mathbf{i}' \in \mathcal{C}_{i'}} \mathbf{R}[\mathbf{i}, \mathbf{i}'],$$

Eq. 4 (node and path conditional probability), letting $\mathbf{i} = (\alpha, i_k, \beta)$, $p = r^*_{\mathcal{S}}[\alpha]$:

$$\boldsymbol{\pi}[\mathbf{i}|p, i_k] = \boldsymbol{\pi}[\alpha|p, i_k]\boldsymbol{\pi}[\beta|\alpha, i_k],$$

and Eq. 6 (EV*MDD encoding **R**):

$$\forall \mathbf{i}, \mathbf{i}' \in \mathcal{S}, \mathbf{R}[\mathbf{i}, \mathbf{i}'] = \rho^*_{\mathbf{R}} r^*_{\mathbf{R}}[\mathbf{i}, \mathbf{i}'].v.$$

Combining these equations, and utilizing Eq. 5, we compute the rate from macrostate $[\![p, i_k]\!]$ to macrostate $[\![q, i'_k]\!]$ in $\mathbf{R}_k$ as

$$\begin{aligned}
&\mathbf{R}_k[[\![p, i_k]\!], [\![q, i'_k]\!]] \\
&\quad = \sum_{\mathbf{i} \in [\![p, i_k]\!]} \boldsymbol{\pi}[\mathbf{i}|[\![p, i_k]\!]] \sum_{\mathbf{i}' \in [\![q, i'_k]\!]} \rho^*_{\mathbf{R}} r^*_{\mathbf{R}}[\mathbf{i}, \mathbf{i}'].v \\
&\quad = \sum_{\substack{\alpha \in \mathcal{A}(p) \\ \beta \in \mathcal{B}(p[i_k])}} \boldsymbol{\pi}[\alpha|p, i_k]\boldsymbol{\pi}[\beta|\alpha, i_k] \sum_{\substack{\alpha' \in \mathcal{A}(q) \\ \beta' \in \mathcal{B}(q[i'_k])}} \rho^*_{\mathbf{R}} r^*_{\mathbf{R}}[\alpha, \alpha'].v \cdot r[i_k, i'_k].v \cdot r'[\beta, \beta'].v \\
&\quad = \rho^*_{\mathbf{R}} \sum_{\alpha \in \mathcal{A}(p)} \boldsymbol{\pi}[\alpha|p, i_k] \sum_{\alpha' \in \mathcal{A}(q)} r^*_{\mathbf{R}}[\alpha, \alpha'].v \cdot r[i_k, i'_k].v \\
&\qquad \sum_{\beta \in \mathcal{B}(p[i_k])} \boldsymbol{\pi}[\beta|\alpha, i_k] \sum_{\beta' \in \mathcal{B}(q[i'_k])} r'[\beta, \beta'].v, \qquad (7)
\end{aligned}$$

where $r = r^*_{\mathbf{R}}[\alpha, \alpha'].d$ and $r' = r[i_k, i'_k].d$. This exact aggregation equation unfortunately cannot be applied in practice to reduce the size of the CTMC under study, because, in general, the quantities $\boldsymbol{\pi}[\alpha|p, i_k]$ and $\boldsymbol{\pi}[\beta|\alpha, i_k]$ cannot be determined without first computing $\boldsymbol{\pi}[\mathbf{i}]$ for all states $\mathbf{i}$ in the original CTMC, which is the computation we are hoping to avoid. We then propose an *approximate* aggregation that uses *estimates* for $\boldsymbol{\pi}[\alpha|p, i_k]$ and $\boldsymbol{\pi}[\beta|\alpha, i_k]$.

### *4.2 Approximate aggregation*

The motivation for our approximation comes from the observation that each complete path through $r^*_{\mathcal{S}}$ corresponds to a reachable state. Now, consider a random walk through $r^*_{\mathcal{S}}$, such that the probability of a path equals the stationary probability of the state described by that path. Along the random walk, say at node $p$, the decision probabilities for the outgoing edges of node $p$ depend in general on the entire path traversed to reach node $p$. To simplify the equations, we therefore *assume* a "memoryless" property, i.e., we assume that these decision probabilities are independent of the path used to reach node $p$. Formally, we assume that

$$\forall p \in \mathcal{N}(r^*_{\mathcal{S}}, x_k),\ \forall \alpha \in \mathcal{A}(p),\ \boldsymbol{\pi}[i_k|\alpha] \ = \ \boldsymbol{\pi}[i_k|p], \qquad (8)$$

where partial state probabilities such as $\boldsymbol{\pi}[i_k]$ are obtained by summing $\boldsymbol{\pi}[\mathbf{i}]$ over appropriate sets of states. This assumption and its derivations let us simplify Eq. 7 into a form that allows a fixpoint computation to determine the aggregate probabilities. This approximation idea is also adopted in [20] and has an important justification: if the estimated probability is 0, then the actual probability is also 0, and vice versa.

Before proceeding with the simplification of Eq. 7, we derive some properties necessary for this work. Because these properties are *equivalent* to Eq. 8, our underlying assumption cannot be relaxed without changing our approximation technique.

**Theorem 2** Given $p \in \mathcal{N}(r^*_{\mathcal{S}}, x_k)$, the following properties are equivalent:

(P1) $\boldsymbol{\pi}[i_k|\alpha] = \boldsymbol{\pi}[i_k|p], \quad \forall \alpha \in \mathcal{A}(p).$

(P2) $\boldsymbol{\pi}[i_h|p,\gamma] = \boldsymbol{\pi}[i_h|p[\gamma]], \quad \forall (\gamma, i_h, \gamma') \in \mathcal{B}(p).$

(P3) $\boldsymbol{\pi}[\alpha|p,\gamma] = \boldsymbol{\pi}[\alpha|p], \quad \forall \alpha \in \mathcal{A}(p), \forall (\gamma,\gamma') \in \mathcal{B}(p).$

(P4) $\boldsymbol{\pi}[\gamma|\alpha] = \boldsymbol{\pi}[\gamma|p], \quad \forall \alpha \in \mathcal{A}(p), \forall (\gamma,\gamma') \in \mathcal{B}(p).$

(P5) $\boldsymbol{\pi}[\gamma'|p,\gamma] = \boldsymbol{\pi}[\gamma'|p[\gamma]], \quad \forall (\gamma,\gamma',\gamma'') \in \mathcal{B}(p).$

**Proof.** (P1) implies (P2):

$$\begin{aligned}
\boldsymbol{\pi}[i_h|p,\gamma] &= \frac{\boldsymbol{\pi}[p,\gamma,i_h]}{\boldsymbol{\pi}[p,\gamma]} = \frac{\sum_{\alpha\in\mathcal{A}(p)} \boldsymbol{\pi}[\alpha,\gamma,i_h]}{\boldsymbol{\pi}[p,\gamma]} = \frac{\sum_{\alpha\in\mathcal{A}(p)} \boldsymbol{\pi}[i_h|\alpha,\gamma]\cdot\boldsymbol{\pi}[\alpha,\gamma]}{\boldsymbol{\pi}[p,\gamma]} \\
&= \frac{\sum_{\alpha\in\mathcal{A}(p)} \boldsymbol{\pi}[i_h|p[\gamma]]\cdot\boldsymbol{\pi}[\alpha,\gamma]}{\boldsymbol{\pi}[p,\gamma]} = \boldsymbol{\pi}[i_h|p[\gamma]]\cdot\frac{\sum_{\alpha\in\mathcal{A}(p)} \boldsymbol{\pi}[\alpha,\gamma]}{\boldsymbol{\pi}[p,\gamma]} \\
&= \boldsymbol{\pi}[i_h|p[\gamma]],
\end{aligned}$$

where the critical step uses (P1) and the fact that $(\alpha,\gamma) \in \mathcal{A}(p[\gamma])$.

(P2) implies (P3): letting $\gamma = (i_k,\ldots,i_h)$, we have

$$\begin{aligned}
\boldsymbol{\pi}[\alpha|p,\gamma] &= \frac{\boldsymbol{\pi}[\alpha,p,i_k,\ldots,i_h]}{\boldsymbol{\pi}[p,i_k,\ldots,i_h]} \\
&= \frac{\boldsymbol{\pi}[i_h|\alpha,p,i_k,\ldots,i_{h+1}]\cdot\boldsymbol{\pi}[\alpha,p,i_k,\ldots,i_{h+1}]}{\boldsymbol{\pi}[i_h|p,i_k,\ldots,i_{h+1}]\cdot\boldsymbol{\pi}[p,i_k,\ldots,i_{h+1}]} \\
&= \frac{\boldsymbol{\pi}[i_h|r^*_{\mathcal{S}},\alpha,i_k,\ldots,i_{h+1}]\cdot\boldsymbol{\pi}[\alpha,p,i_k,\ldots,i_{h+1}]}{\boldsymbol{\pi}[i_h|p,i_k,\ldots,i_{h+1}]\cdot\boldsymbol{\pi}[p,i_k,\ldots,i_{h+1}]} \\
&= \frac{\boldsymbol{\pi}[i_h|p[i_k,\ldots,i_{h+1}]]\cdot\boldsymbol{\pi}[\alpha,p,i_k,\ldots,i_{h+1}]}{\boldsymbol{\pi}[i_h|p[i_k,\ldots,i_{h+1}]]\cdot\boldsymbol{\pi}[p,i_k,\ldots,i_{h+1}]} \\
&= \frac{\boldsymbol{\pi}[\alpha,p,i_k,\ldots,i_{h+1}]}{\boldsymbol{\pi}[p,i_k,\ldots,i_{h+1}]},
\end{aligned}$$

where the first step is due to the fact that $\alpha$ determines $p$ and all paths include the root node $r^*_{\mathcal{S}}$. The critical step is based on (P2) and the fact that $r^*_{\mathcal{S}}[\alpha,i_k,\ldots,i_{h+1}] = p[i_k,\ldots,i_{h+1}]$. Repeating this argument, we can eliminate

terms $i_{h+1}$ through $i_k$, which gives us

$$\boldsymbol{\pi}[\alpha|p,\gamma] \;=\; \frac{\boldsymbol{\pi}[\alpha,p]}{\boldsymbol{\pi}[p]} \;=\; \boldsymbol{\pi}[\alpha|p].$$

(P3) implies (P4):

$$\boldsymbol{\pi}[\gamma|\alpha] \;=\; \boldsymbol{\pi}[\gamma|\alpha,p] \;=\; \frac{\boldsymbol{\pi}[\alpha,p,\gamma]}{\boldsymbol{\pi}[\alpha,p]} \;=\; \frac{\boldsymbol{\pi}[\alpha|p,\gamma]}{\boldsymbol{\pi}[\alpha|p]} \cdot \frac{\boldsymbol{\pi}[p,\gamma]}{\boldsymbol{\pi}[p]} \;=\; \frac{\boldsymbol{\pi}[p,\gamma]}{\boldsymbol{\pi}[p]} \;=\; \boldsymbol{\pi}[\gamma|p].$$

(P4) implies (P5):

$$\begin{aligned}
\boldsymbol{\pi}[\gamma'|p,\gamma] \;&=\; \frac{\boldsymbol{\pi}[p,\gamma,\gamma']}{\boldsymbol{\pi}[p,\gamma]} \;=\; \frac{\sum_{\alpha\in\mathcal{A}(p)} \boldsymbol{\pi}[\alpha,\gamma,\gamma']}{\boldsymbol{\pi}[p,\gamma]} \\
&=\; \frac{\sum_{\alpha\in\mathcal{A}(p)} \boldsymbol{\pi}[\gamma'|\alpha,\gamma]\cdot\boldsymbol{\pi}[\alpha,\gamma]}{\boldsymbol{\pi}[p,\gamma]} \;=\; \frac{\sum_{\alpha\in\mathcal{A}(p)} \boldsymbol{\pi}[\gamma'|p[\gamma]]\cdot\boldsymbol{\pi}[\alpha,\gamma]}{\boldsymbol{\pi}[p,\gamma]} \\
&=\; \boldsymbol{\pi}[\gamma'|p[\gamma]] \cdot \frac{\sum_{\alpha\in\mathcal{A}(p)} \boldsymbol{\pi}[\alpha,\gamma]}{\boldsymbol{\pi}[p,\gamma]} \;=\; \boldsymbol{\pi}[\gamma'|p[\gamma]],
\end{aligned}$$

where the critical step is due to (P4) and the fact that $(\alpha,\gamma) \in \mathcal{A}(p[\gamma])$.

Finally, (P5) implies (P1), trivially proved using $\gamma' = i_k$, $p = r^*_{\mathcal{S}}$, and $\gamma = \alpha$. □

Properties (P1) and (P2) are similar: (P2) says that the probability of a state containing $i_h$, given that its path through the MDD reaches node $p$ and then follows edges $\gamma$, equals the probability of a state containing $i_h$, given that its path reaches node $p[\gamma]$. Note that (P1) is the special case of (P2) when $p = r^*_{\mathcal{S}}$. In an MDD, there could be several paths that lead to node $p$; (P3) says that the probability of a particular such path $\alpha$ does not depend on the path chosen after node $p$ is reached. Finally, properties (P4) and (P5) are the "path analogs" of properties (P1) and (P2).

We now turn our attention to how (P1) affects the structure of the probability vector $\boldsymbol{\pi}$.

**Theorem 3** The following properties are equivalent:

(P1) Given $p \in \mathcal{N}(r_{\mathcal{S}}^*, x_k)$, $\forall \alpha \in \mathcal{A}(p), \boldsymbol{\pi}[i_k|\alpha] = \boldsymbol{\pi}[i_k|p]$.

(P6) $\forall (i_L, \ldots, i_1) \in \mathcal{S}, \boldsymbol{\pi}[i_L, \ldots, i_1] = \boldsymbol{\pi}[i_L|p_L] \cdots \boldsymbol{\pi}[i_1|p_1]$, where $p_L = r_{\mathcal{S}}^*$ and $p_{k-1} = p_k[i_k], \forall k \in \{2, \ldots, L\}$.

(P7) There exists a function $f_{\boldsymbol{\pi}} : \mathbb{R}^{\geq 0} \times \mathcal{N}(r_{\mathcal{S}}^*) \rightarrow$ EV*MDD such that EV*MDD $f_{\boldsymbol{\pi}}(1, r_{\mathcal{S}}^*)$ encodes $\boldsymbol{\pi}$ in its "reachable portion", and encodes $\mathbf{0}$ in its "unreachable portion", where $f_{\boldsymbol{\pi}}$ has the properties $(i)$ $f_{\boldsymbol{\pi}}(1, p) = \langle \rho, q \rangle \implies f_{\boldsymbol{\pi}}(\sigma, p) = \langle \sigma\rho, q \rangle$ and $(ii)$ if $p$ is a non-terminal node, then $f_{\boldsymbol{\pi}}(1, p) = \langle \rho, q \rangle \implies q[i].d = f_{\boldsymbol{\pi}}(1, p[i]).d$

(P8) Given $p \in \mathcal{N}(r_{\mathcal{S}}^*, x_k)$, $\forall \alpha, \alpha' \in \mathcal{A}(p), \boldsymbol{\pi}[i_k|\alpha] = \boldsymbol{\pi}[i_k|\alpha']$

**Proof.** (P1) implies (P6):

$$\begin{aligned} \boldsymbol{\pi}[i_L, \ldots, i_1] &= \boldsymbol{\pi}[i_1|i_L, \ldots, i_2] \cdot \boldsymbol{\pi}[i_L, \ldots, i_2] \\ &= \boldsymbol{\pi}[i_1|i_L, \ldots, i_2] \cdot \boldsymbol{\pi}[i_2|i_L, \ldots, i_3] \cdot \cdots \cdot \boldsymbol{\pi}[i_L] \\ &= \boldsymbol{\pi}[i_1|p_1] \cdot \boldsymbol{\pi}[i_2|p_2] \cdot \cdots \cdot \boldsymbol{\pi}[i_L|p_L] \end{aligned}$$

where the critical step uses (P1) and the fact that $(i_L, \ldots, i_k) \in \mathcal{A}(p[k-1])$.

(P6) implies (P7): We first give a function $f'$ that satisfies the requirements for $f_{\boldsymbol{\pi}}$, except it produces an EV*MDD that is not reduced:

$$f'(\sigma, p) = \begin{cases} \langle \sigma, \Omega \rangle & \text{if } p.var = x_0 \\ \langle \sigma, q \rangle & \text{otherwise, with } q[i] = f'(\boldsymbol{\pi}[i|p], p[i]) \end{cases}$$

Now we show that $f'(1, r_{\mathcal{S}}^*)$ encodes $\boldsymbol{\pi}$ in its "reachable portion", and encodes $\mathbf{0}$ in its "unreachable portion". Let $\varpi$ be the function encoded by $f'(1, r_{\mathcal{S}}^*)$. By construction, if $(i_L, \ldots, i_1) \in \mathcal{S}$, then we have

$$\varpi(i_L, \ldots, i_1) = 1 \cdot \boldsymbol{\pi}[i_L|r_{\mathcal{S}}^*] \cdot \boldsymbol{\pi}[i_{L-1}|p_{L-1}] \cdot \cdots \cdot \boldsymbol{\pi}[i_1|p_1]$$

which is equal to $\boldsymbol{\pi}[i_L, \ldots, i_1]$ by (P6). If $(i_L, \ldots, i_1) \notin \mathcal{S}$, then for some $k$, $p_k[i_k] = \mathbf{0}$ and $\boldsymbol{\pi}[i_k|p_k] = 0$; $f'(1, p_k)$ will then give a node $q_k$ with $q_k[i_k] = \langle 0, \Omega \rangle$ and we have $\varpi(i_L, \ldots, i_1) = 0$. Finally, we define $f_{\boldsymbol{\pi}}(\sigma, p) = \mathit{Reduce}(f'(\sigma, p))$, where *Reduce* is given in Fig. 6.

(P7) implies (P8) : Given $(i_L, \ldots, i_1) \in \mathcal{S}$, let $p_L, \ldots, p_0$ and $\langle \sigma_L, q_L \rangle, \ldots, \langle \sigma_0, q_0 \rangle$ be, respectively, the sequence of MDD nodes starting from $r_{\mathcal{S}}^*$ and of EV*MDD edges starting from $f_{\boldsymbol{\pi}}(1, r_{\mathcal{S}}^*)$ that are traversed according to $(i_L, \ldots, i_1)$; thus, $p_L = r_{\mathcal{S}}^*$ and $\langle \sigma_L, q_L \rangle = f_{\boldsymbol{\pi}}(1, r_{\mathcal{S}}^*)$ and, for $L \geq k > 0$, $p_{k-1} = p_k[i_k]$ and $\langle \sigma_{k-1}, q_{k-1} \rangle = q_k[i_k]$. Now, let $\alpha = (i_L, \ldots, i_{k+1})$ and $\alpha' = (i'_L, \ldots, i'_{k+1})$, with $\alpha, \alpha' \in \mathcal{A}(p_k)$, and let $\langle \sigma'_L, q'_L \rangle, \ldots, \langle \sigma'_0, q'_0 \rangle$ be the sequence of EV*MDD edges

starting from $f_{\boldsymbol{\pi}}(1, r_S^*)$ that are traversed according to $(i'_L, \ldots, i'_{k+1}, i_k, \ldots, i_1)$. Since $\alpha' \in \mathcal{A}(p_k)$ and (P7) implies that $q_k$ is uniquely determined by $p_k$, we have that $q'_l = q_l$ for all $k \geq l \geq 0$, therefore $\sigma'_l = \sigma_l$ for all $k > l \geq 0$, and

$$\begin{aligned}
\boldsymbol{\pi}[i_k|\alpha] &= \frac{\boldsymbol{\pi}[\alpha, i_k]}{\boldsymbol{\pi}[\alpha]} = \frac{\sum_{(i_{k-1},\ldots,i_1)\in\mathcal{B}(p_{k-1})} \boldsymbol{\pi}[i_L, \ldots, i_1]}{\sum_{(i_k,\ldots,i_1)\in\mathcal{B}(p_k)} \boldsymbol{\pi}[i_L, \ldots, i_1]} \\
&= \frac{\prod_{j=k}^{L} \sigma_j}{\prod_{j=k}^{L} \sigma_j} \cdot \frac{\sigma_{k-1} \sum_{(i_{k-1},\ldots,i_1)\in\mathcal{B}(p_{k-1})} \prod_{j=0}^{k-2} \sigma_j}{\sum_{(i_k,\ldots,i_1)\in\mathcal{B}(p_k)} \prod_{j=0}^{k-1} \sigma_j} \\
&= \frac{\prod_{j=k}^{L} \sigma'_j}{\prod_{j=k}^{L} \sigma'_j} \cdot \frac{\sigma_{k-1} \sum_{(i_{k-1},\ldots,i_1)\in\mathcal{B}(p_{k-1})} \prod_{j=0}^{k-2} \sigma_j}{\sum_{(i_k,\ldots,i_1)\in\mathcal{B}(p_k)} \prod_{j=0}^{k-1} \sigma_j} \\
&= \frac{\boldsymbol{\pi}[\alpha', i_k]}{\boldsymbol{\pi}[\alpha']} = \boldsymbol{\pi}[i_k|\alpha'].
\end{aligned}$$

(P8) implies (P1):

$$\begin{aligned}
\boldsymbol{\pi}[i_k|p] &= \frac{\boldsymbol{\pi}[p, i_k]}{\boldsymbol{\pi}[p]} = \frac{\sum_{\alpha\in\mathcal{A}(p)} \boldsymbol{\pi}[\alpha, i_k]}{\boldsymbol{\pi}[p]} = \frac{\sum_{\alpha\in\mathcal{A}(p)} \boldsymbol{\pi}[i_k|\alpha]\boldsymbol{\pi}[\alpha]}{\boldsymbol{\pi}[p]} \\
&= \boldsymbol{\pi}[i_k|\alpha]\frac{\sum_{\alpha'\in\mathcal{A}(p)} \boldsymbol{\pi}[\alpha']}{\boldsymbol{\pi}[p]} = \boldsymbol{\pi}[i_k|\alpha]
\end{aligned}$$

where the term $\boldsymbol{\pi}[i_k|\alpha]$ can be factored out of the sum due to (P8). □

Note that (P6) relates to our original motivation: a random walk through the MDD, where the choice for state variable $i_k$ depends only on the MDD node $p_k$ reached at level $k$, will choose the path corresponding to state $(i_L, \ldots, i_1)$ with probability $\boldsymbol{\pi}[i_L, \ldots, i_1]$. If we annotate the MDD edges with the choice probabilities (i.e., edge $i_k$ out of $p_k$ is given value $\boldsymbol{\pi}[i_k|p_k]$), then we obtain "almost" the EV*MDD for $\boldsymbol{\pi}$, except the nodes are not reduced, as stated by (P7). Since reduction can only eliminate duplicate nodes, not add new nodes, we know that the EV*MDD for $\boldsymbol{\pi}$ cannot contain more nodes than the MDD for $\mathcal{S}$. Furthermore, we have the following corollary.

**Corollary 4** For any distribution $\boldsymbol{\pi}$ with $\boldsymbol{\pi}[\mathbf{i}] > 0$ for all $\mathbf{i} \in \mathcal{S}$ and such that Eq. 8 holds, the EV*MDD representation of $\boldsymbol{\pi}$ is isomorphic to the MDD representation of $\mathcal{S}$.

**Proof.** Follows from (P7) and, since $\boldsymbol{\pi}[\mathbf{i}] > 0$, the pattern of non–zero downward edges in EV*MDD node $f_{\boldsymbol{\pi}}(1, p)$ matches the pattern of non–zero downward edges in $p$. But this implies that $f_{\boldsymbol{\pi}}(1, p).d \neq f_{\boldsymbol{\pi}}(1, p').d$ for any $p \neq p'$. Therefore, there is a one–to–one mapping from nodes in the MDD to nodes in the EV*MDD. □

We are now ready to present our approximate aggregation. Since our approximation assumes that (P1) holds, by Theorem 2 we have that (P3) and (P5) hold. We therefore simplify Eq. 7 using $\boldsymbol{\pi}[\alpha|p]$ instead of $\boldsymbol{\pi}[\alpha|p, i_k]$ due to (P3), and using $\boldsymbol{\pi}[\beta|p[i_k]]$ instead of $\boldsymbol{\pi}[\beta|\alpha, i_k]$ due to (P5) since $\boldsymbol{\pi}[\beta|\alpha, i_k] = \boldsymbol{\pi}[\beta|r^*_{\mathcal{S}}, \alpha, i_k]$. Substituting these into Eq. 7, we obtain

$$\mathbf{R}_k[[\![p, i_k]\!], [\![q, i'_k]\!]] \approx \rho^*_{\mathbf{R}} \cdot \sum_{\alpha \in \mathcal{A}(p)} \boldsymbol{\pi}[\alpha|p] \cdot \sum_{\alpha' \in \mathcal{A}(q)} r^*_{\mathbf{R}}[\alpha, \alpha'].v \cdot r[i_k, i'_k].v \cdot \sum_{\beta \in \mathcal{B}(p[i_k])} \boldsymbol{\pi}[\beta|p[i_k]] \cdot \sum_{\beta' \in \mathcal{B}(q[i'_k])} r'[\beta, \beta'].v \tag{9}$$

as our estimate for the entries of $\mathbf{R}_k$. We can decompose the right–hand side of Eq. 9 into three parts by letting

$$\mathbf{A}[r, p, q] \equiv \rho^*_{\mathbf{R}} \cdot \sum_{\alpha \in \mathcal{A}(p), \alpha' \in \mathcal{A}(q), r^*_{\mathbf{R}}[\alpha, \alpha'].d = r} \boldsymbol{\pi}[\alpha|p] \cdot r^*_{\mathbf{R}}[\alpha, \alpha'].v,$$

$$\mathbf{B}[r, p, q] \equiv \sum_{\beta \in \mathcal{B}(p), \beta' \in \mathcal{B}(q), r[\beta, \beta'].d = \Omega, r[\beta, \beta'].v \neq 0} \boldsymbol{\pi}[\beta|p] \cdot r[\beta, \beta'].v,$$

where the indices $r$, $p$, and $q$ for the 3-dimensional matrices $\mathbf{A}$ and $\mathbf{B}$ satisfy $r.var = p.var = q.var$, so that Eq. 9 becomes

$$\mathbf{R}_k[[\![p, i_k]\!], [\![q, i'_k]\!]] \approx \sum_{r \in \mathcal{N}(r^*_{\mathbf{R}}, x_k)} \mathbf{A}[r, p, q] \cdot r[i_k, i'_k].v \cdot \mathbf{B}[r[i_k, i'_k].d, p[i_k], q[i'_k]], \tag{10}$$

where $\mathbf{A}$ stands for "above", since $\mathbf{A}[r, p, q]$ depends only on $\mathcal{A}(p)$ and $\mathcal{A}(q)$, while $\mathbf{B}$ stands for "below", since $\mathbf{B}[r, p, q]$ depends only on $\mathcal{B}(p)$ and $\mathcal{B}(q)$.

To compute $\mathbf{A}$ in practice, we note that, for $\alpha' \in \mathcal{A}(p'), p = p'[i_k], \alpha = (\alpha', i_k)$, and thanks to our underlying assumption,

$$\boldsymbol{\pi}[\alpha|p] = \frac{\boldsymbol{\pi}[i_k|\alpha']\boldsymbol{\pi}[\alpha']}{\boldsymbol{\pi}[p]} = \frac{\boldsymbol{\pi}[i_k|p']\boldsymbol{\pi}[\alpha']}{\boldsymbol{\pi}[p]} = \frac{\boldsymbol{\pi}[i_k, p']\boldsymbol{\pi}[\alpha']}{\boldsymbol{\pi}[p']\boldsymbol{\pi}[p]} = \boldsymbol{\pi}[p', i_k|p] \cdot \boldsymbol{\pi}[\alpha'|p'].$$

This allows us to compute $\mathbf{A}[r, p, q]$ top–down using the recurrence

$$\mathbf{A}[r, p, q] = \sum_{p'[i_k] = p, q'[i'_k] = q, r'[i_k, i'_k].d = r} \mathbf{A}[r', p', q'] \cdot \boldsymbol{\pi}[p', i_k|p] \cdot r'[i_k, i'_k].v, \tag{11}$$

terminated with $\mathbf{A}[r^*_{\mathbf{R}}, r^*_{\mathcal{S}}, r^*_{\mathcal{S}}] = \rho^*_{\mathbf{R}}$.

Similarly, for a node $p$ with $\beta = (i_k, \beta') \in \mathcal{B}(p)$,

$$\boldsymbol{\pi}[\beta|p] = \frac{\boldsymbol{\pi}[i_k, \beta', p]}{\boldsymbol{\pi}[p]} = \frac{\boldsymbol{\pi}[\beta'|p, i_k]\boldsymbol{\pi}[p, i_k]}{\boldsymbol{\pi}[p]} = \boldsymbol{\pi}[\beta'|p, i_k]\boldsymbol{\pi}[i_k|p] = \boldsymbol{\pi}[\beta'|p[i_k]]\boldsymbol{\pi}[i_k|p]$$

where the last equality holds due to (P5). We can therefore compute $\mathbf{B}[r, p, q]$ bottom–up using the recurrence

$$\mathbf{B}[r, p, q] = \sum_{i_k, i'_k \in \mathcal{V}_k} \mathbf{B}[r[i_k, i'_k].d, p[i_k], q[i'_k]] \cdot \boldsymbol{\pi}[i_k \mid p] \cdot r[i_k, i'_k].v, \qquad (12)$$

terminated with $\mathbf{B}[\Omega, \mathbf{1}, \mathbf{1}] = 1$.

We exploit (P6) to obtain the probability of a given state. The value $\boldsymbol{\pi}[i_k|p_k]$ is obtained as $\boldsymbol{\pi}[p_k, i_k]/\boldsymbol{\pi}[p_k]$. The probability of a node can be obtained by summing the probabilities of its outgoing edges, i.e.,

$$\boldsymbol{\pi}[p_k] = \sum_{i_k} \boldsymbol{\pi}[p_k, i_k], \qquad (13)$$

where the probabilities $\boldsymbol{\pi}[p_k, i_k]$ correspond to the probabilities of the states in aggregate chain $k$; i.e., $\boldsymbol{\pi}[p_k, i_k] = \boldsymbol{\pi}_k[[\![p_k, i_k]\!]]$. Alternatively, we can determine $\boldsymbol{\pi}[p_k]$ from its incoming edges:

$$\begin{aligned}
\boldsymbol{\pi}[p_k] &= \boldsymbol{\pi}[\mathcal{A}(p_k) \times \mathcal{B}(p_k)] \\
&= \boldsymbol{\pi}\left[\bigcup_{(p_{k+1}, i_{k+1}) \in \mathcal{P}(p_k)} \mathcal{A}(p_{k+1}) \times \{i_{k+1}\} \times \mathcal{B}(p_k)\right] \\
&= \sum_{(p_{k+1}, i_{k+1}) \in \mathcal{P}(p_k)} \boldsymbol{\pi}[\mathcal{A}(p_{k+1}) \times \{i_{k+1}\} \times \mathcal{B}(p_k)] \\
&= \sum_{(p_{k+1}, i_{k+1}) \in \mathcal{P}(p_k)} \boldsymbol{\pi}[p_{k+1}, i_{k+1}], \qquad (14)
\end{aligned}$$

where $\mathcal{P}(p_k)$ denotes the set of pairs $(p_{k+1}, i_{k+1})$ such that $p_{k+1}[i_{k+1}] = p_k$.

Finally, although Corollary 4 gives us the fixed shape of the EV*MDD representation of $\boldsymbol{\pi}$, in practice we do not use an EV*MDD to represent $\boldsymbol{\pi}$. Instead we obtain $\boldsymbol{\pi}$ from the probability distributions of the aggregate chains, as discussed above. However, the two representations are equivalent: it can be shown that the edge value $q_k[i_k].v$ is proportional to $\boldsymbol{\pi}[i_k|p_k]$, where $q_k$ is the node in EV*MDD $f_{\boldsymbol{\pi}}(1, r^*_{\mathcal{S}})$ corresponding to node $p_k$ in MDD $r^*_{\mathcal{S}}$. Note that, if EV*MDD normalization required that the *sum*, instead of the *maximum* of the edge values leaving a node were equal to one (as in "probabilistic decision diagrams" [3]), then $q_k[i_k].v$ would instead exactly equal $\boldsymbol{\pi}[i_k|p_k]$.

### *4.3 The algorithm*

The key step is the computation of the three-dimensional matrices $\mathbf{A}$ and $\mathbf{B}$. Observe that $\mathbf{A}[r, p, q] > 0$ only if either $r = r^*_{\mathbf{R}}$ and $p = q = r^*_{\mathcal{S}}$, or there

are nodes $r', p', q'$ and integers $i, j$ such that $\mathbf{A}[r', p', q'] > 0$, $r'[i, j].d = r$, $p'[i] = p$, and $q'[j] = q$. Similarly, $\mathbf{B}[r, p, q] > 0$ only if either $r = \Omega$ and $p = q = \mathbf{1}$, or there exist nodes $r', p', q'$ and integers $i, j$ such that $\mathbf{B}[r', p', q'] > 0$, $r[i, j].d = r'$, $p[i] = p'$, and $q[j] = q'$. Thus, the *positions* (not the *values*) of the non–zero elements of $\mathbf{A}$ and $\mathbf{B}$ can be found *a priori* by concurrently traversing EV*MDD $r^*_{\mathbf{R}}$ and MDD $r^*_{\mathcal{S}}$ top–down and bottom–up, respectively.

Another important observation is that, if either $\mathbf{A}[r, p, q]$ or $\mathbf{B}[r, p, q]$ is zero according to the above traversal, we do not need to store the other. This is because $\mathbf{A}[r, p, q]$ is only used in a product with some $\mathbf{B}[r', p', q']$, where there exist $i, j$ such that $r[i, j].d = r', p[i] = p', q[j] = q'$; if $\mathbf{B}[r, p, q]$ is zero, any of these $\mathbf{B}[r', p', q']$ must also be zero. This further reduces the memory requirements for $\mathbf{A}$ and $\mathbf{B}$, which are stored as sparse matrices whose non–zero elements have the same set of indices except for the terminal cases. More precisely, we only need to store entries $[r, p, q]$ such that there exist paths $\alpha, \alpha'$ where $r^*_{\mathbf{R}}[\alpha, \alpha'].d = \Omega, r^*_{\mathbf{R}}[\alpha, \alpha'].v > 0, r^*_{\mathcal{S}}[\alpha] = r^*_{\mathcal{S}}[\alpha'] = \mathbf{1}$, and $r$ lies on path $(\alpha, \alpha')$, $p$ lies on $\alpha$, $q$ lies on $\alpha'$, which leads to the following algorithm.

In Fig 8, procedure *BuildCorr* finds all *corresponding* MDD node pairs for each EV*MDD node, such that $(p, q) \in r.corr$ means that we need to compute element $[r, p, q]$ in $\mathbf{A}$ and $\mathbf{B}$, by traversing down from the root node. We also compute $r.corr$ if $r$ is associated with a primed variable, which might at worst double the size of $\mathbf{A}$ and $\mathbf{B}$ but can greatly simplify the computation and better utilize the cache. Storing those extra elements for $\mathbf{A}$ and $\mathbf{B}$ is also efficient when computing $\mathbf{A}[r, p, q]$ or $\mathbf{B}[r, p, q]$ from another $\mathbf{A}[r', p', q']$ or $\mathbf{B}[r', p', q']$ according to Eq. 11 and Eq. 12, which allows a recursive computation with the least information, shown later.

*BuildCorr* returns a boolean value to indicate whether we should add $(p, q)$ to $r.corr$. It returns *false* when we can reach $r$ from $r^*_{\mathbf{R}}$ through $(\alpha, \alpha')$, but $\alpha$ is not a valid path in $r^*_{\mathcal{S}}$. This can happen because each $r^*_{\mathcal{T}_e}$ encodes the *potential* transitions [10], i.e., $(\mathbf{i}, \mathbf{i}') \in \mathcal{T}_e$ does not necessarily indicates $\mathbf{i} \in \mathcal{S}$ (the same "fictitious rates" arise with a Kronecker representation; this is a common strategy to encode transition relations as it usually results in a much more compact encoding). Since $r^*_{\mathbf{R}}$ is obtained from $r^*_{\mathcal{T}_e}$, it, too, might contain fictitious rates. Of course, unlike a Kronecker representation, the EV*MDD encoding allow us to use $\mathcal{S}$ as a filter and actually eliminate these fictitious rates; this preprocessing can be seamlessly integrated into *BuildCorr* as shown in the pseudocode. Sect. 5 lists the number of nodes and edges of the EV*MDD encoding the actual $\mathbf{R}$ after filtering these fictitious rates; we do not perform this filtering in our approach and present this data for comparison only.

Fig. 9 and Fig. 10 show the pseudocode for our approximation. Procedure *ComputeAbove*$(x_k)$ computes $\mathbf{A}$ for all elements $[r, p, q]$ with $r.var = x'_{k+1}$ and then $r.var = x_k$, by traversing all elements $[r', p', q']$ with $r.var = x_{k+1}$

and then $r.var = x'_{k+1}$ and applying Eq. 11. Procedure *ComputeBelow*$(x_k)$ computes all elements $[r, p, q]$ of $\mathbf{B}$, with $r.var = x'_k$ and then $r.var = x_k$, by traversing all elements $[r', p', q']$ with $r.var = x'_k$ and then $r.var = x_k$, based on Eq. 12. These two procedures benefit from a sparse representation of matrices $\mathbf{A}$ and $\mathbf{B}$, since they traverse only the non–zero elements. During the computation, $r[i, i'].v$ is broken into $r[i].v \cdot r[i][i'].v$; $\boldsymbol{\pi}[p', i_k|p]$ is computed as $\boldsymbol{\pi}_k[[\![p', i_k]\!]]/\boldsymbol{\pi}[p]$; and $\boldsymbol{\pi}[i_k|p]$ is obtained from $\boldsymbol{\pi}_k[[\![p, i_k]\!]]/\boldsymbol{\pi}[p]$. We can then see how storing $r.corr$ for $r$ associated with a primed variable simplifies the computation, since now we can compute $\mathbf{A}[r, p, q]$ from another $\mathbf{A}[r', p', q']$ by traversing only $r$'s outgoing edges and either $p$ or $q$'s outgoing edges (depending on $r.var$); the same holds for $\mathbf{B}$. Thus, in the actual data structures set up for $\mathbf{A}$ and $\mathbf{B}$, only one EV*MDD node pointer and one MDD node pointer need to be stored for each non–zero element.

Procedure *SolveVariable*$(x_k)$ is used to build and solve the aggregated CTMC for $x_k$. The probability vector $\boldsymbol{\pi}_k$ satisfying $\boldsymbol{\pi}_k \cdot \mathbf{Q}_k = \mathbf{0}$ can be determined using an iterative method such as Jacobi or Gauss-Seidel [24]. As the probability vectors change, so do the rates between these macrostates, and the vector $\boldsymbol{\pi}_k$ must be recomputed. To accelerate this process, we use the previous solution to $\boldsymbol{\pi}_k$ as the initial vector for the iterative method when recomputing $\boldsymbol{\pi}_k$. Doing so causes the number of iterations required to solve $\boldsymbol{\pi}_k$ to decrease as the overall fixpoint computation converges. In particular, if the recomputation of $\boldsymbol{\pi}_k$ requires a *single* iteration only, then we say that $\boldsymbol{\pi}_k$ is *stable*; otherwise we say $\boldsymbol{\pi}_k$ is *unstable*. Note that changing $\boldsymbol{\pi}_h$ for some other variable $x_h$ can cause a stable $\boldsymbol{\pi}_k$ to again become unstable. When $\boldsymbol{\pi}_k$ is computed at each global iteration, the probability of each node $p$, $\boldsymbol{\pi}[p]$, with $p.var = x_k$, is updated according to Eq. 13 and is used for the next global iteration; in the meantime, $\boldsymbol{\pi}[p]$, with $p.var = x_{k-1}$, is also updated, this time according to Eq. 14, as it is used for *ComputeAbove*$(x_{k-1})$ in the current global iteration, also improving convergence.

Procedure *Solve* performs the overall fixpoint computation, assuming that the MDD encoding $\mathcal{S}$ and the EV*MDD encoding $\mathbf{R}$ have been built, and that the vectors $\boldsymbol{\pi}_k$ for each $x_k$ are set to some initial distribution (we use the uniform distribution). A stopping criterion must be used to determine when the overall computation has reached a fixpoint. We stop the global iterations when every local vector $\boldsymbol{\pi}_k$ is stable.

### *4.4 Computing measures*

After obtaining an approximation for $\boldsymbol{\pi}$, we typically wish to compute measures of interest defined via a reward rate function $\theta : \mathcal{S} \to \mathbb{R}$. For example, $m = \sum_{\mathbf{i} \in \mathcal{S}} \theta(\mathbf{i}) \cdot \boldsymbol{\pi}[\mathbf{i}]$ represents the expected reward rate in steady state. Since

```
boolean BuildCorr(evmdd node r, mdd p, mdd q)
 1 if r = Ω then return true;                          • terminal case
 2 if CacheHit(CORR, r, p, q, answer) then return answer;   • check cache
 3 answer ← false;
 4 if r.var = p.var then                  • r is associated to unprimed variable
 5   foreach i ∈ V_r s.t. p[i] ≠ 0 and r[i] ≠ ⟨0,Ω⟩ do
 6     if BuildCorr(r[i].d, p[i], q) = true then answer ← true;
 7 else                                     • r is associated to primed variable
 8   foreach i ∈ V_r s.t. q[i] ≠ 0 and r[i] ≠ ⟨0,Ω⟩ do
 9     if BuildCorr(r[i].d, p, q[i]) = true then answer ← true;
10 if answer = true then          • [r, p, q] is a non–zero element for A and B
11   r.corr ← r.corr ∪ {(p, q)};
12 CacheAdd(CORR, r, p, q, answer);
13 return answer;
```

Fig. 8. Setting up non–zeros for $\mathbf{A}$ and $\mathbf{B}$.

```
Solve()
 1 BuildCorr(r*_R, r*_S, r*_S);
 2 while not converged do
 3   A ← 0; B ← 0; R ← 0;
 4   for k ← 0 to L do ComputeBelow(x_k);
 5   for k ← L to 1 do
 6     ComputeAbove(x_k);
 7     SolveVariable(x_k);
```

Fig. 9. The global iteration.

computing $\boldsymbol{\pi}[\mathbf{i}]$ requires a product according to the tuple of $L$ nodes visited along the path $\mathbf{i}$, the time to compute $m$ according to this sum is $O(L \cdot |\mathcal{S}|)$, which can be extremely large.

However, if the reward function can be expressed as the product of functions on state variables, $\theta(\mathbf{x}) = \theta_L(x_L) \cdots \theta_2(x_2) \cdot \theta_1(x_1)$, we can express $m$ as

$$m = \sum_{(i_L,\ldots,i_1)\in\mathcal{S}} \boldsymbol{\pi}[i_L,\ldots,i_1] \prod_{k=1}^{L} \theta_k(i_k) \tag{15}$$

$$= \sum_{(i_L,\ldots,i_1)\in\mathcal{S}} \prod_{k=1}^{L} \boldsymbol{\pi}[i_k|p_k] \cdot \theta_k(i_k) \tag{16}$$

when (P6) holds. This equation can be written as a recurrence, by defining the value of the measure for a node $p \in \mathcal{N}(r^*_{\mathcal{S}}, x_k)$ as

$$m(p) \;=\; \sum_{i_k : p[i_k]\neq 0} \theta_k(i_k) \cdot \boldsymbol{\pi}[i_k|p] \cdot m(p[i_k]), \tag{17}$$

*ComputeAbove*(*variable* $x_k$)

1 if $k = L$ then $\mathbf{A}[r^*_{\mathbf{R}}, r^*_{\mathcal{S}}, r^*_{\mathcal{S}}] \leftarrow \rho^*_{\mathbf{R}}$; return; • *terminal case*
2 foreach $r \in \mathcal{N}(r^*_{\mathbf{R}}, x_{k+1})$ do
3  foreach $(p, q) \in r.corr$ do
4  foreach $i \in \mathcal{V}_{k+1}$ s.t. $p[i] \neq \mathbf{0}$ and $r[i] \neq \langle 0,\Omega \rangle$ do
5  $r' \leftarrow r[i].d$;  $p' \leftarrow p[i]$;
6  $\mathbf{A}[r', p', q] \leftarrow \mathbf{A}[r', p', q] + \mathbf{A}[r, p, q] \cdot \boldsymbol{\pi}_{k+1}[[\![p, i]\!]] \cdot r[i].v$;
7 foreach $r \in \mathcal{N}(r^*_{\mathbf{R}}, x'_{k+1})$ do
8  foreach $(p, q) \in r.corr$ do
9  foreach $i \in \mathcal{V}_{k+1}$ s.t. $q[i] \neq \mathbf{0}$ and $r[i] \neq \langle 0,\Omega \rangle$ do
10  $r' \leftarrow r[i].d$;  $q' \leftarrow q[i]$;
11  $\mathbf{A}[r', p, q'] \leftarrow \mathbf{A}[r', p, q'] + \mathbf{A}[r, p, q] \cdot r[i].v / \boldsymbol{\pi}_k[p]$;

*ComputeBelow*(*variable* $x_k$)

1 if $k = 0$ then $\mathbf{B}[\Omega, \mathbf{1}, \mathbf{1}] \leftarrow 1$; return; • *terminal case*
2 foreach $r \in \mathcal{N}(r^*_{\mathbf{R}}, x'_k)$ do
3  foreach $(p, q) \in r.corr$ do
4  foreach $i \in \mathcal{V}_k$ s.t. $q[i] \neq \mathbf{0}$ and $r[i] \neq \langle 0,\Omega \rangle$ do
5  $r' \leftarrow r[i].d$;  $q' \leftarrow q[i]$;
6  $\mathbf{B}[r, p, q] \leftarrow \mathbf{B}[r, p, q] + \mathbf{B}[r', p, q'] \cdot r[i].v$;
7 foreach $r \in \mathcal{N}(r^*_{\mathbf{R}}, x_k)$ do
8  foreach $(p, q) \in r.corr$ do
9  foreach $i \in \mathcal{V}_k$ s.t. $p[i] \neq \mathbf{0}$ and $r[i] \neq \langle 0,\Omega \rangle$ do
10  $r' \leftarrow r[i].d$;  $p' \leftarrow p[i]$;
11  $\mathbf{B}[r, p, q] \leftarrow \mathbf{B}[r, p, q] + \mathbf{B}[r', p', q] \cdot r[i].v \cdot \boldsymbol{\pi}_k[[\![p, i]\!]] / \boldsymbol{\pi}_k[p]$;

*SolveVariable*(*variable* $x_k$)

1 foreach $r \in \mathcal{N}(r^*_{\mathbf{R}}, x_k)$ do
2  foreach $(p, q) \in r.corr$ do
3  foreach $i \in \mathcal{V}_k$ s.t. $p[i] \neq \mathbf{0}$ and $r[i] \neq \langle 0,\Omega \rangle$ do
4  foreach $j \in \mathcal{V}_k$ s.t. $q[j] \neq \mathbf{0}$ and $r[i][j] \neq \langle 0,\Omega \rangle$ do
5  $r' \leftarrow r[i][j].d$;  $p' \leftarrow p[i]$;  $q' \leftarrow q[i]$;
6  $\mathbf{R}_k[[\![p, i]\!], [\![q, j]\!]] \leftarrow \mathbf{R}_k[[\![p, i]\!], [\![q, j]\!]] + \mathbf{A}[r, p, q] \cdot \mathbf{B}[r', p', q'] \cdot r[i].v \cdot r[i][j].v$;
7 $\boldsymbol{\pi}_k \leftarrow$ solution of $\boldsymbol{\pi}_k \cdot \mathbf{Q}_k = \mathbf{0}$; • *e.g., using Gauss-Seidel or Jacobi*
8 foreach $p \in \mathcal{N}(r^*_{\mathcal{S}}, x_k)$ do
9  $\boldsymbol{\pi}[p] \leftarrow \sum_{i \in \mathcal{V}_k} \boldsymbol{\pi}_k[[\![p, i]\!]]$; • *update* $\boldsymbol{\pi}[p]$ *according to Eq. 13*
10 if $k > 1$ then
11  foreach $p \in \mathcal{N}(r^*_{\mathcal{S}}, x_{k-1})$ do $\boldsymbol{\pi}_{k-1}[p] \leftarrow 0$;
12  foreach $[\![p, i]\!]$ s.t. $p.var = x_k$ do
13  $\boldsymbol{\pi}[p[i]] \leftarrow \boldsymbol{\pi}[p[i]] + \boldsymbol{\pi}_k[[\![p, i]\!]]$; • *update* $\boldsymbol{\pi}[p[i]]$ *according to Eq. 14*

Fig. 10. Steps in the global iteration.

where the recurrence terminates with $m(\mathbf{1}) = 1$. This allows us to compute $m = m(r^*_{\mathcal{S}})$ in a bottom–up fashion, by visiting each MDD node $p$ only once and scanning the non–zero edges of $p$, with a computational cost proportional

to the number of non–zero edges in the MDD, normally *much* smaller than $O(L \cdot |\mathcal{S}|)$. Related statistics are listed in Sect. 5.

### *4.5 Product–form models*

We now prove that the assumption required to derive our algorithm is satisfied when the model has a product–form. The proof is a generalization of the one in [20]. Suppose the model has a product–form stationary distribution and has been decomposed accordingly. Then, for any reachable state $\mathbf{i} \in \mathcal{S}$, the product–form property says that

$$\boldsymbol{\pi}[\mathbf{i}] = G^{-1} \prod_{k=1}^{L} g_k(i_k) \tag{18}$$

where $g_L, \ldots, g_1$ are appropriate functions and $G$ is a normalization constant simply needed to ensure that the probabilities sum to one.

**Theorem 5** If Equation 18 holds, then so does Equation 8.

**Proof.** Suppose Eq. 18 holds. Consider some node $p \in \mathcal{N}(r^*_{\mathcal{S}}, x_k)$, and a path $\alpha = (i_L, \ldots, i_{k+1}) \in \mathcal{A}(p)$. Then, we have

$$\boldsymbol{\pi}[\alpha] = \sum_{\beta \in \mathcal{B}(p)} \boldsymbol{\pi}[\alpha, \beta] = \sum_{(i_k, \ldots, i_1) \in \mathcal{B}(p)} G^{-1} \prod_{j=1}^{L} g_j(i_j) = G^{-1} \prod_{j=k+1}^{L} g_j(i_j) \sum_{(i_k, \ldots, i_1) \in \mathcal{B}(p)} \prod_{j=1}^{k} g_j(i_j).$$

This gives us

$$\begin{aligned} \boldsymbol{\pi}[i_k|\alpha] = \frac{\boldsymbol{\pi}[\alpha, i_k]}{\boldsymbol{\pi}[\alpha]} &= \frac{G^{-1} \prod_{j=k}^{L} g_j(i_j) \sum_{(i_{k-1}, \ldots, i_1) \in \mathcal{B}(p[i_k])} \prod_{j=1}^{k-1} g_j(i_j)}{G^{-1} \prod_{j=k+1}^{L} g_j(i_j) \sum_{(i'_k, \ldots, i'_1) \in \mathcal{B}(p)} \prod_{j=1}^{k} g_j(i'_j)} \\ &= \frac{g_k(i_k) \sum_{(i_{k-1}, \ldots, i_1) \in \mathcal{B}(p[i_k])} \prod_{j=1}^{k-1} g_j(i_j)}{\sum_{(i'_k, \ldots, i'_1) \in \mathcal{B}(p)} \prod_{j=1}^{k} g_j(i'_j),} \end{aligned}$$

an expression that depends on $p$ but not on $\alpha$. Thus, it follows that (P8) holds and, from Theorem 3, we have that Eq. 8 holds as well. □

Since all estimates used in our approximation algorithm are derived from Eq. 8, Theorem 5 establishes that the exact solution vector does indeed correspond to a fixpoint for our algorithm when the model has a product form. This provides further justification for our use of Eq. 8 as an underlying assumption, as well as insight about the class of models for which our algorithm can give exact or accurate results.

We note that the converse of Theorem 5 does not hold. For instance, consider a model that produces a state space $\mathcal{S}$ such that the MDD encoding $\mathcal{S}$ has the property that $|\mathcal{A}(p)| = 1$ for any node $p$. For *any* such model, product–form or not, we know that Eq. 8 holds, since $\alpha$ is unique for a given $p$. While this is theoretically interesting, it is not useful in practice, as the MDD is actually a tree in this case. As such, the MDD will contain a total of $|\mathcal{S}|$ edges from nodes in $\mathcal{N}(r^*_{\mathcal{S}}, x_1)$, and since each macrostate in $\mathbf{R}_1$ corresponds to exactly one state of $\mathcal{S}$, we have that $\mathbf{R}_1 = \mathbf{R}$. In other words, in this case our approximation algorithm must analyze the entire original CTMC, which is of course the problem we are trying to avoid.

### *4.6 Complexity, accuracy, and convergence*

The overall time complexity of our approximate solution is $O(iters \cdot (\eta(\mathbf{A}) + T))$, where *iters* is the number of global iterations, $\eta(\mathbf{A})$ is the number of non–zero elements of (three–dimensional) matrix $\mathbf{A}$ and $T$ is the average complexity to compute each $\boldsymbol{\pi}_k$ using the Gauss-Seidel or Jacobi method at each global iteration, which is of course related to each $\eta(\mathbf{R}_k)$.

Considering the number of possible non–zero elements in matrix $\mathbf{A}$, we see that, for a given state variable $x_k$, there can be at most

$$|\mathcal{N}(r^*_{\mathbf{R}}, x_k)| \cdot |\mathcal{N}(r^*_{\mathcal{S}}, x_k)|^2 + |\mathcal{N}(r^*_{\mathbf{R}}, x'_k)| \cdot |\mathcal{N}(r^*_{\mathcal{S}}, x_k)|^2 \tag{19}$$

non–zero elements in $\mathbf{A}$, namely, one for each triple $(r, p, q)$ that could possibly occur for state variable $x_k$. A bound for the worst–case size of $\eta(\mathbf{A})$ can be determined by summing Eq. 19 over all levels $L \geq k \geq 1$; alternatively, a looser bound is given by $|\mathcal{N}(r^*_{\mathbf{R}})| \cdot |\mathcal{N}(r^*_{\mathcal{S}})|^2$. In practice, however, $\eta(\mathbf{A})$ tends to be much smaller than the worst-case bound. For example, the value of $|\mathcal{N}(r^*_{\mathbf{R}})|$ is shown in Table 4, and the values of $|\mathcal{N}(r^*_{\mathcal{S}})|$ and $\eta(\mathbf{A})$ are shown in Table 5, for a flexible manufacturing system model. Looking at the case $N = 20$ with decomposition $L = 22$, we see that a single–root EV*MDD requires $|\mathcal{N}(r^*_{\mathbf{R}})| =$ 18,916 nodes, the MDD encoding of $\mathcal{S}$ requires $|\mathcal{N}(r^*_{\mathcal{S}})| =$ 1,946 nodes, and the loose bound is $\eta(\mathbf{A}) \leq |\mathcal{N}(r^*_{\mathbf{R}})| \cdot |\mathcal{N}(r^*_{\mathcal{S}})|^2 =$ 71,633,303,056, while, summing Eq. 19 over all levels, we obtain a tighter bound of $\eta(\mathbf{A}) \leq$ 412,946,429. In reality, the actual value is $\eta(\mathbf{A}) =$ 280,144.

We recall that the approximation in our algorithm is due to assuming Eq. 8, or equivalently, any of the properties listed in Theorems 2 and 3. Thus, the accuracy of our algorithm depends on how well this assumption holds. If the assumption holds true (as for product–form models), then our algorithm has the potential to give exact results, in the sense that, if it converges to a unique fixpoint, then the exact solution must be that fixpoint. In general, it seems to be difficult to state anything meaningful if the assumption "almost" holds

| Place | Exact | Approx | |
|---|---|---|---|
| | | $L = 3$ | $L = 4$ |
| $a$ | 0.656393 | 0.656393 | 0.656393 |
| $b$ | 1.31279 | 1.31278 | 1.31278 |
| $c$ | 0.328196 | 0.328196 | 0.328196 |
| $d$ | 0.0308214 | 0.0308214 | 0.0308214 |
| $e$ | 1.01541 | 1.01541 | 1.01541 |

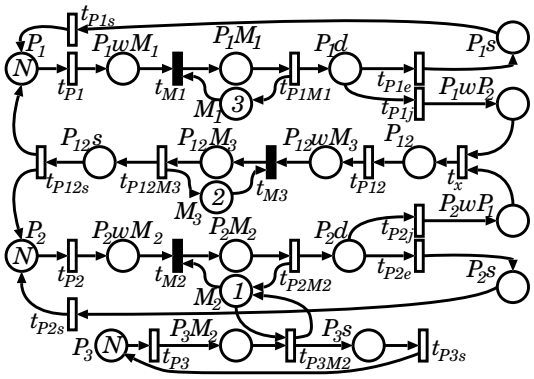
Table 2
Running example: expected number of tokens in each place, for $N = 2$.

(e.g., if $\max_{\alpha \in \mathcal{A}(p)}\{|\boldsymbol{\pi}[i_k|p] - \boldsymbol{\pi}[i_k|\alpha]|\} < \epsilon$), because, even assuming the algorithm can exactly compute the quantities $\boldsymbol{\pi}[i_k|p]$ from the aggregate CTMCs (with rates calculated incorrectly), the error of a particular measure $m$ will depend on its reward function $\theta$: comparing the exact value of $m$, given by Eq. 15, with the computed value of $m$, given by Eq. 16, we see that the error depends on $\theta(i_k)$ and the difference between $\boldsymbol{\pi}[i_k|p]$ and $\boldsymbol{\pi}[i_k|\alpha]$. To make matters worse, it is difficult to say how our computed $\boldsymbol{\pi}[i_k|p]$ will be affected by errors in the rates of the aggregate CTMCs. In practice, though, our algorithm appears to give quite accurate results. For example, Table 2 shows the results from an exact solution and from our approximation algorithm, for our running example model with $N = 2$. In this case, the maximum observed relative error is within 0.001%.

Using an under–relaxation parameter (e.g., $\omega = 0.95$) with the Gauss-Seidel or Jacobi methods, the computation of each $\boldsymbol{\pi}_k$ is guaranteed to converge [24]. However, global convergence is not guaranteed in general, thus we can only set a bound on the number of global iterations, to guard against models and decompositions that do not converge.

## 5 Experimental results

In this section, we report experimental results and related statistics of our algorithm, when run on a set of benchmarks. We will mainly discuss results of a flexible manufacturing system (FMS) model from [12], but also list results for other models to demonstrate the applicability of our method. All algorithms are implemented in SMART [6] and all experiments are run on an Intel Xeon 3.0Ghz workstation with 4GB RAM under SuSE Linux 9.1. For all experiments, we set up a time bound of one hour, a memory bound of 4GB, and a maximum of 10,000 global iterations.




| trans. | rate | trans. | rate |
|---|---|---|---|
| $t_{P1}$ | $\#(P_1)\cdot\mu$ | $t_{P2}$ | $\#(P_2)\cdot\mu$ |
| $t_{P3}$ | $\#(P_3)\cdot\mu$ | $t_{P12}$ | $\#(P_{12})\cdot\mu$ |
| $t_{P1M1}$ | $\#(P_{1M1})/4$ | $t_{P2M2}$ | 1/6 |
| $t_{P3M2}$ | 1/2 | $t_{P12M3}$ | $\#(P_{12M3})$ |
| $t_{P1s}$ | $\#P_{1s}/60$ | $t_{P2s}$ | $\#P_{2s}/60$ |
| $t_{P3s}$ | $\#P_{3s}/60$ | $t_{P12s}$ | $\#P_{12s}/60$ |
| $t_{P1e}$ | 80 | $t_{P1j}$ | 20 |
| $t_{P2e}$ | 60 | $t_{P2j}$ | 40 |
| $t_x$ | 100 | | |

Fig. 11. Benchmark: the FMS model.

### *5.1 Models used in the experiments*

Fig. 11 models a flexible manufacturing system (FMS) [12] with $N$ pallets to move different types of parts to various machines. Transitions $t_{P1}$, $t_{P2}$, $t_{P3}$, and $t_{P12}$ share these pallets in a "processor sharing" fashion; for example, the rate of $t_{P1}$ is $\#(P_1)\cdot\mu$, where $\#(P_1)$ is the number of tokens in the input place of $t_{P1}$, and $\mu = \min\left\{1, \frac{N}{\#(P_1)+\#(P_2)+\#(P_3)}\right\}$ is a slowdown factor evaluating to 1 if the total number of pallets in use is at most $N$, or to less than 1 if the requests exceed the pallets. Due to this dependency, under the Kronecker restriction, $P_1$, $P_2$ and $P_3$ must be encoded by the same state variable. Alternatively, transitions $t_{P_1}$, $t_{P_2}$, and $t_{P_3}$ can each be split into $(N+1)^3$ distinct transitions (one for each combination of tokens in places $P_1$, $P_2$, and $P_3$), such that each split transition satisfies the Kronecker requirement; however, this approach leads to excessive storage and computational requirements, thus we do not consider it and we do not present results for it. Additionally, due to the presence of immediate transitions, $\{P_1wM_1, M_1, P_1M_1\}$, $\{P_2wM_2, M_2, P_2M_2\}$, and $\{P_{12}wM_3, M_3, P_{12}M_3\}$ must also be put into one variable each to satisfy the Kronecker requirement. The other places in the model can each be encoded by one distinct variable. As such, the finest possible Kronecker decomposition uses $L = 14$ state variables (denoted *fms*-14). We also consider a decomposition with one state variable per place, requiring $L = 22$ variables (denoted *fms*-22); this decomposition does not satisfy the Kronecker requirement. Our target measure is $\Pr\{\#(P_{12}s) = 0\}$, the probability that there are no completed parts of type $P_{12}$ waiting to ship.

We also consider: the kanban system from [25], with two different decompositions, *kan*-4, a decomposition with $L = 4$ state variables (using one state variable per "component"), and *kan*-16, a decomposition with $L = 16$ state variables; a multiserver polling system [18] (model "*polling*"); and models of round robin and slotted ring protocols [13] (models "*robin*" and "*slot*"). For

each of these models, we use a decomposition where each system component is represented with a single state variable.

### *5.2 Algorithm comparison*

Table 3 compares the performance of four different solution approaches.

- The *Explicit* solution stores $\mathbf{R}$ explicitly and computes $\boldsymbol{\pi}$ using Gauss-Seidel (*G-S*) for the numerical solution, with a relative stopping criterion set to $10^{-4}$, i.e., the iterations stop when $\max_{\mathbf{i}\in\mathcal{S}} |\boldsymbol{\pi}[\mathbf{i}] - \boldsymbol{\pi}^{old}[\mathbf{i}]|/\boldsymbol{\pi}[\mathbf{i}] \leq 10^{-4}$. For this solution, we use the general state–space generation technique from [26].
- Discrete–event *Simulation* computes the given measure to within an interval of relative width 2% with 99% confidence, i.e., the number of simulation runs is such that, with probability 0.99, the exact value of the measure is the computed point estimate plus or minus 1%. Of course, simulation does not require state–space generation.
- The *Kronecker–based* approximation uses the approximate technique in [20], where $\mathbf{R}$ is stored as a Kronecker expression, and is applicable only when the decomposition of the model is Kronecker–consistent. When solving each aggregate CTMC, we use Gauss-Seidel or Jacobi with a relative precision of $10^{-4}$, and the global fixpoint iterations stop when every aggregate CTMC solution converges within one Gauss-Seidel or Jacobi iteration. To guarantee convergence in the aggregate CTMC solutions, we use a relaxation parameter of 0.95 for both Gauss-Seidel and Jacobi. This solution uses the state–space generation from [8].
- Our EV*MDD–based approximation method, always applicable, uses the same stopping criteria as the Kronecker–based solution, and the state–space generation technique from [26].

Except for simulation, each approach requires an "initialization" step that includes state–space generation, generation of the transition rate matrix or its symbolic encoding, and setting up related computational data structures. $T_{init}$ denotes the time required to perform these steps. $T_{solve}$ denotes the time required to compute the desired performance measure, after the initialization step is completed, i.e., the time to solve the CTMC using Gauss-Seidel for exact solution, the total time required for simulation, or the time to run the appropriate approximation algorithm.

For cases where explicit solution is possible (currently, when the CTMC has fewer than about 10 million states), we see that our EV*MDD-based technique requires much lower runtimes and produces results that are quite accurate, usually within 1%. For larger cases, we can only compare our approximation results with those from simulation. We realize that comparing the runtime of

| Model | $N$ | Explicit solution | | | | Simulation | | Kronecker-based approximation | | | | | | EV*MDD-based approximation | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | G-S | | | | | | G-S | | Jacobi | | | | G-S | | Jacobi | | |
| | | $T_{init}$ | $T_{solve}$ | iters | meas. | $T_{solve}$ | meas. | $T_{init}$ | $T_{solve}$ | iters | $T_{solve}$ | iters | meas. | $T_{init}$ | $T_{solve}$ | iters | $T_{solve}$ | iters | meas. |
| *fms*-14 | 5 | 64.97 | 33.91 | 785 | 0.3621 | 18.64 | 0.3688 | 0.23 | 0.47 | 22 | 0.56 | 23 | 0.3630 | 0.34 | 0.24 | 20 | 0.31 | 23 | 0.3630 |
| *fms*-22 | 5 | — | — | — | — | — | — | n/a | n/a | n/a | n/a | n/a | n/a | 0.08 | 0.18 | 21 | 0.04 | 58 | 0.3630 |
| *fms*-14 | 6 | 334.03 | 200.25 | 946 | 0.2952 | 21.21 | 0.3040 | 0.39 | 0.90 | 20 | 1.05 | 22 | 0.2964 | 0.60 | 0.51 | 20 | 0.64 | 22 | 0.2964 |
| *fms*-22 | 6 | — | — | — | — | — | — | n/a | n/a | n/a | n/a | n/a | n/a | 0.11 | 0.24 | 22 | 0.03 | 19 | 0.2964 |
| *fms*-14 | 10 | o-m | o-m | o-m | o-m | 119.62 | 0.1431 | 2.57 | 6.38 | 22 | 8.15 | 23 | 0.1319 | 4.61 | 4.44 | 22 | 5.35 | 23 | 0.1319 |
| *fms*-22 | 10 | — | — | — | — | — | — | n/a | n/a | n/a | n/a | n/a | n/a | 0.32 | 4.31 | 24 | 0.15 | 17 | 0.1319 |
| *fms*-14 | 20 | o-m | o-m | o-m | o-m | 451.17 | 0.0191 | 123.62 | 557.38 | 144 | 590.97 | 106 | 0.0178 | 139.88 | 359.08 | 100 | 453.94 | 106 | 0.0178 |
| *fms*-22 | 20 | — | — | — | — | — | — | n/a | n/a | n/a | n/a | n/a | n/a | 1.87 | 45.42 | 98 | 18.03 | 217 | 0.0178 |
| *kan*-4 | 5 | 122.38 | 43.13 | 267 | 0.6344 | 0.27 | 0.6288 | 0.06 | 0.00 | 10 | 0.00 | 10 | 0.6346 | 0.09 | 0.00 | 10 | 0.00 | 10 | 0.6346 |
| *kan*-16 | 5 | — | — | — | — | — | — | 0.03 | 0.01 | 11 | 0.01 | 12 | 0.6344 | 0.04 | 0.00 | 11 | 0.00 | 12 | 0.6344 |
| *kan*-4 | 15 | o-m | o-m | o-m | o-m | 0.65 | 0.5282 | 0.93 | 0.12 | 8 | 0.20 | 8 | 0.5319 | 22.08 | 0.09 | 8 | 0.14 | 8 | 0.5319 |
| *kan*-16 | 15 | — | — | — | — | — | — | 0.37 | 0.08 | 8 | 0.12 | 9 | 0.5318 | 0.26 | 0.06 | 8 | 0.07 | 9 | 0.5318 |
| *polling* | 5 | 2.43 | 0.15 | 58 | 0.8889 | 0.98 | 0.8890 | 0.03 | 0.00 | 6 | 0.00 | 16 | 0.8889 | 0.02 | 0.00 | 6 | 0.00 | 38 | 0.8889 |
| *polling* | 30 | o-m | o-m | o-m | o-m | 1.23 | 0.9815 | 1.02 | 0.19 | 31 | n-c | n-c | 0.9815 | 4.36 | 0.09 | 31 | n-c | n-c | 0.9815 |
| *robin* | 5 | 0.06 | 0.00 | 15 | 0.2048 | 0.53 | 0.2046 | 0.03 | 0.01 | 16 | n-c | n-c | 0.2046 | 0.02 | 0.00 | 16 | n-c | n-c | 0.2046 |
| *robin* | 15 | o-m | o-m | o-m | o-m | 2.29 | 0.0682 | 0.09 | 0.17 | 47 | n-c | n-c | 0.0682 | 0.11 | 0.02 | 45 | n-c | n-c | 0.0682 |
| *slot* | 5 | 1.86 | 0.10 | 49 | 0.9282 | 0.25 | 0.9284 | 0.02 | 0.01 | 25 | 0.01 | 25 | 0.9249 | 0.02 | 0.00 | 24 | 0.01 | 25 | 0.9249 |
| *slot* | 10 | o-m | o-m | o-m | o-m | 0.50 | 0.9326 | 0.05 | 0.10 | 64 | 0.13 | 63 | 0.9281 | 0.06 | 0.03 | 61 | 0.06 | 60 | 0.9281 |

Table 3. Results for our test models. Runtimes are in seconds; "o-m" means "needs more than 4GB of memory"; "n-c" means "did not converge within 10,000 global iterations"; "—" means "same as entry above"; "n/a" means "not applicable (not Kronecker–consistent)".

| | | | Kronecker | | EV*MDD | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | Disjunctive | | Single root | | Actual **R** | |
| $N$ | $\eta(\mathbf{R})$ | $L$ | matrs | nz | nodes | edges | nodes | edges | nodes | edges |
| 5 | $6.60\times10^{6}$ | 14 | 51 | 1587 | 1392 | 2942 | 1046 | 3342 | 39939 | 63587 |
| | | 22 | n/a | n/a | 924 | 1642 | 2374 | 4527 | 5481 | 8454 |
| 6 | $3.23\times10^{7}$ | 14 | 51 | 2528 | 2202 | 4693 | 1330 | 4767 | 70673 | 116604 |
| | | 22 | n/a | n/a | 1087 | 2055 | 2904 | 5780 | 7637 | 12202 |
| 10 | $4.25\times10^{9}$ | 14 | 51 | 9972 | 8642 | 18577 | 3164 | 15205 | 393995 | 715570 |
| | | 22 | n/a | n/a | 1969 | 4439 | 6071 | 13454 | 24615 | 43222 |
| 20 | $7.33\times10^{12}$ | 14 | 51 | 71142 | 61842 | 132947 | 14259 | 92260 | 4992281 | 10073732 |
| | | 22 | n/a | n/a | 5059 | 14224 | 18916 | 46649 | 144515 | 284812 |

Table 4
Data structure statistics for various encodings of **R** (model *fms*).

simulation and of our approximation is somewhat arbitrary, since either can be made very fast or very slow by appropriately changing the stopping criteria, but the conclusion remains that our approximation is quite accurate, within 10%, assuming the simulation results are reliable.

Comparing with our previous Kronecker–based approximation when possible, we observe that the initialization time for the new method is usually a little worse but the computation time is much better in our more general setting, because using a single–root representation instead of a disjunctive encoding saves numerous additions of rates during the computation and because the implementation has improved. When both methods can be run, they produce essentially the same numerical results. We also observe that the solution times for the approximations using Gauss-Seidel and Jacobi are different because (1) they require different number of *global* iterations and (2) in each global iteration, they require different numbers of *local* iterations to converge. Between these two methods, there is no clear winner. For the explicit solution, Gauss-Seidel requires instead lower runtimes than Jacobi for all models we tested, as one would expect, thus we only report the performance for Gauss-Seidel.

Finally, we note that, for the models we tested with different decompositions, the choice of decomposition does not significantly affect the computed measure (the maximum observed difference was 0.03%), which attests to the stability of the approach. However, we find that the use of a finer decomposition tends to produce a smaller initialization time $T_{init}$ and a smaller solution time $T_{solve}$. We discuss this in more detail in the next section.

| $N$ | $\lvert\mathcal{S}\rvert$ | $L$ | MDD nodes | $\sum_{k=1}^{L} \lvert\boldsymbol{\pi}_k\rvert$ | $\max_k \lvert\boldsymbol{\pi}_k\rvert$ | $\eta(\mathbf{A})$ | $\sum_{k=1}^{L} \eta(\mathbf{R}_k)$ | $\max_k \eta(\mathbf{R}_k)$ |
|---|---|---|---|---|---|---|---|---|
| 5 | $8.52\times10^5$ | 14 | 1669 | 4779 | 756 | 5712 | 26335 | 4590 |
| | | 22 | 236 | 431 | 52 | 6531 | 1007 | 133 |
| 6 | $3.84\times10^6$ | 14 | 2612 | 8414 | 1372 | 8979 | 48543 | 8673 |
| | | 22 | 301 | 579 | 74 | 9629 | 1433 | 202 |
| 10 | $4.14\times10^8$ | 14 | 9824 | 45694 | 7986 | 33937 | 291245 | 55055 |
| | | 22 | 631 | 1421 | 202 | 37424 | 4007 | 628 |
| 20 | $5.91\times10^{11}$ | 14 | 66634 | 547974 | 101871 | 229962 | 3795790 | 754110 |
| | | 22 | 1946 | 5626 | 1371 | 280144 | 17932 | 4848 |

Table 5
Aggregation statistics for single-root EV*MDD-based approximation (model *fms*).

### *5.3 Data structure comparison*

Table 4 reports the number of non–zero elements in the overall transition rate matrix, the number of matrices and non–zero elements in the Kronecker representation, and the number of nodes and non–zero edges in the EV*MDD encoding for the FMS model. These can be seen as the memory requirements for $\mathbf{R}$ in an explicit solution, a Kronecker–based approximation, and a EV*MDD-based approximation, respectively. For the EV*MDD encoding, we compare the disjunctive and the single–root representations for a potential $\mathbf{R}$ as discussed in Sect. 3.3 and 3.4; for comparison, the last two columns report the statistics for the EV*MDD encoding actual $\mathbf{R}$, as discussed in Sect. 4.3.

Table 5 reports detailed statistics for the data structures used in our EV*MDD approximation on the FMS model: size of $\mathcal{S}$, number of nodes in the MDD encoding $\mathcal{S}$, total ($\sum_{k=1}^{L} \lvert\boldsymbol{\pi}_k\rvert$) and maximum ($\max_k \lvert\boldsymbol{\pi}_k\rvert$) number of states in the aggregated CTMCs (relating to the computational cost for each global iteration), number of non–zeros in matrix $\mathbf{A}$ (relating to the cost of building the aggregated CTMCs), and total and maximum number of non–zeros for the transition rate matrices in the aggregated CTMCs. The number of edges in the MDD encoding $\mathcal{S}$ equals $\sum_{k=1}^{L} \lvert\boldsymbol{\pi}_k\rvert$. From this table, we can also see how the size of the involved matrices reduces sharply; for $N = 20$, the exact solution requires a matrix of size $5.91 \times 10^{11}$ while our approximation uses matrices of size at most around $10^3$ or $10^5$, depending on the chosen decomposition.

From Table 4, we can see that both Kronecker and EV*MDD encodings are quite compact, and much more efficient than explicit sparse storage of $\mathbf{R}$, which is proportional to $\eta(\mathbf{R})$. The disjunctive EV*MDD approach uses fewer than twice as many edges as the non–zeros for Kronecker matrices, as discussed in Sect. 3.4. The single–root EV*MDD encoding we adopt for this paper can use more or fewer non–zero edges than the disjunctive encoding

and the memory usage of the three (or the latter two when the Kronecker is not applicable) encodings has similar magnitude for a given decomposition. Interestingly, for this model, the single–root encoding tends to be more compact than the disjunctive encoding (due to greater node sharing) for the $L = 14$ decomposition, while the reverse is true for the $L = 22$ decomposition. Regardless of the decomposition chosen, the EV*MDD encoding the actual matrix $\mathbf{R}$ (i.e., removing fictitious entries) always uses more edges, at times by several orders of magnitude, than when encoding the potential $\mathbf{R}$ (e.g., for FMS with $N = 20$, $10^7$ versus $10^5$ edges).

Using a finer decomposition tends to produce fewer nodes and edges in the MDD encoding $\mathcal{S}$, and fewer edges in the EV*MDD encoding $\mathbf{R}$, especially as the model parameter $N$ increases. This, along with the fact that the finer decomposition results in smaller local state spaces (thereby reducing both state–space generation and transition–rate–matrix generation times), causes the decrease in initialization times $T_{init}$ as seen in Table 3 for the FMS model. For this model, the value of $\eta(\mathbf{A})$ increases, but only slightly, for the finer decomposition. More importantly for our approximation algorithm, a finer decomposition tends to produce smaller (both in terms of number of states, and number of non–zero entries) aggregate CTMCs. As such, the time for each global iteration tends to decrease with a finer partition, causing a decrease in the solution times $T_{solve}$ for both the Gauss-Seidel and Jacobi methods. For instance, looking at Table 3, we see that for model FMS with $N = 20$, the $L = 22$ decomposition requires an average of $18.03/217 \approx 0.083$ seconds per global iteration, while the $L = 14$ decomposition requires instead an average of $453.94/106 \approx 4.28$ seconds per global iteration.

### *5.4 Summary*

We can conclude that (1) our new method provides quite accurate results, with less than 10% relative error in our experiments; (2) when both Kronecker–based and EV*MDD–based approximation are applicable for the same decomposition, the EV*MDD-based method uses less time to produce the same results; (3) a finer decomposition can improve the time and memory requirements of our new method, without sacrificing accuracy; (4) our new method can always choose the "finest" decomposition, while the Kronecker–based approximation is instead restricted to the finest decomposition that still satisfies Kronecker consistency; (5) our EV*MDD encoding of the transition rate matrix provides a seamless and efficient construction from the transition relation MDDs built during state–space generation, which makes our methods eminently applicable when paired with a symbolic state–space generation such as that of [26]; finally, (6) symbolic generation of the state space and transition rate matrix also benefits from a finer decomposition.

## 6 Conclusion

We presented a new algorithm for the approximate numerical solution of structured ergodic CTMCs, based on an MDD representation of the state space and an EV*MDD representation of the transition rate matrix. The method retains the favorable properties of a previously proposed approximation algorithm, but avoids the need for a Kronecker–consistent model decomposition. Elimination of this restriction allows for the use of finer model decompositions, which can reduce the time and memory requirements for state space generation, transition rate matrix generation, and the approximation algorithm itself, without sacrificing the accuracy of the results. As such, our new algorithm is fully general and applicable to larger and more complex models.

## References


[1] M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with Generalized Stochastic Petri Nets*. John Wiley & Sons, New York, 1995.

[2] V. Amoia, G. De Micheli, and M. Santomauro. Computer-oriented formulation of transition-rate matrices via Kronecker algebra. *IEEE Trans. Rel.*, 30:123–132, June 1981.

[3] M. Bozga and O. Maler. On the representation of probabilities over structured domains. In N. Halbwachs and D. Peled, editors, *Proc. CAV*, LNCS 1633, pages 261–273, Trento, Italy, July 1999. Springer-Verlag.

[4] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, Aug. 1986.

[5] P. Buchholz, G. Ciardo, S. Donatelli, and P. Kemper. Complexity of memory-efficient Kronecker operations with applications to the solution of Markov models. *INFORMS J. Comp.*, 12(3):203–222, 2000.

[6] G. Ciardo, R. L. Jones, A. S. Miner, and R. Siminiceanu. Logical and stochastic modeling with SMART. *Perf. Eval.*, 63:578–608, 2006.

[7] G. Ciardo, G. Lüttgen, and A. S. Miner. Exploiting interleaving semantics in symbolic state-space generation. *Formal Methods in System Design*, 31:63–100, 2007.

[8] G. Ciardo, G. Lüttgen, and R. Siminiceanu. Saturation: An efficient iteration strategy for symbolic state space generation. In T. Margaria and W. Yi, editors, *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS 2031, pages 328–342, Genova, Italy, Apr. 2001. Springer-Verlag.

[9] G. Ciardo, R. Marmorstein, and R. Siminiceanu. Saturation unbound. In H. Garavel and J. Hatcliff, editors, *Proc. Tools and Algorithms for*

*the Construction and Analysis of Systems (TACAS)*, LNCS 2619, pages 379–393, Warsaw, Poland, Apr. 2003. Springer-Verlag.

[10] G. Ciardo, R. Marmorstein, and R. Siminiceanu. The saturation algorithm for symbolic state space exploration. *Software Tools for Technology Transfer*, 8(1):4–25, Feb. 2006.

[11] G. Ciardo and R. Siminiceanu. Using edge-valued decision diagrams for symbolic generation of shortest paths. In M. D. Aagaard and J. W. O'Leary, editors, *Proc. Fourth International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, LNCS 2517, pages 256–273, Portland, OR, USA, Nov. 2002. Springer-Verlag.

[12] G. Ciardo and K. S. Trivedi. A decomposition approach for stochastic reward net models. *Perf. Eval.*, 18(1):37–59, 1993.

[13] G. Ciardo and A. J. Yu. Saturation-based symbolic reachability analysis using conjunctive and disjunctive partitioning. In D. Borrione and W. Paul, editors, *Proc. CHARME*, LNCS 3725, pages 146–161, Saarbrücken, Germany, Oct. 2005. Springer-Verlag.

[14] E. Clarke, M. Fujita, P. C. McGeer, J. C.-Y. Yang, and X. Zhao. Multi-terminal binary decision diagrams: an efficient data structure for matrix representation. In *IWLS '93 International Workshop on Logic Synthesis*, May 1993.

[15] P. Fernandes, B. Plateau, and W. J. Stewart. Efficient descriptor-vector multiplication in stochastic automata networks. *J. ACM*, 45(3):381–414, 1998.

[16] T. Kam, T. Villa, R. K. Brayton, and A. Sangiovanni-Vincentelli. Multi-valued decision diagrams: theory and applications. *Multiple-Valued Logic*, 4(1–2):9–62, 1998.

[17] A. S. Miner. Efficient solution of GSPNs using canonical matrix diagrams. In R. German and B. Haverkort, editors, *Proc. 9th Int. Workshop on Petri Nets and Performance Models (PNPM'01)*, pages 101–110, Aachen, Germany, Sept. 2001. IEEE Comp. Soc. Press.

[18] A. S. Miner. Implicit GSPN reachability set generation using decision diagrams. *Perf. Eval.*, 56(1-4):145–165, Mar. 2004.

[19] A. S. Miner. Saturation for a general class of models. *IEEE Trans. Softw. Eng.*, 32(8):559–570, Aug. 2006.

[20] A. S. Miner, G. Ciardo, and S. Donatelli. Using the exact state space of a Markov model to compute approximate stationary measures. In J. Kurose and P. Nain, editors, *Proc. ACM SIGMETRICS*, pages 207–216, Santa Clara, CA, USA, June 2000. ACM Press.

[21] E. Pastor, O. Roig, J. Cortadella, and R. M. Badia. Petri net analysis using boolean manipulation. In R. Valette, editor, *Proc. International Conference on Applications and Theory of Petri Nets (ICATPN)*, LNCS 815, pages 416–435, Zaragoza, Spain, June 1994. Springer-Verlag.

[22] M. Solé and E. Pastor. Traversal techniques for concurrent systems. In M. D. Aagaard and J. W. O'Leary, editors, *Proc. FMCAD*, LNCS 2517, pages 220–237, Portland, OR, USA, Nov. 2002. Springer-Verlag.

[23] A. Srinivasan, T. Kam, S. Malik, and R. K. Brayton. Algorithms for discrete function manipulation. In *Int. Conference on CAD*, pages 92–95. IEEE Comp. Soc. Press, 1990.

[24] W. J. Stewart. *Introduction to the Numerical Solution of Markov Chains*. Princeton University Press, 1994.

[25] M. Tilgner, Y. Takahashi, and G. Ciardo. SNS 1.0: Synchronized Network Solver. In *1st Int. Workshop on Manufacturing and Petri Nets*, pages 215–234, Osaka, Japan, June 1996.

[26] M. Wan and G. Ciardo. Symbolic state-space generation of asynchronous systems using extensible decision diagrams. In M. Nielsen et al., editors, *Proc. 35th Int. Conf. Current Trends in Theory and Practice of Computer Science (SOFSEM)*, LNCS 5404, pages 582–594, Špindlerův Mlýn, Czech Republic, Feb. 2009. Springer-Verlag.