# STRUCTURAL APPROACHES FOR SPN ANALYSIS

Gianfranco Ciardo* Andrew S. Miner†
Department of Computer Science
College of William and Mary
{ciardo, asminer}@cs.wm.edu

**KEYWORDS** Kronecker algebra, structured models, decision diagrams, performance analysis, Markov chains.

## Abstract

Petri nets and Markovian Petri nets are excellent tools for logic and performability system modeling. However, the size of the underlying reachability set is a major limitation in practice. One approach gaining attention among researchers is the use of structured representations, which require us to decompose a net into, or compose a net from, subnets. This paper surveys the state-of-the-art in advanced techniques for storing the reachability set and the transition rate matrix, with particular attention to the use of decision diagrams, Kronecker representations, and their interplay. The conclusion is that, in most practical applications, it is now possible to generate and store enormous reachability sets for logical analysis, while the size of the probability vector being sought is the main limitation when performing the exact solution of the underlying Markov chain.

## 1 Introduction

Stochastic Petri nets (SPNs) are an increasingly popular formalism for describing and analyzing systems. This is due to the ability of SPNs to capture complex system behavior in a concise way, while still allowing for precise reliability and performance computations. An SPN is usually depicted graphically, and, under appropriate timing and probabilistic assumptions, its underlying stochastic process is a continuous-time Markov chain (CTMC) and can be the subject of various types of analysis. The primary difficulty of using SPNs to model large systems is the well-known state explosion problem: the number of states in the underlying stochastic process of an SPN can be extremely large, even for simple models.

Techniques for dealing with the state explosion problem can typically be classified along three main lines. First, there are techniques that exploit special characteristics of the SPN to allow for an efficient solution. An excellent example of this kind of technique is that of product-form SPNs [35]. Unfortunately, these techniques apply only to a restricted subset of SPNs. Approximation techniques form another group of approaches [3, 11, 18, 28]. These can be quite effective, as they typically require a small fraction of the memory and CPU time needed by exact techniques, at a cost of an approximation error in the results. The two above approaches are not mutually exclusive: applying a product-form technique to a non-product-form SPN may yield a close approximation to the exact results [34].

The techniques we consider in this paper fall instead under the broad category of "state space tolerance". These use data structures and algorithms that attempt to tolerate the large number of states of the SPN, and can be applied to a wide range of models to yield exact results. The main difficulty with state space tolerance is that of large memory requirements, as exact numerical analysis requires the representation of three large objects: $\mathcal{S}$, the set of reachable states; $\mathbf{R}$, the transition rate matrix of the underlying CTMC; and $\boldsymbol{\pi}$, the stationary probability vector to be computed (for brevity, we focus on steady-state behavior of ergodic models). Each of these could easily require an amount of storage in excess of the total memory available (including RAM, disk space, and distributed shared memory). Clearly, memory is of primary concern, as it limits the size of problem that can be studied. However, memory-efficient techniques with excessive CPU requirements are of little use. Thus we are faced with the challenge of finding suitable representations for $\mathcal{S}$, $\mathbf{R}$, and $\boldsymbol{\pi}$ that are both memory and CPU efficient.

In this paper, we provide a survey of several techniques for representing these structures efficiently. After defin-

*G. Ciardo was supported by NASA grant NAG-1-2168) and by a Semester Research Assignment from the College of William and Mary

†A. Miner was supported by fellowships from the NASA Graduate Student Research Program (NGT-1-52195) and the Virginia Space Grant Consortium

ing the essential terminology and notation for SPNs and their underlying reachability set $\mathcal{S}$ and transition rate matrix $\mathbf{R}$ (Sect. 2), we discuss several techniques for the storage of $\mathcal{S}$ and $\mathbf{R}$ (Sect. 3 and 4). Then, we discuss how the choices for the storage of $\mathcal{S}$ and $\mathbf{R}$ influence the choice and efficiency of the solution algorithm (Sec. 5). We conclude with some remarks about the current state-of-the-art and directions for future research (Sec. 6).

# 2 Background

We consider a general definition of SPNs, which includes inhibitor arcs [2], transition guards, and marking-dependent arc cardinalities [14] and transition rates. Formally, a Petri net is a finite, directed, bipartite graph containing the following.

- A set of places $\mathcal{P} = \{p_1, \ldots, p_{|\mathcal{P}|}\}$. A marking $\mathbf{m} \in \mathbb{N}^{|\mathcal{P}|}$ assigns a number of tokens to each place.
- A set of transitions $\mathcal{T} = \{t_1, \ldots, t_{|\mathcal{T}|}\}, \mathcal{P} \cap \mathcal{T} = \emptyset$.
- An initial marking $\mathbf{m}^{[0]} \in \mathbb{N}^{|\mathcal{P}|}$.
- Functions $I : \mathcal{P} \times \mathcal{T} \times \mathbb{N}^{|\mathcal{P}|} \to \mathbb{N}$, $O : \mathcal{T} \times \mathcal{P} \times \mathbb{N}^{|\mathcal{P}|} \to \mathbb{N}$, and $H : \mathcal{P} \times \mathcal{T} \times \mathbb{N}^{|\mathcal{P}|} \to \mathbb{N} \cup \{\infty\}$ to describe the marking-dependent cardinalities of the input, output, and inhibitor arcs.
- Function $g : \mathcal{T} \times \mathbb{N}^{|\mathcal{P}|} \to \{0, 1\}$ to describe the transition guards.
- Function $\lambda : \mathcal{T} \times \mathbb{N}^{|\mathcal{P}|} \to \mathbb{R}$ to describe the marking-dependent transition rates[1].

A transition $t$ is said to be *enabled* in marking $\mathbf{m}$, if

$$g(t, \mathbf{m}) = 1 \text{ and } \forall p \in \mathcal{P},\ I(p, t, \mathbf{m}) \leq \mathbf{m}_p < H(p, t, \mathbf{m})$$

where $\mathbf{m}_p$ denotes the number of tokens in place $p$ in marking $\mathbf{m}$. An enabled transition may *fire*, leading to a new marking $\mathbf{n}$, written $\mathbf{m} \xrightarrow{t} \mathbf{n}$, given by

$$\forall p \in \mathcal{P}, \quad \mathbf{n}_p = \mathbf{m}_p - I(p, t, \mathbf{m}) + O(t, p, \mathbf{m}).$$

The reachability set $\mathcal{S}$ is then defined as the smallest set containing $\mathbf{m}^{[0]}$ and such that, if $\mathbf{m} \in \mathcal{S}$ and $\mathbf{m} \xrightarrow{t} \mathbf{n}$, then $\mathbf{n} \in \mathcal{S}$ as well. We also define a bijection $\Psi : \mathcal{S} \to \{0, \ldots, |\mathcal{S}| - 1\}$ to index each reachable marking. The *transition rate* matrix $\mathbf{R} \in \mathbb{R}^{|\mathcal{S}|\times|\mathcal{S}|}$ of the underlying CTMC of the SPN has elements defined by

$$\mathbf{R}[\Psi(\mathbf{m}), \Psi(\mathbf{n})] = \sum_{\forall t : \mathbf{m} \xrightarrow{t} \mathbf{n}} \lambda(t, \mathbf{m})$$

for all reachable markings $\mathbf{m}, \mathbf{n} \in \mathcal{S}$, $\mathbf{m} \neq \mathbf{n}$. The *infinitesimal generator* matrix $\mathbf{Q}$ is defined by

$$\mathbf{Q} = \mathbf{R} - \mathit{Diag}(\mathbf{h})^{-1} = \mathbf{R} - \mathit{Diag}(\mathbf{R} \cdot \mathbf{1})$$

where $\mathbf{R} \cdot \mathbf{1}$ is a vector whose elements are the row sums of $\mathbf{R}$ and $\mathit{Diag}(\mathbf{x})$ is a matrix with $\mathbf{x}$ on the diagonal and zeroes elsewhere. Thus, $\mathbf{Q}$ is identical to $\mathbf{R}$ except on the diagonal, which is set so that the rows of $\mathbf{Q}$ sum to zero. The entries of vector $\mathbf{h}$ are the inverse of the diagonal of $\mathbf{Q}$ and represent the expected *holding times* in each state; it is customary to store $\mathbf{h}$ instead of the diagonal entries of $\mathbf{Q}$, since these would be used only as divisors in many numerical solution algorithms. Finally, $\boldsymbol{\pi} \in \mathbb{R}^{|\mathcal{S}|}$ is the stationary probability vector satisfying

$$\boldsymbol{\pi} \cdot \mathbf{Q} = \mathbf{0}.$$

Given a Petri net, we can partition its set of places into $K$ subsets $\mathcal{P} = \mathcal{P}_K \cup \cdots \cup \mathcal{P}_1$, effectively defining $K$ subnets. Then, a marking $\mathbf{m}$ can be partitioned as $(\mathbf{m}_K, \ldots, \mathbf{m}_1)$, and we can define the set of reachable "local" markings for each subnet as

$$\mathcal{S}_k = \{\mathbf{m}_k : \exists \mathbf{n} \in \mathcal{S}, \mathbf{n}_k = \mathbf{m}_k\}.$$

The set of *potential* markings is

$$\hat{\mathcal{S}} = \mathcal{S}_K \times \cdots \times \mathcal{S}_1,$$

which is guaranteed to include all reachable markings. As with $\mathcal{S}$, we define a bijection $\hat{\Psi} : \hat{\mathcal{S}} \to \{0, \ldots, |\hat{\mathcal{S}}| - 1\}$ to index each potential marking:

$$\hat{\Psi}(\mathbf{m}_K, \ldots, \mathbf{m}_1) = \sum_{k=1}^{K} \left( \Psi_k(\mathbf{m}_k) \cdot \prod_{j=1}^{k-1} |\mathcal{S}_j| \right)$$

where $\Psi_k$ indexes the local markings $\mathcal{S}_k$. This is equivalent to lexicographical ordering of the markings based on the submarking indices.

We say a transition $t$ *affects* subnet $k$ if its firing affects or is affected by places in $\mathcal{P}_k$ (i.e., $t$ has input, output or inhibitor arcs connected to a place in $\mathcal{P}_k$, or has a marking-dependent arc cardinality or guard that depends on a place in $\mathcal{P}_k$). The set of transitions that affect subnet $k$ is denoted $\mathcal{T}_k$. If a transition affects more than one subnet, it is said to be *synchronizing*, otherwise we say it is *local* to subnet $k$. The set of synchronizing transitions is $\mathcal{T}_S = \{t \in \mathcal{T} : \exists k, l, k \neq l \wedge t \in \mathcal{T}_k \cap \mathcal{T}_l\}$. The sets $\mathcal{T}_K, \ldots, \mathcal{T}_1$ and $\mathcal{T}_S$ can be easily determined from the SPN definition.

[1] For brevity, we focus on the case of exponentially distributed firing times, so that a marking of the untimed Petri net corresponds to a state of the underlying process. However, immediate transitions that fire in zero time and phase-type distributions that approximate general timing behavior can be used in practice with any of the approaches we describe.

# 3 State space storage

In some applications such as software or hardware verification, analysis of the reachable markings is the primary concern. For our case, the reachable markings correspond to the states of the underlying CTMC. Thus, the first step in the analysis of an SPN is often the construction of the set of reachable markings $\mathcal{S}$. Indeed, the discussion in this section can be made assuming an untimed Petri net.

Several data structures have been proposed for storing $\mathcal{S}$. The algorithm used to generate $\mathcal{S}$ may depend on our choice of data structure. Once $\mathcal{S}$ has been built, we may wish to move it into another data structure for long-term storage, either due to different access requirements after generation or to conserve memory. After generation, the requirements of our data structure for $\mathcal{S}$ will depend on the technique we use for CTMC representation. If we use straightforward sparse storage for $\mathbf{R}$, we require a data structure for $\mathcal{S}$ that will allow us to perform two operations efficiently:

1. Given a potential marking $\mathbf{m}$, determine either that it is unreachable ($\mathbf{m} \notin \mathcal{S}$) or its index $\Psi(\mathbf{m})$ in $\mathcal{S}$, if $\mathbf{m}$ is reachable.

2. Given an index $i \in \{0 \dots |\mathcal{S}| - 1\}$, determine the reachable marking with index $i$ in $\mathcal{S}$, $\Psi^{-1}(i)$.

The first operation is used to build $\mathbf{R}$. The second operation is used to enumerate the markings for computation of rewards once $\boldsymbol{\pi}$ has been computed.

In the remainder of this section, we consider data structures and algorithms for generation and storage of $\mathcal{S}$. For each, we discuss the types of models the technique can handle properly, the data structure used during generation, the generation algorithm, and the final data structure used once the set $\mathcal{S}$ is known.

## 3.1 Traditional approach

The most general algorithm we consider for generating the set $\mathcal{S}$ is a straightforward breadth-first search. This technique can be applied to any Petri net with a finite reachability set. Algorithm 1 can be used to generate $\mathcal{S}$. If the set $\mathcal{U}$ is implemented as a queue (i.e., in line 4, we pick the marking that has remained unexplored for the longest amount of time), then Algorithm 1 follows a breadth-first search. If the set $\mathcal{U}$ is implemented as a stack, then it follows a depth-first search. The set $\mathcal{R}$ holds the markings discovered so far, and will be equivalent to $\mathcal{S}$ when the algorithm terminates. Since we add a finite number of markings to $\mathcal{R}$ during each iteration, the algorithm will not terminate if $\mathcal{S}$ is infinite.

```
Generate(Petri net M, Marking m^0)
 1: U ← {m^0}                          • Unexplored markings
 2: R ← {m^0}                          • Reachable markings
 3: while U ≠ ∅ do
 4:   Pick an unexplored marking m ∈ U
 5:   U ← U \ {m}
 6:   for each transition t ∈ T do
 7:     if t is enabled in m then
 8:       Compute n such that m -t-> n
 9:       if n ∉ R then
10:         R ← R ∪ {n}
11:         U ← U ∪ {n}
12: return R
```

Algorithm 1: Traditional reachability set generation.

To use Algorithm 1, we need an efficient data structure for representing sets of markings, and also a compact representation for a single marking. Conceptually, we can store a single marking as an array of integers, representing a marking of the Petri net. If the Petri net contains many places, it may be beneficial to store the markings in sparse format, where only the places that contain tokens are represented. The storage requirements of a single marking can be further reduced by exploiting place invariants [12]. From now on, we ignore the specifics of storing a single marking, and assume that it simply requires some number of bits.

Looking at Algorithm 1, we see that our data structure for $\mathcal{R}$ must be able to efficiently perform insertions (line 10) and searches (line 9), and our data structure for $\mathcal{U}$ must be able to efficiently perform insertions (line 11) and deletions (line 5). Since $\mathcal{U}$ is always a subset of $\mathcal{R}$, we can use a linked list or other simple structure to designate the portion of $\mathcal{R}$ that corresponds to $\mathcal{U}$. A thorough discussion of techniques for representing $\mathcal{U}$ can be found in [15]. For the rest of the paper, we only consider data structures to represent $\mathcal{R}$.

A possible data structure for $\mathcal{R}$ is a hashing table [22]. With it, we apply a hashing function $h : \hat{\mathcal{S}} \to \{0 \dots M - 1\}$ to each marking to determine its location in an $M$-element table. To use a hash table, we must choose $h$ carefully. If the hash table has fixed size, the size of $M$ is critical. In practice, dynamic hashing tables should be used to allow the table to grow over time.

Other commonly used structures for $\mathcal{R}$ are binary search trees [15]. Unlike hash tables, binary trees always require memory proportional to the number of markings. To use a binary tree, the markings must be ordered according to some total order; lexicographic order is normally used. To avoid the linear worst-case behavior of ordinary binary search trees, some kind of balancing

strategy must be used. Examples are *AVL* [1] and *splay* [36] trees, which achieve balance through rotations.

After $\mathcal{S}$ has been generated using either a hash table or a binary tree, we can copy the markings into an array sorted according to the total order. Then, to determine whether a given marking is reachable, we can simply perform a binary search. The position of the marking in the array, if found, gives us the index of the marking. Once the sorted array has been built, we can destroy the hash table or binary tree. This representation of $\mathcal{S}$ requires $|\mathcal{S}| \cdot b_s$ memory, where $b_s$ is the average number of bits required to encode each marking.

## 3.2 BDDs

A binary decision diagram (BDD) [26] is a directed, acyclic graph used to represent a boolean function $f : \{0,1\}^n \rightarrow \{0,1\}$. The graph consists of *terminal* nodes and *non-terminal* nodes. The terminal nodes represent the constant functions 0 and 1, and are labeled accordingly. Each non-terminal node represents some logic function $f$, is labeled with a variable $x$, and contains exactly two outgoing arcs, labeled 0 and 1, that point to the cofactors $f_{x=0}$ and $f_{x=1}$, respectively[2].

An *ordered* BDD (OBDD) has a total ordering on the variables such that any path of the graph visits variables according to the total order. A *reduced* OBDD (ROBDD) is of minimal size (i.e., it does not contain two distinct nodes representing the same function). ROBDDs are a canonical representation: given a variable ordering, two logic functions $f$ and $g$ are equivalent iff $f$ and $g$ are represented by the same ROBDD. Bryant [5, 6] showed how ROBDDs can be efficiently manipulated. In the following we use "BDD" to mean "ROBDD".

Pastor et al. [29, 30, 31] use BDDs for the generation and storage of the reachability set of a *safe* Petri net, where each place can contain at most one token. In [31], each place is considered to be a boolean variable, and so a set of markings $\mathcal{S}$ is equivalent to a logic function $f$, where $f(\mathbf{m}_{|\mathcal{P}|}, \ldots, \mathbf{m}_1) = 1$ iff $\mathbf{m} \in \mathcal{S}$. Encodings more sophisticated than one place per variable are discussed in [29, 30]. The main result in [31] is that, given a BDD encoding a set of markings $\mathcal{X}$, we can compute the BDD encoding the set of markings $\Delta(\mathcal{X})$ reachable from $\mathcal{X}$ in one transition firing, where the operator $\Delta$ is easily built based on the Petri net definition. Algorithm 2 illustrates the idea (the algorithm we show is a simplified version of the one in [31]), where the sets $\mathcal{O}$, $\mathcal{R}$, and $\Delta(\mathcal{R})$ are all encoded as BDDs. The number of iterations performed by Algorithm 2 is bounded by the *sequential depth* of the Petri net (i.e., the maximum number of transition firings required to reach a marking from the initial marking). Thus, while each iteration usually implies a substantial computation, few iterations are usually required.

```
Generate(Safe Petri net M, Marking m^[0])
1: O ← ∅                       • O : old reachability set
2: R ← {m^[0]}                 • R : reachable markings so far
3: while R ≠ O do
4:     O ← R
5:     R ← Δ(R) ∪ R
6: return R
```

Algorithm 2: A BDD-based procedure to generate $\mathcal{S}$.

To search for a marking $\mathbf{m}$ in our BDD encoding of $\mathcal{S}$, we traverse the BDD starting at the node representing $\mathcal{S}$. When we encounter a node labeled with variable $x_k$, we follow the 1 arc if place $p_k$ contains a token in marking $\mathbf{m}$, otherwise we follow the 0 arc. This continues until we reach terminal node 1 ($\mathbf{m} \in \mathcal{S}$) or 0 ($\mathbf{m} \notin \mathcal{S}$). We discuss the technique for determining the index of a marking stored in a BDD in Sect. 3.6.

If the Petri net is not safe, that is, if place $p_k$ can contain up to $c > 1$ tokens, two encodings have been proposed in [31]: we either use a "one-hot" encoding requiring $c$ boolean variables, at most one of which can be set to 1 (they are all zero if $p_k$ is empty), or a standard binary encoding using $\log(c+1)$ variables. It should be noted that the binary encoding is not necessarily the best choice, even if it uses fewer variables.

## 3.3 Bit vectors

Another way to represent the set $\mathcal{S}$ for structured nets is to specify which markings in $\hat{\mathcal{S}}$ are reachable. That is, we use a bit vector [25] of size $|\hat{\mathcal{S}}|$, where the bit in position $\hat{\Psi}(\mathbf{m})$ is 1 iff $\mathbf{m} \in \mathcal{S}$. This approach is applicable only if we can build the local reachability sets (or reasonable supersets of them) *a priori*. To generate $\mathcal{S}$, we use an algorithm similar to Algorithm 1, with the unexplored markings stored in a linked list. This requires exactly $|\hat{\mathcal{S}}|$ bits to store $\mathcal{S}$, plus $O(U \cdot \log |\hat{\mathcal{S}}|)$ memory during exploration, where $U$ is the maximum size of the set of unexplored markings $\mathcal{U}$.

Once the set $\mathcal{S}$ is known, we generate an array of markings, where each marking is represented as an integer in the range $\{0 \ldots |\hat{\mathcal{S}}| - 1\}$, which is the potential index of the marking; then, we can destroy the bit vector. This potential index may exceed the machine integer capabilities. For instance, this happens on a machine with 32-bit integers, if $|\hat{\mathcal{S}}| > 2^{32}$. The array of potential indices requires $O(|\mathcal{S}| \cdot \log |\hat{\mathcal{S}}|)$ bits of storage. This final

[2] A cofactor of a function $f$ with respect to a variable $x_i$ is $f_{x_i=c} = f(x_n, \ldots, x_{i+1}, c, x_{i-1}, \ldots, x_1)$.

representation of $\mathcal{S}$ is essentially the same as that described at the end of Sect. 3.1.

### 3.4 Folded bit vector

The bit vector approach can be improved based on the following observation [7]. Consider arranging the bit vector of size $|\hat{\mathcal{S}}|$ representing $\mathcal{S}$ into a bit matrix with $|\mathcal{S}_K|$ rows, where row $i$ corresponds to the reachable markings when subnet $K$ is in local marking $i$ (i.e., $i$ is the marking of the places in $\mathcal{P}_K$). If row $i$ is identical to row $j$, the "environment" of local marking $i$ in subnet $K$ is identical to the environment of local marking $j$. That is, the local markings that the other subnets can reach when subnet $K$ is in local marking $i$ is the same as when it is known to be in local marking $j$. In [7], Buchholz uses this property to merge local markings into "macro-markings", where (conceptually) we only need to store row $i$, not both rows. This technique is repeated for each subnet, not just subnet $K$. The result is a hierarchical structuring of $\mathcal{S}$, which has macro-markings at the top level and actual markings at the bottom level. The storage of the probability vector and the indexing of reachable markings follow the same idea.

Furthermore, [7] describes a technique to determine some of the macro-markings in a preprocessing step to reduce the memory requirements of $\mathcal{S}$ during generation.

### 3.5 Multi-level search structures

Another approach for representing the reachability set of a structured Petri net is to use a multi-level structure [12, 15]. At the top level of a $K$-level structure, we store each of the reachable local markings $\mathbf{m}_K$ of subnet $K$, along with a downward pointer to a $(K-1)$-level structure that represents the possible local markings reachable in the other $K-1$ subnets when subnet $K$ is in local marking $\mathbf{m}_K$. If the local reachability sets can be generated *a priori*, we can use downward pointer arrays of size $|\mathcal{S}_k|$ at level $k > 1$, and bit arrays of size $|\mathcal{S}_1|$ at level 1. For an unreachable local marking we store either a null pointer or a 0 bit at the appropriate position, otherwise we store a pointer to the next level or a 1 bit. An example of this data structure is shown in Fig. 1, where we have a Petri net composed of three subnets with local reachability sets $\mathcal{S}_3 = \{B, C, D\}$, $\mathcal{S}_2 = \{A, E, I\}$, and $\mathcal{S}_1 = \{L, N\}$. For clarity, null pointers and zero bits are omitted from the figure (the meaning of the integers immediately below the local markings will be explained later).

If instead the local reachability sets cannot be generated *a priori*, we can use trees instead of arrays to store the reachable local markings, resulting in a multi-level tree structure. Each node in the tree stores the local marking, pointers to the left and right children, and, for levels above 1, a downward pointer. Once $\mathcal{S}$ has been generated, the trees in a multi-level structure can be replaced with arrays (Fig. 2), thus eliminating the left and right pointers. Dynamic arrays could be used instead of trees during generation. This requires to store size information along with the array; an attempt to read past the end of an array returns a null pointer or 0 bit, while an attempt to write past the end of an array requires us to enlarge the array.

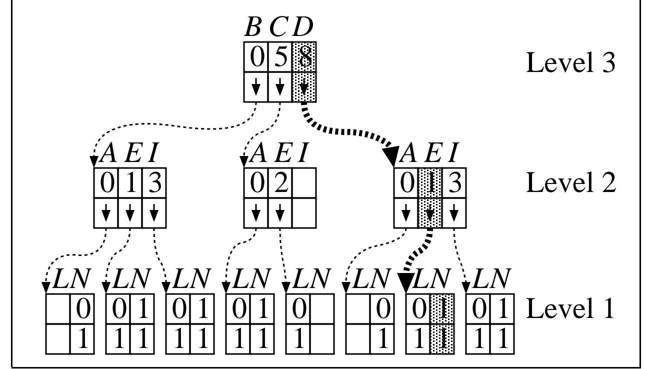


Figure 1: A multi-level structure.

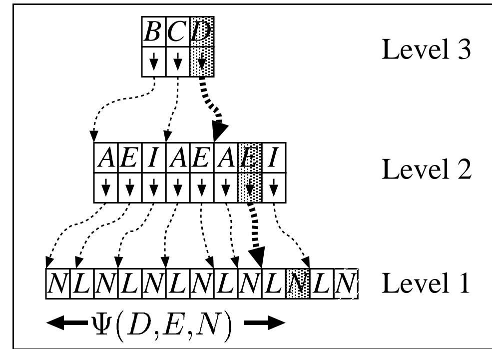


Figure 2: A compacted multi-level structure.

To search for a marking $(\mathbf{m}_K, \ldots, \mathbf{m}_1)$ in our multi-level structure, we first search for $\mathbf{m}_K$ in level $K$ array or tree. If found (and the downward pointer is not null), we follow the downward pointer to a level $K-1$ array or tree. This continues until either we fail to find a non-null pointer for $\mathbf{m}_k$ in the appropriate level $k$ array or tree, in which case the marking is not reachable, or we find $\mathbf{m}_1$ (with its bit set) in the appropriate level 1 array or tree, in which case the marking is reachable. For example, the highlighted path in Fig. 1 or 2 corresponds to the reachable marking $(D, E, N)$. Inserting a new marking is similar to searching, except that we must create a new array or tree whenever we encounter a null pointer.

An important result from [15] is the concept of the *locality* of a transition: it is possible to determine from the Petri net definition which local markings are affected by a transition. In particular, the firing of transition $t$ does not change submarking $\mathbf{m}_k$ if $t \notin \mathcal{T}_k$. Using this information, we can start our searches and insertions at levels other than $K$ in our multi-level structure. This can save substantial computation, especially if trees are used and $K$ is large. In our example, a search for marking $(D, E, L)$, immediately after having found marking $(D, E, N)$, can begin at level 1 instead of level 3.

Once $\mathcal{S}$ has been explored, there are two ways to compute the index of a marking, depending on whether sparse or full arrays are used to store $\mathcal{S}$. In the case of sparse arrays, only the non-null pointers are stored in the intermediate levels (along with indexing information), and the indices of the set bits are stored at level 1. Since the number of set bits at level 1 is exactly equal to the number of reachable markings, the level 1 indices are stored in an array of size $|\mathcal{S}|$, whose elements are indices of local markings for level 1 (i.e., they require $\lceil \log |\mathcal{S}_1| \rceil$ bits each). When we search for a given marking $\mathbf{m}$, the position of $\mathbf{m}_1$ in the level 1 array gives us the index of $\mathbf{m}$ in $\mathcal{S}$, see Fig. 2.

In the case of full arrays, we must instead associate an integer index to each entry of the level 1 boolean vectors: if the bit is set, the associate integer will give us the index of the marking. However, this requires $|\mathcal{S}| \cdot \lceil \log |\mathcal{S}| \rceil$ bits in the best case, when all bits at level 1 are set, and even more if the many of those bits are not set. We can somewhat improve the memory requirements by storing partial index information with each reachable local marking in each node of the multilevel structure. The integers immediately below the local markings in Fig. 1 are used for this purpose. For example, the path for marking $(D, E, N)$ contains the integers $(8, 1, 1)$, indicating that $8 + 1 + 1 = 10$ reachable markings precede $(D, E, N)$ in lexicographic order, i.e., $(D, E, N)$ is the eleventh reachable marking. This reduces the memory for the last level from $\lceil \log |\mathcal{S}| \rceil$ to $\lceil \log |\mathcal{S}_1| \rceil$ bits per state.

The main difference between the approaches in Figs. 1 and 2 is a memory-time tradeoff. With full arrays, each local marking at each level can be found immediately, so the index computation requires $O(K)$ operations. With sparse arrays, a search is required at each level, so the index computation requires $O(\log |\hat{\mathcal{S}}|)$ operations, or $O(\log |\mathcal{S}|)$ in the best case, when the search structure is perfectly balanced. However, the sparse array uses less memory, possibly much less, if many pointers are null and many bits are unset at level 1.

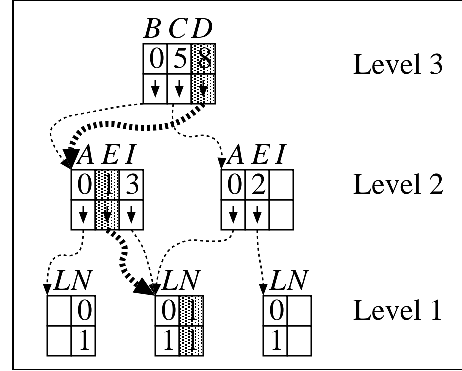


Figure 3: A merged multi-level structure.

### 3.6 MDDs

In the multi-level example of Fig. 1, many of the level 1 arrays are identical. Instead of storing duplicates, we can merge them into a single copy. This process can be repeated in a bottom-up fashion to give us the merged multi-level structure of Fig. 3.

As for the folded bit vector technique, we can perform this merging after generating $\mathcal{S}$. If we instead perform the merging as often as possible, during generation of $\mathcal{S}$, then our approach is similar to the BDD method, but is based on *multi-valued* decision diagrams (MDDs) [37]. This is the idea behind the approach presented in [27]. Maintaining a reduced MDD is both memory- and time-efficient, because large sets of markings can be represented and manipulated in a compact way.

In [27], a technique is presented for generating $\mathcal{S}$ by manipulating MDDs for Petri nets that have been partitioned so that the following properties hold:

**Local dependency for the arc cardinalities.** The cardinality of any input, output, or inhibitor arc connected to a place $p \in \mathcal{P}_k$ can depend only on the local marking of $\mathcal{P}_k$.

**Product form for the transition guards.** It must be possible to express the guard of a transition $t$ as the product of $K$ local guards: $g(t, \mathbf{m}) = 1$ iff $g_K(t, \mathbf{m}_K) = \cdots = g_1(t, \mathbf{m}_1) = 1$.

These properties are required so that the manipulations to determine the enabling of a transition and the new markings reached by firing a transition can be done by checking each level of the MDD independently.

In particular, the enabling of a transition local to subnet $k$ can be determined by checking a single level only. [27] exploits this property to devise an efficient technique for generating markings reached when local transitions fire, where the enabling, firing, and set union

computations can be combined into a single operation. Synchronizing transitions instead require manipulating MDD nodes at more than one level, and distinct operations are needed to determine the set of markings that enable transition $t$, to compute the set of markings reached after $t$ fires, and to add those markings to $\mathcal{S}$. However, even in the case of synchronizing transitions, the idea of locality can be used to improve performance. For example, [27] uses the knowledge that a transition does not affect levels $K$ through $k+1$ to speed up computation when visiting the nodes in those levels.

After $\mathcal{S}$ has been generated, we can search for markings and determine their indices in the same way as the multi-level structure when full arrays are used [16] (as shown in Fig. 3, exactly the same partial index information as in Fig. 1 is used).

BDDs are special cases of MDDs where the number of possible values at every level is exactly two. Indeed, the MDD technique of [27], when applied to a safe Petri net partitioned such that each subnet contains a single place, computes the same BDD as the BDD technique described in [31]. Thus, the indexing technique used for MDDs shown in Fig. 3 can also be used for BDDs.

The MDD just defined can be implemented using, for each node, a full array (assuming we know the sizes of the local reachability set $\mathcal{S}_k$ beforehand) or a dynamic array or search tree, with exactly the same memory-time tradeoffs discussed in the previous section. As MDDs save much memory by not having duplicate nodes, however, it is often the case that using full arrays is the best option: the overall memory requirements are still small anyway, and the time savings can be substantial.

# 4 Transition matrix storage

In this section we discuss techniques for representing the transition rate matrix $\mathbf{R}$ of the underlying CTMC.

## 4.1 Traditional method

The most straightforward approach is to store the matrix $\mathbf{R}$ explicitly, in full or sparse storage. Full storage requires $O(|\mathcal{S}|^2)$ memory; hence, it is rarely used, especially because $\mathbf{R}$ is usually extremely sparse, i.e., most of its entries are zero. Sparse techniques [32] store each row or column of $\mathbf{R}$ as a linked list of nonzero entries. If we store $\mathbf{R}$ by columns, then we have $|\mathcal{S}|$ linked lists, where each element of list $j$ stores a row index $i$ and the value of the nonzero entry in position $(i, j)$. If the number of nonzero elements $\eta(\mathbf{R})$ is known *a priori*, we can instead use an array for the nonzero entries, and store for each column the index of the first nonzero entry. Both approaches require only $O(\eta(\mathbf{R}))$ memory to represent $\mathbf{R}$. Another important reason to use sparse storage is to reduce CPU time: vector-matrix multiplication requires $O(|\mathcal{S}|^2)$ floating point multiplications if $\mathbf{R}$ is stored in full, but only $O(\eta(\mathbf{R}))$ with sparse storage.

Chiola [13] uses a technique to reduce memory consumption by storing an index into a floating point array, instead of the floating point value itself, for each nonzero entry of $\mathbf{R}$. If $\mathbf{R}$ contains $d$ different values in its entries, this allows us to use $\lceil \log d \rceil$ bits in each entry, instead of the usual 32 or 64 bits required to store a floating point number, at the expense of an additional floating point array of size $d$. Of course, this approach introduces overhead due to the additional indirection, and it is beneficial memory-wise only when $d \ll \eta(\mathbf{R})$.

If we use full storage for $\mathbf{R}$, we could store the holding time vector $\mathbf{h}$ on the diagonal, since the diagonal of $\mathbf{R}$ is zero and is not used. If we use sparse storage for $\mathbf{R}$, the holding time vector $\mathbf{h}$ is best stored separately as a full vector, since none of its entries are zero.

## 4.2 Kronecker plus state space

First, we recall the definition of the Kronecker product $\mathbf{A} = \mathbf{A}^K \otimes \cdots \otimes \mathbf{A}^1$ of $K$ square matrices $\mathbf{A}^k \in I\!R^{n_k \times n_k}$. In the following, we use a fixed *mixed-base* sequence $(n_K, \ldots, n_1)$ corresponding to the sizes of the $K$ matrices. We will then identify a sequence $(i_K, \ldots, i_1)$ with its mixed-base value:

$$(\cdots((i_K)n_{K-1} + i_{K-1})\cdots)n_1 + i_1 = \sum_{k=1}^{K} \left( i_k \cdot \prod_{j=1}^{k-1} n_j \right).$$

Then, we can define the Kronecker product $\mathbf{A}$ as [19]

$$\mathbf{A}[(i_K, \ldots, i_1), (j_K, \ldots, j_1)] = \mathbf{A}^K[i_K, j_K] \cdots \mathbf{A}^1[i_1, j_1]$$

and the Kronecker sum as

$$\bigoplus_{k=1}^{K} \mathbf{A}^k = \sum_{k=1}^{K} \mathbf{I}_{n_K} \otimes \cdots \otimes \mathbf{I}_{n_{k+1}} \otimes \mathbf{A}^k \otimes \mathbf{I}_{n_{k-1}} \otimes \cdots \otimes \mathbf{I}_{n_1}$$

where $\mathbf{I}_n$ is the $n \times n$ identity matrix.

The idea behind Kronecker approaches to SPN solution [9, 17, 21, 24] is to represent the transition rate matrix $\mathbf{R}$ as the submatrix corresponding to the reachable portion of the matrix $\hat{\mathbf{R}} \in I\!R^{|\hat{\mathcal{S}}| \times |\hat{\mathcal{S}}|}$ defined by

$$\hat{\mathbf{R}} = \sum_{t \in \mathcal{T}_S} \bigotimes_{k=1}^{K} \mathbf{W}_t^k + \bigoplus_{k=1}^{K} \mathbf{R}^k \qquad (1)$$

where $\mathbf{W}_t^k$ describes the effect of synchronizing transition $t$ on subnet $k$ ($\mathbf{W}_t^k = \mathbf{I}_{|\mathcal{S}_k|}$ if $t \notin \mathcal{T}_k$), while $\mathbf{R}^k$ describes the effect of transitions local to subnet $k$. Thus,

we need to store only the matrices $\mathbf{W}_t^k$ and $\mathbf{R}^k$, which require a negligible amount of storage compared to $\mathbf{R}$.

This approach is possible provided that, in addition to the "local dependency for the arc cardinalities" and the "product form form the transition guards" of Sect. 3.6, a third structural decomposition property holds:

**Product form for the rates.** It must be possible to express the rate of a transition $t$ as the product of $K$ local functions: $\lambda(t, \mathbf{m}) = \lambda_K(t, \mathbf{m}_K) \cdots \lambda_1(t, \mathbf{m}_1)$.

We extend the concept of transition locality to include the rate function: if $\lambda_k(t, \bullet)$ is not identically equal one, then $t \in \mathcal{T}_k$.

Then, we can define the entries of $\mathbf{W}_t^k$ and $\mathbf{R}^k$. $\mathbf{W}_t^k[i,j] = g_k(t,i) \cdot \lambda_k(t,i)$ if $t$ is locally enabled by local marking $i$ and its firing brings the places in $\mathcal{P}_k$ to the local marking $j$; otherwise $\mathbf{W}_t^k[i,j] = 0$. This implies that, under our assumption of no immediate transitions, $\mathbf{W}_t^k$ has at most one nonzero entry per row. Analogously, $\mathbf{R}[i,j]$ is the sum of $g_k(t,i) \cdot \lambda_k(t,i)$ over all transitions local to subnet $k$ which are enabled in $i$ and whose firing changes the local marking to $j$.

As with traditional storage techniques, we must also store either $\mathbf{h}$ or the diagonal elements of $\mathbf{Q}$. We can still choose to use a full vector of size $|\mathcal{S}|$ and store $\mathbf{h}$ explicitly, for the best performance. Alternatively, we can compute its entries as needed [39] using a second Kronecker expression, since the diagonal of $\mathbf{Q}$ is given by the diagonal entries corresponding to $\mathcal{S}$ in

$$\sum_{t \in \mathcal{T}_S} \bigotimes_{k=1}^{K} Diag(\mathbf{W}_t^k \cdot \mathbf{1}) + \bigoplus_{k=1}^{K} Diag(\mathbf{R}^k \cdot \mathbf{1})$$

(i.e., we store vectors containing the row sums for each $\mathbf{W}_t^k$ and $\mathbf{R}^k$ matrix instead of storing $\mathbf{h}$).

Using a Kronecker representation introduces two sources of overhead. First, simply applying the definition of Kronecker product has the potential of increasing computational costs by a factor $O(K)$, since each entry of a matrix expressed as a Kronecker product is obtained as the product of $K$ real numbers. Second, Eq. 1 describes $\hat{\mathbf{R}}$, not $\mathbf{R}$; the indexing difference between the two needs to be managed somehow (more on this in Sect. 5).

## 4.3 MTBDDs

An alternative technique for representing $\mathbf{R}$ is to use *multi-terminal* BDDs (MTBDDs), an extension to BDDs where the terminal nodes are not labeled with 0 and 1, but with real numbers. Thus, MTBDDs can encode functions of the form $f : \{0,1\}^n \rightarrow I\!R$. In particular, $\mathbf{R}$ can be stored by the MTBDD encoding the function $f(i_K, \ldots, i_1, j_K, \ldots, j_1) = \mathbf{R}[i,j]$, where $(i_K, \ldots, i_1)$ and $(j_K, \ldots, j_1)$ are the binary encodings of $i$ and $j$. In practice, various encodings are used for $i$ and $j$, and the function variables are usually "interleaved"; [23] discusses such issues in detail.

Not unlike Chiola's approach mentioned in Sect. 4.1, the efficiency of this approach depends very much on the number of unique values of the nonzero entries in $\mathbf{R}$. In the worst case, the nonzero values of $\mathbf{R}$ are all distinct and the MTBDD is a binary tree with $\eta(\mathbf{R})$ terminal nodes, hence at least $\eta(\mathbf{R}) - 1$ non-terminal nodes. In this case, an MTBDD representation requires an amount of memory comparable to the traditional sparse storage. In the best case, all the nonzero values are equal, and the MTBDD simply encodes the incidence matrix of the reachability graph. In this case, the MTBDD representation can be much more compact than traditional sparse storage, although pathological cases exist, where the MTBDD still requires $O(\eta(\mathbf{R}))$ memory, for pathological incidence matrix structures.

## 4.4 Matrix diagrams

Another way to represent $\mathbf{R}$ is to use *matrix diagrams* [16], which combine Kronecker- and BDD-based approaches. A $K$-level matrix diagram is analogous to an MDD, except that the nodes at each level are matrices, not vectors. An entry of a level $k$ matrix is a list of pairs of the form $(r_k, \phi_k)$ where $r_k$ is a real number and $\phi_k$ is a pointer to a level $k-1$ matrix (for $k=1$, $\phi_k$ is "undefined" and the list contains only one element, that is, a level 1 matrix is an ordinary real matrix). An element of $\mathbf{R}$ is accessed in a manner similar to that of the Kronecker representation: $\mathbf{R}[(i_K, \ldots, i_1), (j_k, \ldots, j_1)]$ is the sum of the products $r_K \cdots r_1$ over all sequences of the form $((r_K, \phi_K), \ldots, (r_1, \phi_1))$ where $(r_k, \phi_k)$ is an element of the list in position $[i_k, j_k]$ of the matrix pointed by $\phi_{k+1}$. For instance, looking at the matrix diagram shown in Fig. 4, we can compute the value of element $\mathbf{R}[(B,E,N),(C,A,N)]$ as $3.1 \cdot 2.0 \cdot 0.0 + 7.0 \cdot 1.0 \cdot 1.0 = 7.0$.

This approach can be applied provided the same three structural decomposition conditions required for the Kronecker approach are satisfied.

First we build the matrix diagram representation for $\hat{\mathbf{R}}$ according to Eq. 1 (it is immediate to see how Kronecker products and sums apply to matrix diagrams as well). Then, using an appropriate MDD representation for $\mathcal{S}$, the unreachable rows and columns of $\hat{\mathbf{R}}$ are eliminated, leaving the matrix diagram encoding $\mathbf{R}$.

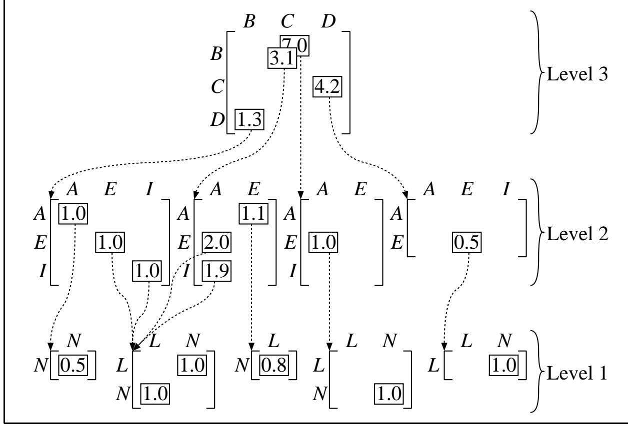


Figure 4: A matrix diagram.

# 5 Solution algorithms

Once we have a representation for the transition rate matrix $\mathbf{R}$ or the infinitesimal generator matrix $\mathbf{Q}$, we can compute $\boldsymbol{\pi}$ by solving the linear system $\boldsymbol{\pi}\cdot\mathbf{Q} = \mathbf{0}$, subject to the constraint that the elements of $\boldsymbol{\pi}$ sum to one. Iterative approaches are normally used. These compute a sequence $\boldsymbol{\pi}^{(0)}, \boldsymbol{\pi}^{(1)}, \ldots, \boldsymbol{\pi}^{(N)}$ of approximations to $\boldsymbol{\pi}$, stopping when $\boldsymbol{\pi}^{(N)}$ satisfies some criterion such as

**Residual:** $||\boldsymbol{\pi}^{(N)} \cdot \mathbf{Q}||_1 < \epsilon$.

**Absolute increments:** $||\boldsymbol{\pi}^{(N)} - \boldsymbol{\pi}^{(N-1)}||_\infty < \epsilon$.

**Relative increments:** $\max_i \frac{|\boldsymbol{\pi}^{(N)}[i] - \boldsymbol{\pi}^{(N-1)}[i]|}{\boldsymbol{\pi}^{(N)}[i]} < \epsilon$.

This is done by vector-matrix multiplications of the form

$$\boldsymbol{\pi}^{(n)} = \boldsymbol{\pi}^{(n-1)} \cdot \mathbf{M} \tag{2}$$

where the definition of $\mathbf{M}$ depends on our iterative method. These techniques require us to use some initial "guess" probability vector $\boldsymbol{\pi}^{(0)}$; often this is chosen to be the uniform vector $\boldsymbol{\pi}^{(0)} = \frac{1}{|\mathcal{S}|} \cdot \mathbf{1}$.

Commonly used iterative methods include *Power* and *Jacobi*, which use $\mathbf{M} = \mathbf{R} \cdot h$, with $h \geq \max_i \mathbf{h}[i]$ and $\mathbf{M} = \mathbf{R} \cdot \mathit{Diag}(\mathbf{h})$, respectively. $\mathbf{M}$ is not computed explicitly in practice. For example, $\boldsymbol{\pi}^{(n)} = \boldsymbol{\pi}^{(n-1)} \cdot \mathbf{R} \cdot \mathit{Diag}(\mathbf{h})$, is computed by first performing the multiplication $\boldsymbol{\pi}^{(n-1)} \cdot \mathbf{R}$ and then scaling the elements by $\mathbf{h}$.

The method of Jacobi does not use the most recent values of elements of $\boldsymbol{\pi}$ during the iteration: in our computation of element $\boldsymbol{\pi}^{(n)}[i]$ we use the elements of $\boldsymbol{\pi}^{(n-1)}$, even though some elements of $\boldsymbol{\pi}^{(n)}$ might be available already. The *Gauss-Seidel* iteration, instead, uses the most recent knowledge of $\boldsymbol{\pi}$ during the computation. Formally, the iteration matrix used is $\mathbf{M} = \mathbf{L} \cdot (\mathit{Diag}(\mathbf{R} \cdot 1) - \mathbf{U})^{-1}$, where $\mathbf{L}$ and $\mathbf{U}$ are the lower- and upper-triangular portions of $\mathbf{R}$. In practice, Gauss-Seidel is implemented using a single vector for $\boldsymbol{\pi}$ and over-writing the values of $\boldsymbol{\pi}^{(n-1)}$ with those of $\boldsymbol{\pi}^{(n)}$ as they are computed. This technique requires us to compute the elements of $\boldsymbol{\pi}$ one at a time, since element $\boldsymbol{\pi}^{(n)}[i-1]$ is used in the computation of $\boldsymbol{\pi}^{(n)}[i]$; hence, we must be able to access individual columns of $\mathbf{R}$.

The advantage of Gauss-Seidel over Jacobi, and even more over Power, is that the rate of convergence is usually better, thus fewer iterations are required. *Projection* methods have also been proposed for the solution of CTMCs. These compute increasingly accurate approximations of $\boldsymbol{\pi}$ as linear combinations of the vectors in the Krylov base $\{\mathbf{y}, \mathbf{y} \cdot \mathbf{A}, \ldots, \mathbf{y} \cdot \mathbf{A}^m - 1\}$, for an appropriate choice of $\mathbf{y}$ and $\mathbf{A}$ [38]. While these methods often have excellent convergence rate, they require to store $m$ vectors of size $|\mathcal{S}|$, thus their memory requirements can be very high. We do not consider these further, especially since they have been mainly applied in conjunction with traditional storage of $\mathbf{R}$, although there have been some successful application using the Kronecker approach [8, 38].

## 5.1 Traditional

With this simple approach, we use a traditional sparse representation of $\mathbf{R}$ and a full vector representation of $\boldsymbol{\pi}$. If we use the Jacobi method, we can use any sparse representation for $\mathbf{R}$. Instead, Gauss-Seidel requires us to use a sparse representation with efficient column access for $\mathbf{R}$. The set $\mathcal{S}$ is not used at all during the computation of $\boldsymbol{\pi}$, but only at the end, to computed expected reward measures of the form $\sum_{\mathbf{m}\in\mathcal{S}} \boldsymbol{\pi}_{\Psi(\mathbf{m})} \cdot \rho(\mathbf{m})$, where $\rho(\mathbf{m})$ is the reward rate when the model is in state $\mathbf{m}$.

## 5.2 Kronecker using $\hat{\mathcal{S}}$

This is the first approach proposed for the use of a Kronecker structure [20, 33]. Using a full vector for $\hat{\boldsymbol{\pi}} \in I\!R^{|\hat{\mathcal{S}}|}$, we solve the system $\hat{\boldsymbol{\pi}} \cdot \hat{\mathbf{Q}} = \mathbf{0}$, where $\hat{\mathbf{Q}}$ is represented in Kronecker form. Since $\hat{\mathbf{Q}}$ contains *spurious* entries not corresponding to entries in $\mathbf{Q}$, we must find a way to ignore them, for both correctness and efficiency reasons.

With Jacobi (or Power), we can compute the product $\hat{\boldsymbol{\pi}}^{(n-1)} \cdot \hat{\mathbf{R}}$, where $\hat{\mathbf{R}}$ is represented as in Eq. 1, while accessing $\hat{\mathbf{R}}$ by rows. This requires to store the matrices $\mathbf{W}_t^k$ and $\mathbf{R}^k$ by rows, and the initial guess $\hat{\boldsymbol{\pi}}^{(0)}$ must contain zero probabilities for the unreachable states (this ensures that we ignore spurious entries and that the final probability vector $\hat{\boldsymbol{\pi}}^{(N)}$ will also have this property). In practice, provided we can derive the sets $\mathcal{S}_K, \ldots, \mathcal{S}_1$ in

isolation, we do not even need to generate $\mathcal{S}$. Instead of initializing $\hat{\boldsymbol{\pi}}^{(0)}[i] = 1/|\mathcal{S}|$ if $\hat{\Psi}^{-1}(i) \in \mathcal{S}$ and 0 otherwise, we simply set all its entries to zero except for $\hat{\boldsymbol{\pi}}^{(0)}[\Psi(\mathbf{m}^{[0]})]$, which is set to 1.

With Gauss-Seidel, we must instead access a column of $\hat{\mathbf{R}}$ at a time, so $\mathbf{W}^k_t$ and $\mathbf{R}^k$ are stored by columns. However, it is possible for a reachable column of $\hat{\mathbf{R}}$ to contain spurious entries (i.e., have nonzero entries on unreachable rows). To skip over these entries when performing the multiplications we must use some representation for $\mathcal{S}$ and check each entry of the column to make sure its row corresponds to a reachable state. Algorithms to perform these multiplications are discussed in detail in [10].

### 5.3 Kronecker using $\mathcal{S}$

An alternative to the above is to use a full vector for $\boldsymbol{\pi}$ but still maintain a Kronecker representation for $\hat{\mathbf{Q}}$. Of course, this approach requires to generate $\mathcal{S}$ first.

With the Jacobi method and access by rows, if we access the reachable rows only, we still do not need to worry about spurious entries. However, the indexing change from $\hat{\mathbf{R}}$ to $\mathbf{R}$ requires us to compute $\Psi(\mathbf{m})$ for each entry of $\hat{\mathbf{R}}$ corresponding to a transition from a reachable marking to marking $\mathbf{m}$. If a multi-level structure is used for $\mathcal{S}$, an interleaving algorithm can be used and the computation of $\Psi(\mathbf{m})$ is done by levels. This allows us to amortize the cost of searches, but it still implies an overhead of $\log|\mathcal{S}_1|$ with sparse arrays.

With Gauss-Seidel, we require column access of $\mathbf{R}$ and we must worry about both spurious entries and the indexing change. To make matters worse, the column equivalent of the interleaved multiplication algorithm cannot be used. This means that the overhead factor for the searches can be as high as $\log|\mathcal{S}|$. These algorithms are also analyzed in [10].

### 5.4 MTBDDs for $\mathbf{R}$ and $\boldsymbol{\pi}$

The technique for MTBDD-based solution in [23] uses an MTBDD representation for the matrix $\mathbf{M}$ of Eq. 2. Vectors $\boldsymbol{\pi}^{(n)}$ are also stored using MTBDDs, and computed using vector-matrix multiplication algorithms that work on MTBDDs [4]. A disadvantage of the technique is that it uses Jacobi, as Gauss-Seidel would require to store its iteration matrix $\mathbf{M}$ explicitly, but this would require a nontrivial matrix inversion.

Again, in the worst case, $\boldsymbol{\pi}$ contains no duplicate values and the MTBDD for $\boldsymbol{\pi}$ requires more memory than a simple full vector. When $\boldsymbol{\pi}$ contains instead many duplicates, significant memory and even time savings can be achieved with this approach. However, when this occurs, it might very well be an indication that the SPN contains unrecognized symmetries which were not adequately exploited in the model.

### 5.5 Matrix diagrams

With this approach, we use an MDD representation for $\mathcal{S}$ and a full vector for $\boldsymbol{\pi}$. Using Jacobi, we can simply perform the vector-matrix multiplication $\boldsymbol{\pi}^{(n)} \cdot \mathbf{R}$ accessing one row at a time, as in the case of the Kronecker representation. Using Gauss-Seidel, we must instead access $\mathbf{R}$ by columns. This is done recursively in a bottom-up fashion. Since the unreachable rows are not present in the matrix diagram representation, we do not need to worry about spurious entries as in the Kronecker representation. Just as importantly, each matrix can have multiple incoming pointers (just like a node in a BDD or MDD), and we can reduce computation significantly by maintaining a cache for recently computed subcolumns. This way, duplication of work is avoided and a greater degree of amortization is possible. Nevertheless, in [16], the solution times are still a factor or three or so slower than with traditional sparse storage solution, when $\mathbf{R}$ fits in memory.

## 6 Conclusion

We explored several approaches for the storage of the reachability set $\mathcal{S}$ and the transition rate matrix $\mathbf{R}$.

If only a logical analysis is sought, recent approaches using BDDs and MDDs are clear winners: $\mathcal{S}$ with as many as $10^{20}$ markings can be generated and stored in reasonable time and memory; for particularly “nice” nets, even $10^{600}$ markings are possible [27]. There appears to be little reason to use traditional methods based on hashing or search trees, except perhaps in the case of extremely unstructured models.

If a Markov solution is instead sought, the situation is more complex. For SPNs with up to a few million markings, the transition rate matrix will likely fit in memory. In this case, traditional sparse technology usually provides the fastest answers, and has the additional advantages of being easy to implement and allowing experimentation with different numerical techniques.

Structured approaches based on a Kronecker representation are likely to be a better choice only when $\mathbf{R}$ cannot fit in memory; then, the size of the problem that can be solved is only limited by the storage of the solution vector and any other vector used by the numerical algorithm, perhaps a few tens of million markings. However, the solution has additional overhead due to the structured representation; matrix diagrams reduce this overhead considerably, but not completely. Using

MTBDD techniques for both $\mathbf{R}$ and $\boldsymbol{\pi}$ might further reduce the memory requirements, but only in special cases where the probability vector contains many entries with the same value. With few exceptions, structured approaches have been introduced using simple iterative methods such as Power, Jacobi, or, at best Gauss-Seidel or SOR. A recent line of inquiry is then the adoption of more sophisticated techniques that reduce the number of iterations required for numerical convergence.

# References


[1] G. M. Adelson-Velsky and E. M. Landis. An algorithm for the organization of information. *Soviet Math.*, 3:1259–1263, 1962.

[2] T. Agerwala. A complete model for representing the coordination of asynchronous processes. Hopkins Computer Research Report 32, Johns Hopkins University, Baltimore, Maryland, July 1974.

[3] M. Ajmone Marsan, F. Donatelli, Susanna Neri, and A. Scalia. Approximate GSPN analysis of multiserver polling systems. In *Proceedings of the International Conference on Telecommunication, Distribution and Parallelism (TDP'96)*, La Londe Les Maures, France, June 1996.

[4] R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Maciii, A. Pardo, and F. Somenzi. Algebraic decision diagrams and their applications. *Formal methods in system design*, 10(2/3):171–206, Apr. 1997.

[5] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comp.*, 35(8):677–691, Aug. 1986.

[6] R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comp. Surv.*, 24(3):393–318, 1992.

[7] P. Buchholz. Hierarchical structuring of superposed GSPNs. In *Proc. 7th Int. Workshop on Petri Nets and Performance Models (PNPM'97)*, pages 81–90, St. Malo, France, June 1997. IEEE Comp. Soc. Press.

[8] P. Buchholz. Projection methods for the analysis of stochastic automata networks. In B. Plateau, W. J. Stewart, and M. Silva, editors, *Numerical Solution of Markov Chains*, pages 149–168. Prensas Universitarias de Zaragoza, Zaragoza, Spain, Sept. 1999.

[9] P. Buchholz. Structured analysis approaches for large Markov chains. *Applied Numerical Mathematics*, 31(4):375–404, 1999.

[10] 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.* To appear.

[11] J. Campos, M. Silva, and S. Donatelli. Structured solution of stochastic DSSP systems. In *Proc. 7th Int. Workshop on Petri Nets and Performance Models (PNPM'97)*, pages 91–100, St. Malo, France, June 1997. IEEE Comp. Soc. Press.

[12] G. Chiola. Compiling techniques for the analysis of stochastic Petri nets. In *Proc. 4th Int. Conf. on Modelling Techniques and Tools for Performance Evaluation*, pages 13–27, 1989.

[13] G. Chiola. GreatSPN 1.5 software architecture. In *Proc. 5th Int. Conf. Modeling Techniques and Tools for Computer Performance Evaluation*, pages 117–132, Torino, Italy, Feb. 1991.

[14] G. Ciardo. Petri nets with marking-dependent arc multiplicity: properties and analysis. In R. Valette, editor, *Application and Theory of Petri Nets 1994 (Proc. 15th Int. Conf. on Applications and Theory of Petri Nets, Zaragoza, Spain)*, Lecture Notes in Computer Science 815, pages 179–198. Springer-Verlag, June 1994.

[15] G. Ciardo and A. S. Miner. Storage alternatives for large structured state spaces. In R. Marie, B. Plateau, M. Calzarossa, and G. Rubino, editors, *Proc. 9th Int. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation*, Lecture Notes in Computer Science 1245, pages 44–57, St. Malo, France, June 1997. Springer-Verlag.

[16] G. Ciardo and A. S. Miner. A data structure for the efficient Kronecker solution of GSPNs. In P. Buchholz, editor, *Proc. 8th Int. Workshop on Petri Nets and Performance Models (PNPM'99)*, pages 22–31, Zaragoza, Spain, Sept. 1999. IEEE Comp. Soc. Press.

[17] G. Ciardo and M. Tilgner. On the use of Kronecker operators for the solution of generalized stochastic Petri nets. ICASE Report 96-35, Institute for Computer Applications in Science and Engineering, Hampton, VA, May 1996.

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

[19] M. Davio. Kronecker products and shuffle algebra. *IEEE Trans. Comp.*, C-30:116–125, Feb. 1981.

[20] S. Donatelli. Superposed Stochastic Automata: a class of stochastic Petri nets with parallel solution and distributed state space. *Perf. Eval.*, 18:21–26, 1993.

[21] S. Donatelli. Superposed generalized stochastic Petri nets: definition and efficient solution. In R. Valette, editor, *Application and Theory of Petri Nets 1994 (Proc. 15th Int. Conf. on Applications and Theory of Petri Nets)*, Lecture Notes in Computer Science 815, pages 258–277, Zaragoza, Spain, June 1994. Springer-Verlag.

[22] B. Haverkort, A. Bell, and H. Bohnenkamp. On the efficient sequential and distributed generation of very large Markov chains from stochastic Petri nets. In P. Buchholz, editor, *Proc. 8th Int. Workshop on Petri Nets and Performance Models (PNPM'99)*, pages 12–21, Zaragoza, Spain, Sept. 1999. IEEE Comp. Soc. Press.

[23] H. Hermanns, J. Kayser, and M. Siegle. Multi terminal binary decision diagrams to represent and analyse continuous time Markov chains. In B. Plateau, W. J. Stewart, and M. Silva, editors, *Numerical Solution of Markov Chains*, pages 188–207, Zaragoza, Spain, June 1997. Prensas Universitarias de Zaragoza.

[24] P. Kemper. Numerical analysis of superposed GSPNs. *IEEE Trans. Softw. Eng.*, 22(4):615–628, Sept. 1996.

[25] P. Kemper. Reachability analysis based on structured representations. In J. Billington and W. Reisig, editors, *Application and Theory of Petri Nets 1996 (Proc. 17th Int. Conf. on Applications and Theory of Petri Nets, Osaka, Japan)*, Lecture Notes in Computer Science 1091, pages 269–288. Springer-Verlag, June 1996.

[26] C. Y. Lee. Representation of switching circuits by binary-decision programs. *Bell Syst. Techn. J.*, 38(4):985–999, July 1959.

[27] A. S. Miner and G. Ciardo. Efficient reachability set generation and storage using decision diagrams. In H. Kleijn and S. Donatelli, editors, *Application and Theory of Petri Nets 1999 (Proc. 20th Int. Conf. on Applications and Theory of Petri Nets, Williamsburg, VA, USA)*, Lecture Notes in Computer Science 1639, pages 6–25. Springer-Verlag, June 1999.

[28] A. S. Miner, G. Ciardo, and S. Donatelli. Using the exact state space of a Markov model to compute approximate stationary measures. In *Proc. 2000 ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, Santa Clara, CA, June 2000. To appear.

[29] E. Pastor and J. Cortadella. Efficient encoding schemes for symbolic analysis of Petri nets. In *Proc. Design Automation and Test in Europe*, Feb. 1998.

[30] E. Pastor and J. Cortadella. Structural methods applied to the symbolic analysis of Petri nets. In *Proc. IEEE/ACM International Workshop on Logic Synthesis*, June 1998.

[31] E. Pastor, O. Roig, J. Cortadella, and R. Badia. Petri net analysis using boolean manipulation. In R. Valette, editor, *Application and Theory of Petri Nets 1994, (Proc. 15th Int. Conf. on Applications and Theory of Petri Nets, Zaragoza, Spain)*, Lecture Notes in Computer Science 815, pages 416–435. Springer-Verlag, June 1994.

[32] S. Pissanetzky. *Sparse Matrix Technology.* Academic Press, 1984.

[33] B. Plateau. On the stochastic structure of parallelism and synchronisation models for distributed algorithms. In *Proc. 1985 ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, pages 147–153, Austin, TX, USA, May 1985.

[34] M. Sereno. Approximate mean value analysis technique for non-product form solution stochastic Petri nets: an application to stochastic marked graphs. In *Proc. 6th Int. Workshop on Petri Nets and Performance Models (PNPM'95)*, pages 42–51, Durham, NC, Oct. 1995. IEEE Comp. Soc. Press.

[35] M. Sereno and G. Balbo. Computational algorithms for product form solution stochastic Petri nets. In *Proc. 5th Int. Workshop on Petri Nets and Performance Models (PNPM'93)*, pages 98–107, Toulouse, France, Oct. 1993. IEEE Comp. Soc. Press.

[36] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of the Association for Computing Machinery*, 32(3):652–686, July 1985.

[37] A. Srinivasan, T. Kam, S. Malik, and R. K. Brayton. Algorithms for discrete function manipulation. In *International Conference on CAD*, pages 92–95. IEEE Computer Society, 1990.

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

[39] W. J. Stewart, K. Atif, and B. Plateau. The numerical solution of stochastic automata networks. *Europ. J. of Oper. Res.*, 86:503–525, 1995.