

# Distributed and Structured Analysis Approaches to Study Large and Complex Systems\*

Gianfranco Ciardo

Department of Computer Science  
College of William and Mary  
Williamsburg, VA 23187, USA  
`ciardo@cs.wm.edu`

**Abstract.** Both the logic and the stochastic analysis of discrete-state systems are hindered by the combinatorial growth of the state space underlying a high-level model. In this work, we consider two orthogonal approaches to cope with this “state-space explosion”. Distributed algorithms that make use of the processors and memory overall available on a network of  $N$  workstations can manage models with state spaces approximately  $N$  times larger than what is possible on a single workstation. A second approach, constituting a fundamental paradigm shift, is instead based on decision diagrams and related implicit data structures that efficiently encode the state space or the transition rate matrix of a model, provided that it has some structure to guide its decomposition; with these implicit methods, enormous sets can be managed efficiently, but the numerical solution of the stochastic model, if desired, is still a bottleneck, as it requires vectors of the size of the state space.

## 1 Introduction

As digital systems are becoming ubiquitous and their complexity is steadily growing, it is increasingly important to be able to study their logical and timing behavior. While direct observation, testing, and measurement are sometimes feasible, they are normally very expensive, and, by definition, can only be employed after the system has been built. Discrete-state models are then an attractive, general, and inexpensive alternative that provides a way to study a system at various levels of detail, even when it is still just a concept in the designer’s mind. However, the discrete nature of the system implies that its “state” is the collection of the states of each of its components, resulting in the well-known “combinatorial explosion” of the state space, which poses a formidable analysis challenge even for today’s powerful computers.

In this work, we consider techniques to cope with this state-space explosion problem, focusing in particular on models that have an underlying continuous-time Markov chain (CTMC). Our presentation is split into three main portions.

---

\* This work was supported by the National Aeronautics and Space Administration under NASA Grant NAG-1-2168.

Sec. 3 introduces the main concepts related to the underlying state space and CTMC, and their traditional solution methods. Sec. 4 discusses distributed analysis algorithms that can be used to spread the memory and time requirements over a network of  $N$  workstations, thus are able to cope with state spaces approximately  $N$  time larger. Sec. 5 moves instead to what we call “implicit” techniques which require much less than linear memory to store the state space and the CTMC. In the former case, this also results in enormous time savings, while, in the latter, the approach usually involves a memory-time tradeoff. A unified treatment of these two aspects shows some of the commonalities and research challenges. Finally, in Sec. 6 we briefly conclude with our thoughts about fruitful directions of future research.

## 2 Notation

We use italic letters to indicate scalars (e.g.,  $a$ ), calligraphic letters to indicate sets (e.g.,  $\mathcal{A}$ ), lower case boldface Roman or Greek letters to indicate row vectors (e.g.,  $\mathbf{a}$ ,  $\boldsymbol{\alpha}$ ), and upper case boldface Roman letters to indicate matrices (e.g.,  $\mathbf{A}$ ). Vector and matrix elements are indicated using square brackets (e.g.,  $\mathbf{a}[1]$ ,  $\mathbf{A}[1, 2]$ ), and we extend the notation to sub-vectors or sub-matrices by allowing sets of indices to be used instead of single indices (e.g.,  $\mathbf{a}[\mathcal{A}]$ ,  $\mathbf{A}[\mathcal{A}, \mathcal{B}]$ ).

We indicate with  $\text{diag}(\mathbf{a})$ ,  $\mathbf{I}_n$ , and  $\mathbf{0}$  the diagonal matrix with vector  $\mathbf{a}$  on its main diagonal, the identity matrix of size  $n \times n$  ( $n$  is omitted if clear from the context), and a row vector of 0’s of the appropriate dimension, respectively.

We use subscripts to indicate event indices (e.g.,  $\mathbf{R}_e$ ); in the discussion of distributed approaches, we also use subscripts, but in square brackets, for process indices (e.g.,  $\mathcal{S}_{[n]}$ ) while, in the discussion on implicit methods, we use subscripts, without square brackets, for sub-model indices (e.g.,  $\mathcal{S}_k$ ); if both sub-model and event indices are present, they appear in that order (e.g.,  $\mathbf{W}_{k,e}$ ). We consistently number and use submodel indices “going down” from  $K$  to 1, never up; the reader should keep this in mind when operators such as Kronecker product and sum are used, since these are not commutative.

States are denoted in boldface lower case letters, just as vectors, to stress that they are somehow structured, that is, they are a collection of sub-states.

## 3 High-Level Models and Traditional Solution Methods

Rather than discussing analysis techniques for a particular formalism such as Petri nets [34] or process algebras [3], we introduce instead a general framework for discrete-state models. This is preferable since nothing in our discussion is tied to the particular formalism chosen. However, we will sometimes make specific references to Petri nets, since we regard them as quite graphically intuitive (or perhaps simply because we are most familiar with them!). For more information on Petri nets and their stochastic extensions, see G. Balbo, this volume.

<i>ExploreExplicitSequential</i> : set of state	
Build and return the state space $\mathcal{S}$ .	
declare $\mathcal{S}, \mathcal{U}$ : set of state; declare $\mathbf{i}, \mathbf{j}$ : state;	
1. $\mathcal{S} \leftarrow \emptyset$ ; 2. $\mathcal{U} \leftarrow \{\mathbf{s}^{initial}\}$ ; 3. while $\exists \mathbf{i} \in \mathcal{U}$ do 4.   for each $\mathbf{j} \in \mathcal{N}(\mathbf{i})$ do 5.     if $\mathbf{j} \notin \mathcal{U} \cup \mathcal{S}$ then 6. $\mathcal{U} \leftarrow \mathcal{U} \cup \{\mathbf{j}\}$ ; 7.     end if; 8.   end for; 9. $\mathcal{U} \leftarrow \mathcal{U} \setminus \{\mathbf{i}\}$ ; 10. $\mathcal{S} \leftarrow \mathcal{S} \cup \{\mathbf{i}\}$ ; 11. end while; 12. return $\mathcal{S}$ ;	<ul style="list-style-type: none"> <li>• initialize the explored states</li> <li>• initialize the unexplored states to the initial state</li> <li>• there are still states to explore</li> <li>• found a new state <math>\mathbf{j}</math></li> <li>• move <math>\mathbf{i}</math> from unexplored. . .</li> <li>• . . . to explored</li> </ul>

**Fig. 1.** An explicit sequential algorithm to generate the state space.

### 3.1 State-Space Generation

We consider discrete-state models having a finite underlying state space  $\mathcal{S}$ . Any such high-level model must describe the following objects in a compact way:

- $\widehat{\mathcal{S}}$ , the set of *potential states*, describing the “type” of the system states.
- $\mathbf{s}^{initial} \in \widehat{\mathcal{S}}$ , the *initial state* of the system.
- $\mathcal{N} : \widehat{\mathcal{S}} \rightarrow 2^{\widehat{\mathcal{S}}}$ , the *next-state function*, describing which states can be reached from a given state in a single step, or transition.

In many formalisms, the next-state function  $\mathcal{N}$  is expressed as  $\mathcal{N} = \bigcup_{e \in \mathcal{E}} \mathcal{N}_e$ , where  $\mathcal{E}$  is a finite set of *events*,  $\mathcal{N}_e$  is the next-state function associated with event  $e$ , and the union operator is naturally meant to be applied to the values of the involved functions, that is,  $\mathcal{N}(\mathbf{i}) = \bigcup_{e \in \mathcal{E}} \mathcal{N}_e(\mathbf{i})$ . Then,  $\mathcal{N}_e(\mathbf{i})$  is the set of states the system can enter when event  $e$  occurs, or *fires*, in state  $\mathbf{i}$ . Note that, with a single function, we encompass not only the concept of *next state*, but also that of *enabling*, since event  $e$  is enabled in  $\mathbf{i}$  iff  $|\mathcal{N}_e(\mathbf{i})| > 0$ , and of *non-determinism*, since the effect of event  $e$  is non-deterministic in state  $\mathbf{i}$  iff  $|\mathcal{N}_e(\mathbf{i})| > 1$ . In addition, of course, a model exhibits non-determinism if multiple events are enabled in a state and there is no a priori way to choose among them.

Given a high-level model, we can then start from  $\mathbf{s}^{initial}$  and build  $\mathcal{S}$  using the *explicit state-space generation* algorithm whose pseudo-code is listed in Fig. 1. In other words,  $\mathcal{S} \subseteq \widehat{\mathcal{S}}$  is the smallest set that contains the initial system state  $\mathbf{s}^{initial}$  and is closed with respect to  $\mathcal{N}$ :

$$\mathcal{S} = \{\mathbf{s}^{initial}\} \cup \mathcal{N}(\mathbf{s}^{initial}) \cup \mathcal{N}(\mathcal{N}(\mathbf{s}^{initial})) \cup \dots = \mathcal{N}^*(\mathbf{s}^{initial}),$$

where we have extended the function  $\mathcal{N}$  to allow a set of states to be its argument, and “\*” denotes reflexive and transitive closure.

The runtime and memory requirements for *ExploreExplicitSequential* depend on the data structure used to store  $\mathcal{U}$  and  $\mathcal{S}$ . The key operations are determining the new states reachable from a given state  $\mathbf{i}$  (statement 4), performed  $|\mathcal{S}|$  times, and searching whether each of them is already in  $\mathcal{U}$  or  $\mathcal{S}$  (statement 5), performed  $|\mathcal{A}| = \sum_{\mathbf{i} \in \mathcal{S}} |\mathcal{N}(\mathbf{i})|$  times, where  $\mathcal{A}$  are the possible state-to-state transitions in the system (for a Petri net, this is the cardinality of the arc set in its *reachability graph*). If we assume that the cost of computing  $\mathcal{N}(\mathbf{i})$  is proportional to  $|\mathcal{N}(\mathbf{i})|$ , that states are stored as individual entities, and that a balanced search tree is used to store  $\mathcal{U}$  and  $\mathcal{S}$  (see [19, 28] for a detailed discussion of alternative techniques), the overall time complexity of *ExploreExplicitSequential* is  $O(|\mathcal{A}| \cdot \log |\mathcal{S}|)$  and its memory complexity is  $O(|\mathcal{S}|)$ .

Once  $\mathcal{S}$  has been explored and stored, we can query it for the presence or absence of states satisfying a certain condition, or we can ask more complex questions that involve the existence of *paths* in the reachability graph. In the former case, the query requires at most to scan each state once, so it can be answered in  $O(|\mathcal{S}|)$  time. In the latter case, the complexity depends on the particular query; a recent trend is to employ *temporal logic* to specify the properties to be checked, resulting in the so-called *model checking* [22]. In our discussion, we assume that state-space generation is a goal in itself; this is the case either if we are only interested in reachability-type queries, such as “can the system reach a deadlock” (see, for example, the model checking tool Uppaal [30]), or if state-space generation is just an intermediate step in our ultimate goal, performing a stochastic analysis.

Before concluding this section, we observe that we often need to associate a *unique integer index* to a given a state  $\mathbf{i}$ . We can do so with a mapping

$$\Psi : \mathcal{S} \rightarrow \{0, \dots, |\mathcal{S}| - 1\} \quad \text{satisfying} \quad \Psi(\mathbf{i}) = \Psi(\mathbf{j}) \Rightarrow \mathbf{i} = \mathbf{j}.$$

While any mapping will do, two possibilities are mostly used in practice. In traditional solution methods, it might be convenient to define  $\Psi(\mathbf{i}) = i$  iff  $\mathbf{i}$  was the  $i^{\text{th}}$  state “discovered” by *ExploreExplicitSequential*; hence, in particular,  $\Psi(\mathbf{s}^{\text{initial}}) = 0$ ; this is a very useful choice, especially since it allows us to know the index of a state even before completing the exploration of  $\mathcal{S}$ . In the structured methods we consider,  $\Psi(\mathbf{i})$  is instead the position of  $\mathbf{i}$  in  $\mathcal{S}$  according to lexicographic order; this mapping is well-defined only once the exploration process is complete, and requires a comparison operator defined over  $\mathcal{S}$  (of course, such an operator is already required by traditional methods that use a search tree to store  $\mathcal{S}$ ). Since  $\Psi$  is a bijection, its inverse  $\Psi^{-1}$  exists. Also, we will at times use a set of states  $\mathcal{A}$  as a parameter to  $\Psi$ , with the obvious meaning  $\Psi(\mathcal{A}) = \{\Psi(\mathbf{i}) : \mathbf{i} \in \mathcal{A}\}$ .

### 3.2 Markov Chain Specification and Solution

If we are interested in the timing or probabilistic behavior of the system, the *logic* specification of the previous section must be augmented with *stochastic* information associated with each state-to-state transition. In our discussion, we limit

ourselves to the case of models having an underlying CTMC<sup>1</sup>, so our high-level model needs to specify  $Rate(\mathbf{i}, \mathbf{j})$ , the rate at which the system, when in state  $\mathbf{i}$ , transitions to state  $\mathbf{j}$ , for each reachable state  $\mathbf{i} \in \mathcal{S}$  and each  $\mathbf{j} \in \mathcal{N}(\mathbf{i})$ ; by definition,  $Rate(\mathbf{i}, \mathbf{j}) = 0$  iff  $\mathbf{j} \notin \mathcal{N}(\mathbf{i})$ . For more information on CTMCs, see “Markov Chains for Performance and Dependability Evaluation”, by B. Haverkort, this volume.

Just as for the next-state function, this rate is normally expressed in a per-event fashion:  $Rate(\mathbf{i}, \mathbf{j}) = \sum_{e \in \mathcal{E}} Rate_e(\mathbf{i}, \mathbf{j})$ , where  $Rate_e(\mathbf{i}, \mathbf{j})$  is the rate at which event  $e$  leads from state  $\mathbf{i}$  to state  $\mathbf{j} \in \mathcal{N}_e(\mathbf{i})$ .

We then define the *transition rate matrix*  $\mathbf{R}$  of the underlying CTMC as

$$\mathbf{R} \in \mathbf{R}^{|\mathcal{S}| \times |\mathcal{S}|} \quad \text{where} \quad \mathbf{R}[i, j] = Rate(\mathbf{i}, \mathbf{j}) \quad \text{and} \quad i = \Psi(\mathbf{i}), \quad j = \Psi(\mathbf{j}).$$

The entries of  $\mathbf{R}$  can be computed and stored during the for-loop iterations in *ExploreExplicitSequential*. Alternatively, it is often more efficient to generate the state space  $\mathcal{S}$  first while, at the same time, just counting the number  $\eta(\mathbf{R})$  of nonzero entries in  $\mathbf{R}$ , essentially the number of arcs in  $\mathcal{A}$ . Then, we can allocate an efficient *row-pointer column-index*<sup>2</sup> data structure [40] that requires only  $|\mathcal{S}| + 1 + \eta(\mathbf{R})$  integers and  $\eta(\mathbf{R})$  floating point numbers. A second state-space generation pass can then be used to fill this data structure with the actual values.

Given matrix  $\mathbf{R}$ , we can define the *holding-time vector*  $\mathbf{h}$  expressing the expected time the CTMC spends in each state before making a transition:

$$\mathbf{h}[i] = \left( \sum_{0 \leq j < |\mathcal{S}|} \mathbf{R}[i, j] \right)^{-1},$$

and the *infinitesimal generator matrix*  $\mathbf{Q}$ , which equals  $\mathbf{R}$  except in its diagonal entries<sup>3</sup>, defined as

$$\mathbf{Q}[i, i] = - \sum_{0 \leq i < |\mathcal{S}|, j \neq i} \mathbf{R}[i, j] = \mathbf{R}[i, i] - \mathbf{h}[i]^{-1}.$$

Then, the numerical stationary solution of an *ergodic* CTMC involves the computation of the vector  $\boldsymbol{\pi} \in \mathbf{R}^{|\mathcal{S}|}$  of *stationary state probabilities*, solution of

$$\boldsymbol{\pi} \mathbf{Q} = \mathbf{0} \quad \text{subject to} \quad \sum_{0 \leq i < |\mathcal{S}|} \boldsymbol{\pi}[i] = 1. \quad (1)$$

<sup>1</sup> An analogous discussion is valid for the discrete-time Markovian case, while more general stochastic processes present many subtle difficulties.

<sup>2</sup> Or, rather, *column-pointer row-index*, as we often need only by-column access to  $\mathbf{R}$ .

<sup>3</sup> Unlike most definitions of CTMCs, our allows for the existence of *self-transitions*  $\mathbf{R}[i, i]$  in the transition rate matrix. These are useless from a stochastic point of view, since they can be eliminated without changing the meaning and stochastic behavior of the model. However, we explicitly consider them because, when using the structured approaches of Sect. 5.2, the resulting CTMC might exhibit them. Of course, these are apparent only when considering  $\mathbf{R}$  and  $\mathbf{h}$  separately, while  $\mathbf{Q}$  does not reflect their presence, since they cancel out.

If the CTMC is instead *absorbing*, we might instead be interested in computing the vector  $\boldsymbol{\sigma} \in \mathbb{R}^{|\mathcal{T}|}$  of *expected state sojourn times until absorption*, solution of  $\boldsymbol{\sigma}\mathbf{Q}[\Psi(\mathcal{T}), \Psi(\mathcal{T})] = -\boldsymbol{\pi}(0)[\Psi(\mathcal{T})]$ , where  $\mathcal{T}$  is the set of transient states and  $\boldsymbol{\pi}(0)$  is the *initial probability vector*, whose entries, in our case, would be all zero except for a one in correspondence to  $\mathbf{s}^{initial}$ , i.e.,  $\boldsymbol{\pi}(0)[\Psi(\mathbf{s}^{initial})] = 1$ . Regardless of the nature of the CTMC, we might instead want to compute a transient solution, that is, the probability vector at time  $t$  or the time spent in each state up to time  $t$ . For simplicity, we do not consider these other types of analysis here, since we focus on the data structures and discrete algorithms that allows us to tackle models with large state spaces, but we stress that our discussion applies also to absorbing Markov chains and transient analysis.

Many numerical algorithms are available for the solution of the linear homogeneous system of Eq. 1. In practice,  $\mathcal{S}$  is very large and  $\mathbf{R}$  is very sparse, i.e., only a small portion of its entries are nonzero. Thus, *iterative* methods are preferred, where successive approximations of the exact solution  $\boldsymbol{\pi}$  are computed starting from an initial guess, without modifying  $\mathbf{R}$ , whose sparsity is preserved. We now describe the iteration performed by some popular solution methods:

- **Power:**  $\boldsymbol{\pi}^{new} \leftarrow \boldsymbol{\pi}^{old}(\mathbf{I} + \mathbf{Q}h^*)$ , where  $h^*$  is a value slightly smaller than the smallest expected sojourn time in any state,  $h^* < \min_{0 \leq i < |\mathcal{S}|} \{\mathbf{h}[i]\}$ . Element-wise, this corresponds to:

```

for  $j = 0$  to  $|\mathcal{S}| - 1$  do
   $a \leftarrow \boldsymbol{\pi}^{old}[j]$ ;
  for  $i = 0$  to  $|\mathcal{S}| - 1$  do
     $a \leftarrow a + \boldsymbol{\pi}^{old}[i]\mathbf{Q}[i, j]h^*$ ;
  end for;
   $\boldsymbol{\pi}^{new}[j] \leftarrow a$ ;
end for;

```

•  $a$  is a high-precision accumulator

The Power method is guaranteed to converge in theory, but it is often extremely slow in practice.

- **Jacobi:**  $\boldsymbol{\pi}^{new} \leftarrow \boldsymbol{\pi}^{old}\mathbf{R} \text{diag}(\mathbf{h})$ . Element-wise, this corresponds to:

```

for  $j = 0$  to  $|\mathcal{S}| - 1$  do
   $a \leftarrow 0$ ;
  for  $i = 0$  to  $|\mathcal{S}| - 1$  do
     $a \leftarrow a + \boldsymbol{\pi}^{old}[i]\mathbf{R}[i, j]\mathbf{h}[j]$ ;
  end for;
   $\boldsymbol{\pi}^{new}[j] \leftarrow a$ ;
end for;

```

•  $a$  is a high-precision accumulator

The Jacobi method does not have guaranteed convergence, but it is usually faster than the Power method in practice.

- **(Forward) Gauss-Seidel:**  $\boldsymbol{\pi}^{new} \leftarrow \boldsymbol{\pi}^{old}\mathbf{L}(\text{diag}(\mathbf{h})^{-1} - \mathbf{U})^{-1}$ , where  $\mathbf{L}$  and  $\mathbf{U}$  are the lower and strictly upper triangular portions of  $\mathbf{R}$ , respectively. Element-wise, this corresponds to:

```

for  $j = 0$  to  $|\mathcal{S}| - 1$  do
   $a \leftarrow 0$ ;
  for  $i = 0$  to  $|\mathcal{S}| - 1$  do
     $a \leftarrow a + \boldsymbol{\pi}^{curr}[i]\mathbf{R}[i, j]\mathbf{h}[j]$ ;
  end for;
end for;

```

•  $a$  is a high-precision accumulator

```

    end for;
     $\boldsymbol{\pi}^{curr}[j] \leftarrow a;$ 
  end for;

```

Note that, unlike the Power and Jacobi iterations, which require two distinct vectors,  $\boldsymbol{\pi}^{old}$  and  $\boldsymbol{\pi}^{new}$ , Gauss-Seidel uses a single vector,  $\boldsymbol{\pi}^{curr}$ , since its *old* entries are updated to the *new* values one at a time, in place. The Gauss-Seidel method does not have guaranteed convergence either, but it is at least as fast as the Jacobi method, and often much faster, so it is considered the best among these three methods. Its convergence rate is affected by the order in which the states are considered.

## 4 Explicit Distributed Solution Approaches

Explicit distributed solution methods can make use of the workstations on a local area network to increase the amount of memory available overall, while at the same time attempting to speed up the solution process. We consider first the problem of state-space generation, basing our discussion mostly on our own work [15, 35], except for the use of hashing for the mapping, which was experimented by Haverkort [28]. We should also mention the work of Caselli, Conte, and Marenzoni [14, 31], one of the first groups to work along these lines. We do not consider instead works on parallel, shared-memory, algorithms, such as the one presented in [1]; these are confronted with substantially different issues. For the distributed numerical solution of linear systems, much work is available, thus we focus only on the special case of CTMC solution [32]. We should also note that preliminary attempts at using distributed solutions in conjunction with the implicit Kronecker representations of Sect. 5.2 [11, 25], have also been proposed, but more work is needed in this area.

### 4.1 Explicit Distributed State-Space Generation

Since explicit state-space generation essentially means a breadth-first exploration of a large graph, eventually reaching each node at least once, the idea behind a distributed algorithm for state-space generation is to use multiple processes that perform this exploration concurrently on distinct nodes of the graph. Assuming we have  $N$  processors (and processes), a natural way to do this is to define a mapping

$$Proc : \mathcal{S} \rightarrow \{0, \dots, N - 1\},$$

where  $Proc(\mathbf{i})$  is the *owner* of state  $\mathbf{i}$ , i.e., the process responsible for storing and exploring  $\mathbf{i}$ . This defines a partition of  $\mathcal{S}$  into  $N$  sets  $\mathcal{S}_{[n]} = \{\mathbf{i} : Proc(\mathbf{i}) = n\}$ , for  $0 \leq n < N$ .

The outline of a distributed generation algorithm based on this mapping is given in Fig. 2. Each of the  $N$  processes performs a task analogous to that of the sequential algorithm, with the following differences:

- Only one of the  $N$  processes, the one with index  $Proc(\mathbf{s}^{initial})$ , has initially any work to do (a state to explore).

```

ExploreExplicitDistributed( $n$  : process) : set of state
Build and return  $\mathcal{S}_{[n]}$ , the portion of the state space assigned to process  $n$ .
declare  $\mathcal{S}_{[n]}, \mathcal{U}_{[n]}$  : set of state;
declare  $\mathbf{i}, \mathbf{j}$  : state;
declare  $m$  : process;
1. if  $Proc(\mathbf{s}^{initial}) = n$  then  $\mathcal{U}_{[n]} \leftarrow \{\mathbf{s}^{initial}\}$ ; else  $\mathcal{U}_{[n]} \leftarrow \emptyset$ ; end if;
2.  $\mathcal{S}_{[n]} \leftarrow \emptyset$ ;
3. while "not received terminate message" do
4.   while  $\exists \mathbf{i} \in \mathcal{U}_{[n]}$  do
5.     for each  $\mathbf{j} \in \mathcal{N}(\mathbf{i})$  do
6.        $m \leftarrow Proc(\mathbf{j})$ ;           • determine the process  $m$  owner of  $\mathbf{j}$ . . .
7.       if  $m \neq n$  then
8.          $SendState(m, \mathbf{j})$ ;           • . . . if not  $n$  itself, send  $\mathbf{j}$  to  $m$ . . .
9.       elsif  $\mathbf{j} \notin \mathcal{U}_{[n]} \cup \mathcal{S}_{[n]}$  then   • . . . otherwise explore later, if new
10.         $\mathcal{U}_{[n]} \leftarrow \mathcal{U}_{[n]} \cup \{\mathbf{j}\}$ ;
11.      end if;
12.    end for;
13.     $\mathcal{U}_{[n]} \leftarrow \mathcal{U}_{[n]} \setminus \{\mathbf{i}\}$ ;           • move  $\mathbf{i}$  from unexplored. . .
14.     $\mathcal{S}_{[n]} \leftarrow \mathcal{S}_{[n]} \cup \{\mathbf{i}\}$ ;           • . . . to explored
15.  end while;
16.   $\mathcal{U}_{[n]} \leftarrow \mathcal{U}_{[n]} \cup ReceiveStates \setminus \mathcal{S}_{[n]}$ ;   • get states sent by other processes, if any
17. end while;

```

**Fig. 2.** Distributed state-space generation for process  $n$ .

- Each “destination” state  $\mathbf{j}$  encountered in the innermost loop is managed locally only if it happens to be owned by the same process as the “source” state  $\mathbf{i}$ ; otherwise, it is sent to the correct owner.
- Occasionally, states that have been sent to a process from any of the other  $N - 1$  processes are retrieved and inserted in the set of unexplored states, if they have not yet been encountered before.
- Termination does not simply occur when the set of unexplored states for a given process becomes empty, since more states might be sent later to it by the other processes. Rather, a *termination detection* algorithm must determine when *all the processes have exhausted their unexplored state set and no more messages are in transit*<sup>4</sup>.

Several aspects in the pseudo-code of Fig. 2 can be defined in more detail, such as the mechanism to send states (it might be advantageous to batch multiple states in a single message destined to a given process) and the frequency at which a process polls the receive queue for new states sent to it (we show the call to *ReceiveStates* in the outermost loop, but in practice this check might have to be performed more frequently to avoid “parking” too many states in the

<sup>4</sup> In our studies, we used the circulating probe algorithm by Dijkstra et al. [24]; another possibility would be the scalable “non-committal barrier” described by Nicol [36].

communication buffers). However, the main decision left unspecified so far is the nature of the function used to map states to processes.

A good choice for *Proc* must be efficient to encode and compute, and should also achieve the following performance goals:

- First and foremost, the memory required to store  $\mathcal{S}_{[n]}$  upon termination should be approximately the same for all  $n$ . This is fundamental, since memory is the main resource bottleneck in explicit state-space generation. A good measure for this is the *spatial balance*, which we define as the maximum among the ratios between the sizes of the sets of states allocated to any two processes (the closer this is to one, the better):

$$\max_{0 \leq m, n < N} \frac{|\mathcal{S}_{[n]}|}{|\mathcal{S}_{[m]}|} \geq 1.$$

- The communication between processes should be balanced, that is, the number of states received and sent by each process should be approximately the same. Even more importantly, though, this number should be kept as small as possible, so we measure this quantity as the *fraction of cross-arcs* (the smaller the better):

$$0 \leq \frac{\sum_{\mathbf{i} \in \mathcal{S}} |\{\mathbf{j} \in \mathcal{N}(\mathbf{i}) : Proc(\mathbf{i}) \neq Proc(\mathbf{j})\}|}{\sum_{\mathbf{i} \in \mathcal{S}} |\mathcal{N}(\mathbf{i})|} \leq 1$$

- Finally, we would like to achieve a good *temporal balance*: most of the processes should be active most of the time, i.e., they should rarely be idle with an empty unexplored set, waiting to receive states from other processes. A good temporal balance translates into a good speedup. Of course, ignoring possible super-linear speedup effects due to virtual memory, the best we can hope to achieve is an almost linear speedup, i.e., reducing the generation time by a factor of  $N$ .

We mention three possibilities to define *Proc*.

**Static user-provided definition.** In [15] we proposed a static user-provided function based on the idea of hashing some of the state components. In particular, we assumed a Petri net model and the function was of the form

$$Proc(\mathbf{i}) = (\mathbf{i}[0] + p\mathbf{i}[1] + \dots + p^{r-1}\mathbf{i}[r-1]) \bmod N$$

where  $p$  is a prime number (we used 1,013) and  $\mathbf{i}[0]$  through  $\mathbf{i}[r-1]$  are the number of tokens in  $r$  of the places of the Petri net for the given marking (i.e., state). We observed that the selection of the subset of places upon which we base the definition of *Proc* can affect the quality of the resulting partition. However, in all cases, using several “reasonably informed choices” for this selection, we managed to achieve good results on  $N = 6$  processors: the spatial balance ranged from 1.33 to 1.38 in our experiments, while the fraction of cross-arcs ranged from 24% to 61%, and the speedup was up to 4.9 out of an ideal 6.

One advantage of letting the user specify the function is that this makes it possible to pursue specific goals. For example, we observed how, by setting  $p$  to 1 instead of a prime number and selecting  $r$  places where the tokens can only increase or decrease by one at each transition (i.e., event) firing, all cross-arcs can only exist between processes with contiguous indices, i.e., from  $n$  to  $(n - 1) \bmod N$  or  $(n + 1) \bmod N$ . This property could be extremely important if the interconnection medium between the processors were a ring where any number of pairs of adjacent processors can communicate at the same time, since all communications would have to be only between adjacent processors in this case. Another important observation is that the firing in marking  $\mathbf{i}$  of any transition not connected to any of the  $r$  places will lead to a marking  $\mathbf{j}$  with the same owner, i.e.,  $Proc(\mathbf{i}) = Proc(\mathbf{j})$ . This property can be exploited when attempting to minimize the number of cross-arcs, by choosing an appropriately small set of  $r$  places. Of course care must be taken in both situations: in the former case, choosing  $r$  places constituting an *invariant* of the form  $\mathbf{i}[0] + \dots + \mathbf{i}[r - 1] = c$  would be disastrous, since all markings would be owned by process  $c \bmod N$ , while the other processes would be always idle. In the latter situation, choosing too small a set of places, or places with too small a range of possible token populations, could lead to an uneven partition, hence to a poor spatial balance.

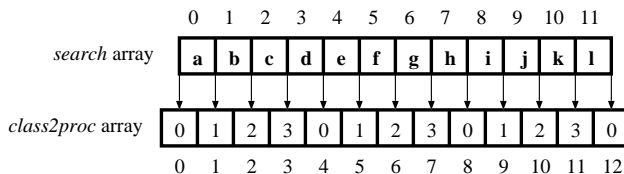
**Dynamic automatic re-mapping.** Relying on a user-provided function requires the user to provide additional information which is related to the solution process, not to the high-level system behavior; in addition, as we just discussed, there is the risk that such a static a-priori definition results in a poor spatial balance, where most of the states are assigned to a small subsets of the processes, or a poor temporal balance, where only a few of the processes are active at any one time.

Thus, we explored a second approach [35] where the definition of  $Proc$  can be dynamically adjusted at run-time, to ensure that both the size of  $\mathcal{U}_{[n]} \cup \mathcal{S}_{[n]}$  and that of  $\mathcal{U}_{[n]}$  alone are balanced across the  $N$  processes, i.e., to ensure good spatial and temporal balance throughout the execution. This is achieved through the use of an intermediate mapping of the states to a large number  $C$  of *classes* (say,  $C = 100N$ ), which are then mapped to processes:

$$Class : \mathcal{S} \rightarrow \{0, \dots, C - 1\} \quad Proc : \{0, \dots, C - 1\} \rightarrow \{0, \dots, N - 1\}.$$

Even when  $C$  is of the order of thousands or tens of thousands, the mapping  $Proc$  can be easily stored by each process as an array of size  $C$  whose entries have value in  $\{0, \dots, N - 1\}$ , so only the mapping  $Class$  is non-trivial. In [35], instead of a user-provided function, we used the lexicographic order between states to define this mapping. The description we give here is a slight variation of this idea.

In a first phase, each of the  $N$  processes independently builds the same *control set* of  $C - 1$  different states found through a depth-first search using discrete-event simulation. More precisely, the  $N$  processes use a pseudo-random number generator initialized with the same seed and start exploring a path through



**Fig. 3.** The definition of the *Class* and *Proc* mappings.

the state space in such a way that, once in state  $\mathbf{i}$ , the next state  $\mathbf{j} \in \mathcal{N}(\mathbf{i})$  to which the model transitions next is chosen with uniform probability (not according to timing specifications in the model, since this might bias the search toward the “likely” states, while, for state-space generation, all states are equally important); then, still without communicating with each other, each process sorts these states into an array “*search*” and allocates a corresponding array “*class2proc*” of size  $C$ , initialized as  $class2proc[i] = i \bmod N$ . Then, the actual state-space generation begins:

- Each process  $n$  initializes  $\mathcal{U}_{[n]}$  with the states in position  $i$  of array *search*, for  $\lfloor C/N \rfloor n \leq i < \lfloor C/N \rfloor (n + 1)$ , then begins the usual iterations.
- When a process  $n$  needs to determine the owner of a state  $\mathbf{j}$ , it performs a binary search for  $\mathbf{j}$  on the array *search*, of which it has a copy.
- If  $\mathbf{j}$  is found, this state is already known to its owner, there is nothing to do.
- Otherwise, the index  $x \in \{0, \dots, C - 1\}$  where the search for  $\mathbf{j}$  failed, i.e.,  $\mathbf{j}$ ’s lexicographic position with respect to the  $C - 1$  elements of the control set, identifies the process owner of  $\mathbf{j}$ :  $m = class2proc[x]$ . Thus, as usual, process  $n$  checks whether  $\mathbf{j}$  is already stored in  $\mathcal{U}_{[n]} \cup \mathcal{S}_{[n]}$ , if  $m = n$ , or sends  $\mathbf{j}$  to process  $m$ , if  $m \neq n$ .

For example, consider Fig. 3, where  $N = 4$  and  $C = 3N$ . If the control set contains states  $\{\mathbf{a}, \dots, \mathbf{l}\}$ ,  $\mathcal{U}_0, \mathcal{U}_1, \mathcal{U}_2$ , and  $\mathcal{U}_3$  will be initialized to  $\{\mathbf{a}, \mathbf{b}, \mathbf{c}\}$ ,  $\{\mathbf{d}, \mathbf{e}, \mathbf{f}\}$ ,  $\{\mathbf{g}, \mathbf{h}, \mathbf{i}\}$ , and  $\{\mathbf{j}, \mathbf{k}\}$ , respectively. Then, during the iterations, if a process searches for a state with a lexicographic position between  $\mathbf{f}$  and  $\mathbf{g}$ , it will determine that this state would belong in position 6 of the control set, if it were in it, but, since it is not, the process owning it is  $class2proc[6]$ , that is, 2.

We now consider the dynamic remapping aspects of this approach. The state-space generation can periodically stop and check the spatial, or temporal, balance, by comparing the size of  $\mathcal{U}_{[n]} \cup \mathcal{S}_{[n]}$ , or  $\mathcal{U}_{[n]}$  alone, over all values of  $n$ . If an unbalance is detected, it can be corrected by dynamically rearranging the allocation of classes to processes, as long as we keep track of the contribution of each class to  $|\mathcal{U}_{[n]}|$  and  $|\mathcal{S}_{[n]}|$ . If the *Class* mapping is fine enough, it should be possible to redefine the *Proc* mapping so that the sets  $\mathcal{U}_{[n]} \cup \mathcal{S}_{[n]} = \{\mathbf{i} : Proc(Class(\mathbf{i})) = n\}$  contain approximately the same number of states, for  $0 \leq n < N$ . Analogously, to focus on temporal balance, we simply need to limit ourselves to the number of unexplored states owned by each process.

The reallocation procedure in [35] follows an approach where each process decides approximately how many states it wants to offload (if it is overloaded) or it is willing to receive (if it is underloaded). Then, a greedy matching algorithm is used to decide which classes to offload and the identity of their new owner; this decision is broadcast to all processes, who can then update the value of the entries in *class2proc* accordingly. Finally, any class that was reallocated from process  $n$  to process  $m$  must exchange owner, i.e., the states in it must be actually transferred.

The tradeoff between frequent checks and a more sensitive triggering condition for a reallocation decision versus less frequent checks and being willing to accept a larger unbalance is clear: the former is more likely to ensure even memory requirements and good processor utilization, but at the cost of larger and more frequent overhead phases during which no useful exploration takes place.

In our experiments [35], we found that dynamic re-balancing was quite beneficial when using 16 processors, while the improvement was minor when using eight processors. We also observed that the overhead as a percentage of the overall runtime was somewhat higher when the goal was temporal balance rather than spatial balance, although it was in any case below 5% except when using very frequent checks. On the other hand, the best speedup on 16 processors was achieved when balancing the temporal load, almost 13 using five re-mapping phases throughout the state-space generation, while the speedup when balancing the spatial load was less than 12.

This dynamic reallocation approach is quite resilient, as long as the classes of equivalence defined by *Class* are fine enough: this is the reason for requiring a value  $C \gg N$ . While we did not experience a problem in our experiments, one or more of these classes could still be too large. In this case, the appropriate step would be to break down these large classes further, splitting them into multiple classes; of course, this requires enlarging the control set, hence extending the *search* and *class2proc* arrays.

**Hashing for state storage and mapping.** In a sequential approach, a hash table is a reasonable alternative to a search tree for storing states and being able to determine whether a state is new or already in  $\mathcal{U} \cup \mathcal{S}$ . The main problem in using a hash table for this application is that the size of  $\mathcal{S}$  cannot be predicted in advance, but this can be remedied by resizing the hash table when the number of collisions grows too much.

The same is true for a distributed approach as well, where we can use a hash table for each of the  $N$  processes, and still require a separate state-to-process mapping. Alternatively, since we already need a hashing function to store a state, we can simply rely on this function to determine the process as well. Conceptually, this requires us to use a single hash table of size  $HN$ , split into  $N$  equal portions that physically reside in the corresponding  $N$  processes. Then, the hash function

$$\text{Hash} : \mathcal{S} \rightarrow \{0, \dots, HN - 1\}$$

can be used by process  $n$  to determine the owner  $m$  of a state  $\mathbf{j}$ :  $m = \lfloor Hash(\mathbf{j})/H \rfloor$ . If  $n = m$ , process  $n$  can use the same function to determine the position  $Hash(\mathbf{j}) \bmod H$  where  $\mathbf{j}$  should be placed in its portion of the hash table.

As one would expect, a completely hash-based approach achieves a reasonably good spatial balance (the authors of [28] report a value of 1.49 for their experiment), but the number of cross-arcs is harder to control (about 50% in [28], using the idea of restricting the definition of the hashing function to a subset of the places of their Petri net model).

## 4.2 Explicit Distributed Markov Chain Generation and Solution

The distributed generation of the entries of matrix  $\mathbf{R}$  follows the same principles as for the sequential approach, with two main differences. First, the mapping  $\Psi$  assigning indices to the states is more complex because it must indicate both the identity of the state and of the process owning it. A reasonable approach is then to define  $N$  per-process mappings

$$\Psi_{[n]} : \mathcal{S}_{[n]} \rightarrow \{0, \dots, |\mathcal{S}_{[n]}| - 1\}$$

so that the overall mapping  $\Psi$  is given by

$$\Psi(\mathbf{i}) = (Proc(\mathbf{i}), \Psi_{[Proc(\mathbf{i})]}(\mathbf{i})).$$

A second, related, difference is the management and storage of the entries, since, if the value  $\Psi_{[n]}(\mathbf{i})$  is computed using either discovery or lexicographic order (as it is normally the case), only process  $n$  can compute its value.

**Storing the transition rate matrix: by rows or by columns?** If we want to generate and store the entries of  $\mathbf{R}$  by rows, i.e., process  $n$  stores all entries  $\mathbf{R}[(n, i), (m, j)]$ , these are the actions process  $n$  must perform when it is exploring state  $\mathbf{i}$  with index  $i = \Psi_{[n]}(\mathbf{i})$ :

```

for each transition from  $\mathbf{i}$  to  $\mathbf{j}$  with rate  $\lambda$ 
  process  $n$  computes  $m = Proc(\mathbf{j})$ 
  if  $m = n$  then
    process  $n$  computes  $j = \Psi_{[n]}(\mathbf{j})$ 
  else
    process  $n$  sends the pair  $\langle n, \mathbf{j} \rangle$  to process  $m$ 
    process  $m$  computes  $j = \Psi_{[m]}(\mathbf{j})$ 
    process  $m$  returns the pair  $\langle m, j \rangle$  to process  $n$ 
  endif
  process  $n$  sets  $\mathbf{R}[(n, i), (m, j)]$  to  $\lambda$ 

```

If instead we want to store the entries by columns, i.e., process  $m$  stores all entries  $\mathbf{R}[(n, i), (m, j)]$ , process  $n$  must perform the following actions when exploring state  $\mathbf{i}$  with index  $i = \Psi_{[n]}(\mathbf{i})$ :

```

for each transition from  $\mathbf{i}$  to  $\mathbf{j}$  with rate  $\lambda$ 

```

```

process  $n$  computes  $m = Proc(\mathbf{j})$ 
if  $m = n$  then
  process  $n$  computes  $j = \Psi_{[n]}(\mathbf{j})$ 
  process  $n$  sets  $\mathbf{R}[(n, i), (m, j)]$  to  $\lambda$ 
else
  process  $n$  sends the tuple  $\langle n, i, \mathbf{j}, \lambda \rangle$  to process  $m$ 
  process  $m$  computes  $j = \Psi_{[m]}(\mathbf{j})$ 
  process  $m$  sets  $\mathbf{R}[(n, i), (m, j)]$  to  $\lambda$ 
endif

```

(in these statements, it might be more correct to say “increments  $\mathbf{R}[(n, i), (m, j)]$  by  $\lambda$ ” instead of “sets  $\mathbf{R}[(n, i), (m, j)]$  to  $\lambda$ ”, since multiple ways to transition from  $\mathbf{i}$  to  $\mathbf{j}$  might exist). Thus, storing  $\mathbf{R}$  by columns is not only preferable in the subsequent numerical solution algorithms, but it also cuts in half the number of logical messages that need to be sent when generating the entries of  $\mathbf{R}$ , since process  $n$  does not require an answer from process  $m$ . We observe that, in [15], we generated both  $\mathcal{S}$  and  $\mathbf{R}$  in a single step. This is possible when the mapping  $\Psi_{[n]}$  is based upon the discovery order (more precisely, the order in which states in  $\mathcal{S}_{[n]}$  become known to process  $n$ ), but it requires the use of more dynamic data structures, such as linked lists, to store the columns of  $\mathbf{R}$ , since the number of nonzero entries process  $n$  might have to store is not known beforehand. Alternatively, we can follow the same two-phase approach discussed for the sequential case, where the  $N$  processes just count the number of entries while generating the state space, in the first phase, and fill these entries in the second phase.

**Block methods for the numerical solution.** For notational simplicity, we define the sub-matrices

$$\mathbf{R}_{[n,m]} =_{\text{df}} \mathbf{R}[\Psi(\mathcal{S}_{[n]}), \Psi(\mathcal{S}_{[m]})].$$

Assuming storage by columns, the entries of  $\mathbf{R}$  are distributed so that process  $m$  stores  $\mathbf{R}_{[n,m]}$ , for  $0 \leq n < N$ . At this point, the distributed numerical solution can begin. Since, as we stressed already, memory is the bottleneck, process  $n$  should store only the portions  $\mathbf{h}_{[n]} =_{\text{df}} \mathbf{h}[\Psi(\mathcal{S}_{[n]})]$  and  $\boldsymbol{\pi}_{[n]} =_{\text{df}} \boldsymbol{\pi}[\Psi(\mathcal{S}_{[n]})]$  of  $\mathbf{h}$  and  $\boldsymbol{\pi}$ , and the same should hold for any other auxiliary vectors required by the solution method, to allow the size of the models that can be solved to scale linearly in  $N$ .

In such a setting, it is natural to employ so-called *block methods* for the numerical solution. For example, the computation performed by process  $n$  in a block-Jacobi iteration is

$$\boldsymbol{\pi}_{[n]}^{new} = \left( \sum_{0 \leq m < N} \boldsymbol{\pi}_{[m]}^{old} \mathbf{R}_{[m,n]} \right) \text{diag}(\mathbf{h}_{[n]}). \quad (2)$$

However, process  $n$  does not store  $\boldsymbol{\pi}_{[m]}^{old}$  for  $m \neq n$ , so this information must be exchanged between processes, and this can be quite time consuming. If one-to-many communication is possible,  $N$  broadcasts are required, where process  $n$

```

 $\pi_{[n]}^{old} \leftarrow \text{"initial guess"};$ 
while "keep doing global synchronizations" do
  BroadcastVector( $\pi_{[n]}^{old}$ );
   $\mathbf{a} \leftarrow \mathbf{0};$  •  $\mathbf{a}$  is a local auxiliary vector of size  $|\mathcal{S}_{[n]}|$ 
  for each  $m \neq n$  do
     $\pi_{[m]}^{old} \leftarrow \text{ReceiveBroadcastVectorFrom}(m);$ 
     $\mathbf{a} \leftarrow \mathbf{a} + \pi_{[m]}^{old} \mathbf{R}_{[m,n]};$ 
  end for;
  while "keep doing local iterations" do
     $\pi_{[n]}^{new} \leftarrow (\pi_{[n]}^{old} \mathbf{R}_{[n,n]} + \mathbf{a}) \text{diag}(\mathbf{h}_{[n]});$ 
     $\pi_{[n]}^{old} \leftarrow \pi_{[n]}^{new};$ 
  end while;
end while;

```

**Fig. 4.** The iterations performed by process  $n$  in a distributed block-Jacobi scheme.

sends  $\pi_{[n]}^{old}$  to all other processes; otherwise,  $N(N-1)$  one-to-one communications are required, for all pairs  $(m, n)$  with  $m \neq n$ .

To reduce the cost of communication as a fraction of the total solution time, the iterative scheme shown in Fig. 4 can be used. In the outer while-loop, each of the  $N$  processes broadcasts its current portion  $\pi_{[n]}^{old}$  of the probability vector and computes the contributions of all the other portions  $\pi_{[m]}^{old}$ , for  $m \neq n$ , to the right-hand-side value in Eq. 2. Then, in the inner while-loop, process  $n$  uses a traditional iteration (we show a variant based on the Jacobi method, but one based on the Gauss-Seidel method would be possible as well), where only the entries of  $\pi_{[n]}^{old}$  are updated, while the ones of  $\pi_{[m]}^{old}$  are effectively kept constant. Thus, there is a tradeoff between more frequent global synchronizations, which imply more communication overhead, vs. less frequent ones, which imply using older data in the local iterations and potentially slowing down the numerical convergence.

We do not discuss the issue of distributed solution further. Our reason for introducing it was simply to illustrate that it is possible to store only vectors of size  $|\mathcal{S}_{[n]}|$  (in the method we presented, four are needed: the current and new iterate, the portion of the holding time vector  $\mathbf{h}_{[n]}$ , and an auxiliary vector  $\mathbf{a}$ ), in addition to the blocks of  $\mathbf{R}$  corresponding to columns in  $\mathcal{S}_{[n]}$  and to the portions of the probability vector received from other processes (these, however, can be “used and discarded on the fly” while computing  $\mathbf{a}$ , so, in principle, impose no additional storage requirements). Distributed solution methods, with either synchronous communication (such as the one we illustrated, where all processes exchange data at the same time) or asynchronous communication, and their convergence properties are an active area of research and a more thorough investigation of the state of the art is beyond the scope of this presentation (see for example [32] and references within).

## 5 Implicit Sequential Solution Approaches

We now turn to *implicit* methods for the storage of the state space and the transition rate matrix. By this we mean methods that exploit certain symmetries, present to some extent in the state space of any system exhibiting some asynchronous behavior, and use data structures that normally require much less than linear space. Another way to state this is that, with explicit methods, we can point at a specific memory location corresponding to a given state in  $\mathcal{S}$  or a given entry of  $\mathbf{R}$ , while, with implicit methods, a state or an entry must be “reconstructed” using information present in several memory locations. As we will see, this results in huge memory savings, and might also result in time savings (as it is normally the case for state-space generation), or it might instead imply an overhead (as is often the case for Markov chain solution).

In all cases, we assume that the model is composed of (or decomposed into)  $K$  sub-models. More precisely, this means that  $\widehat{\mathcal{S}}$  is the cross-product of  $K$  *local state spaces*:  $\widehat{\mathcal{S}} = \mathcal{S}_K \times \cdots \times \mathcal{S}_1$ . The assumption of such a structure in the model is quite reasonable: indeed it is almost always the case that systems, especially complex ones, are built of interacting components. We indicate with  $n_k$  the cardinality of  $\mathcal{S}_k$ , for  $K \geq k \geq 1$ , and stress that  $\mathcal{S}$ , hence  $n_k$ , might be known before exploring  $\mathcal{S}$ , or might become known only afterwards (the latter assumption complicates matters only slightly). From now on, we assume that the local states of  $\mathcal{S}_k$  are stored (once) and *indexed* separately, so that we identify a (global) state with the  $K$ -tuple of its  $K$  local states indices. Thus, a state can be stored in  $\sum_{K \geq k \geq 1} \lceil \log n_k \rceil$  bits. We also identify a state with its *mixed-base value*:

$$\mathbf{i} \equiv (\mathbf{i}_K, \dots, \mathbf{i}_1) = (\dots((\mathbf{i}_K) \cdot n_{K-1} + \mathbf{i}_{K-1}) \cdot n_{K-2} \cdots) \cdot n_1 + \mathbf{i}_1 = \sum_{K \geq k \geq 1} \mathbf{i}_k \cdot n_{k-1:1},$$

where  $n_{k:l} =_{\text{df}} n_k \cdot n_{k-1} \cdots n_l$ , for  $k \geq l$ , and  $n_{k:l} =_{\text{df}} 1$ , for  $k < l$ . The reason for counting the sub-models from  $K$  down to 1 should now be clear: the  $K^{\text{th}}$  index is the “most significant digit”. For example, if  $K = 3$ ,  $|\mathcal{S}_3| = 4$ ,  $|\mathcal{S}_2| = 3$ , and  $|\mathcal{S}_1| = 5$ , the mixed-base value of global state  $(2, 0, 3)$  is  $2 \cdot n_{2:1} + 0 \cdot n_{1:1} + 3 \cdot n_{0:1} = 2 \cdot (3 \cdot 5) + 0 \cdot (5) + 3 = 33$ .

### 5.1 Implicit Sequential State-Space Generation

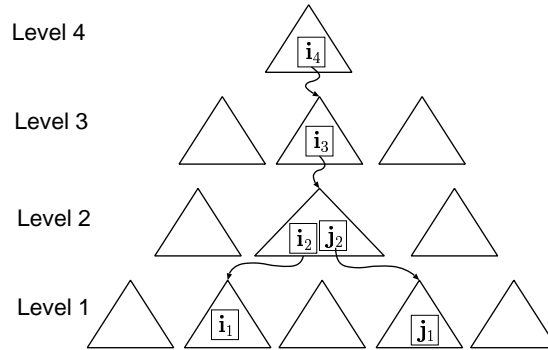
In the area of formal methods, and in particular of model checking [22], *binary decision diagrams* (*BDDs* [5, 6]), have been successfully used to generate and store enormous state spaces [13]. We are going to employ a non-binary version of decision diagrams whose definition [29] can be seen as an extension from the binary to the general discrete world, but can also be reached starting from the explicit data structure we introduced in [19]. We take this second point of view [16, 17, 33].

**An explicit multi-level data structure.** The *multi-level* explicit data structure introduced in [19] can be used in our procedure *ExploreExplicitSequential* to store  $\mathcal{S}$  as it is being built. The idea is that, since a state  $\mathbf{i} \in \mathcal{S}$  is identified with a  $K$ -tuple  $(\mathbf{i}_K, \dots, \mathbf{i}_1)$  of local state indices, we can use  $K$  levels of search trees as shown in Fig. 5, where we assume  $K = 4$ . To search for a state  $\mathbf{i} \equiv (\mathbf{i}_K, \dots, \mathbf{i}_1)$  in the current  $\mathcal{S}$ , we first search for  $\mathbf{i}_K$  in the only search tree at the top level,  $K$ . If  $\mathbf{i}_K$  is found, we follow the corresponding pointer to a tree at level  $K - 1$ , and search  $\mathbf{i}_{K-1}$  in this tree, and so on. The process terminates either if we find  $\mathbf{i}_1$  in the tree at level 1 reached according to the path determined by  $(\mathbf{i}_K, \dots, \mathbf{i}_2)$ , in which case we conclude that  $\mathbf{i}$  is already in  $\mathcal{S}$ , or if we fail to find  $\mathbf{i}_k$  in the corresponding tree at level  $k$ , for some  $k$ ,  $K \geq k \geq 1$ , in which case we conclude that  $\mathbf{i}$  is not yet in  $\mathcal{S}$ , and we know the point at which to start inserting the missing portion  $(\mathbf{i}_k, \dots, \mathbf{i}_1)$  of the state.

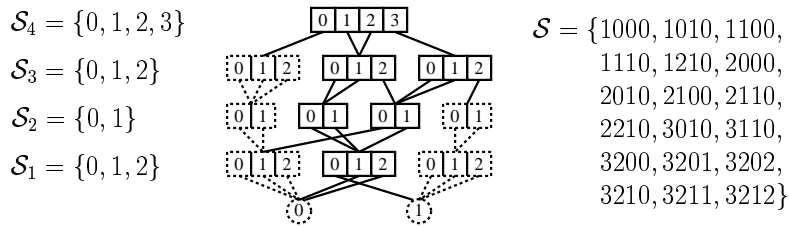
In [19], we pointed out two advantages of this data structure with respect to others used in explicit approaches. First, the storage required for it is mostly due to the bottom level, as long as there is a sufficient fan-out from level to level, i.e., as long as each tree at each level has at least several nodes. The overall number of nodes at the bottom level is exactly the number of states,  $|\mathcal{S}|$ , and to store them we can use, in principle, only integers and pointers of size  $\lceil \log n_1 \rceil$  bits. Assuming a binary tree, this means that, for most practical models, we can store  $\mathcal{S}$  in little over  $3|\mathcal{S}|\lceil \log n_1 \rceil$  bits, as opposed to  $3|\mathcal{S}|\sum_{K \geq k \geq 1} \lceil \log n_k \rceil$  bits. A second property of this data structure results instead in an improved execution time. Assume that we know the pointers  $T_K, \dots, T_1$  to the trees where we found  $\mathbf{i}_K, \dots, \mathbf{i}_1$ , respectively, and that  $\mathbf{j} \in \mathcal{N}(\mathbf{i})$ . To determine whether  $\mathbf{j}$  is in the currently-known portion of the state space, *ExploreExplicitSequential* needs to determine whether there is a path labelled with  $(\mathbf{j}_K, \dots, \mathbf{j}_1)$  in the multi-level data structure. However, the nature of the model might imply that  $\mathbf{j}$  can differ from  $\mathbf{i}$  only in components with index  $k$  or lower (for example, this happens if  $\mathbf{j}$  is reached from  $\mathbf{i}$  through the firing of an event  $e$  that only affects some of the sub-models, a very common situation). When this is the case, the first  $K - k$  components of the two states coincide, that is,  $(\mathbf{i}_K, \dots, \mathbf{i}_{k+1}) = (\mathbf{j}_K, \dots, \mathbf{j}_{k+1})$ , thus the search can start at the tree pointed by  $T_k$ , instead of the top tree. Experimentally, we showed that, on a range of common applications, exploiting this *locality* property can result in considerable execution time reductions, at no extra cost in memory requirements.

**Multi-valued decision diagrams.** The *multi-valued decision diagram* (MDD) definition we adopt is conceptually obtained from the explicit multi-level data structure by recognizing common subtrees at the same level and merging them, in a bottom-up fashion. More formally, a (*quasi-reduced ordered*) MDD [17] is a directed acyclic multi-graph where:

- Nodes are organized into  $K + 1$  *levels*. We write  $\langle k.p \rangle$  to denote a generic node, where  $k$  is the level and  $p$  is a unique index for the nodes at that level. Level  $K$  contains only a single *non-terminal* node  $\langle K.r \rangle$ , the *root*, whereas



**Fig. 5.** The multi-level explicit data structure introduced in [19] ( $K = 4$ ).

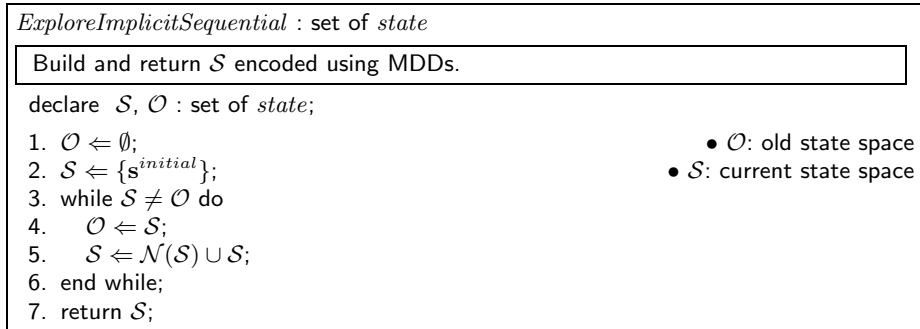


**Fig. 6.** An example MDD and the state space  $\mathcal{S}$  encoded by it.

levels  $K - 1$  through 1 contain one or more non-terminal nodes. Level 0 consists of two *terminal* nodes,  $\langle 0.0 \rangle$  and  $\langle 0.1 \rangle$ .

- A non-terminal node  $\langle k.p \rangle$  has  $n_k$  arcs pointing to nodes at level  $k-1$ . If the  $i^{\text{th}}$  arc, for  $i \in \mathcal{S}_k$ , is to node  $\langle k-1.q \rangle$ , we write  $\langle k.p \rangle[i] = q$ .
- A non-terminal node cannot *duplicate* (i.e., have the same pattern of arcs as) another node at the same level.

Clearly, merging common trees transforms the tree into a directed acyclic graph, but preserves the logic of state search: we can still start from the top level and determine whether a given state belongs to  $\mathcal{S}$  or not, by following the corresponding path and determining whether it leads to  $\langle 0.1 \rangle$  or  $\langle 0.0 \rangle$ . For example, Fig. 6 shows a four-level MDD and the set  $\mathcal{S}$  encoded by it (paths from the root to the node  $\langle 0.1 \rangle$  describe the reachable states). The figure shows arrays being used, instead of search trees: this is more efficient provided a good portion of the array entries is actually used for paths corresponding to reachable states, and it is possible provided that we know a priori the size of each  $\mathcal{S}_k$ , or are willing to use *dynamic arrays*. Paths leading to  $\langle 0.0 \rangle$ , which of course correspond to unreachable states, are present in Fig. 6 because we use arrays, while in the analogous explicit multi-level data structure of Fig. 5 we simply have incomplete paths, because we use search trees. In fact, we showed how, in an actual implementation, it is possible to avoid storing nodes at any level  $k$  corresponding to a logical zero or one, i.e., nodes encoding  $\emptyset$  or  $\mathcal{S}_k \times \dots \times \mathcal{S}_1$ ,



**Fig. 7.** An implicit sequential procedure to generate  $\mathcal{S}$ .

respectively, by reserving the node identifiers  $\langle k.0 \rangle$  and  $\langle k.1 \rangle$  for this purpose (in Fig. 6, these nodes and the arcs emanating from them are shown with dotted lines).

The enormous success of decision diagrams (in the literature, mostly *binary* decision diagrams, or BDDs, have been considered) is not just due to ability of *storing* state spaces in a more compact way, but also of *generating* them more efficiently. This is because we do not use an explicit exploration approach that adds just one state at a time to  $\mathcal{S}$ . Rather, using implicit techniques, we add entire subsets of states in a single operation, as shown in Fig. 7. Considering statement 5, it is clear that we must be able to efficiently compute both the set of states reachable in one step from the currently-know state space,  $\mathcal{N}(\mathcal{S})$ , and their union  $\mathcal{N}(\mathcal{S}) \cup \mathcal{S}$ . Efficient algorithms are known to compute the union of two sets encoded as MDDs, so the focus of much research has been on the encoding and computation of the next-state function.

**Encoding the next-state function.** One popular way to encode the next-state function is to use a decision diagram for it as well, where both the current and the next state are found on a path from the root to  $\langle 0.1 \rangle$  (thus the diagram has  $2K$  levels, usually interleaved for greater efficiency). Rather than following this traditional approach, however, we introduce a different one [16, 17, 33] that is both much more efficient, and also strongly related to the implicit encoding of the transition rate matrix, which we will consider in Sect. 5.2.

The idea is based on the fact that many systems exhibit what can be called a *globally-asynchronous locally-synchronous behavior*, that is, most events are enabled by and affect only a small subset of the  $K$  sub-models, which must then be “synchronized”, while distinct events can occur concurrently and asynchronously in different parts of the system. We then require that, for each event  $e$ , its next-state function  $\mathcal{N}_e$  can be written as the cross-product of  $K$  local functions:

$$\mathcal{N}_e = \mathcal{N}_{K,e} \times \cdots \times \mathcal{N}_{1,e}, \quad (3)$$

where  $\mathcal{N}_{k,e} : \mathcal{S}_k \rightarrow 2^{\mathcal{S}_k}$ . This *product-form* requirement is quite natural for two reasons. First, many modeling formalisms satisfy it (for example, we showed in

[21] that any Petri net model conforms to this behavior for any partition of its places). Second, if a given model does not respect the product-form behavior, we can always coarsen the sub-models or refine  $\mathcal{E}$  so that it does (in the limit, this results in having a single model, i.e.,  $K = 1$ , or defining a different event for each state-to-state transition, i.e.,  $\mathcal{E} = \mathcal{A}$ , but this does not seem to occur in practice).

When an event  $e$  is independent of a sub-model (or level)  $k$ ,  $\mathcal{N}_{k,e}$  is the identity, that is:

$$\forall \mathbf{i}_k \in \mathcal{S}_k, \quad \mathcal{N}_{k,e}(\mathbf{i}_k) = \{\mathbf{i}_k\}. \quad (4)$$

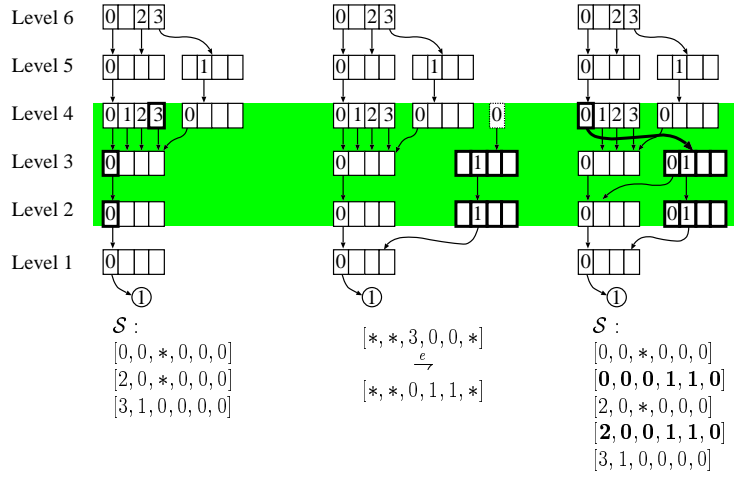
When this is not the case, we say instead that  $e$  *depends* on level  $k$ . We let  $First(e)$  and  $Last(e)$  be the first and last levels on which event  $e$  depends and, as we shall see, one of the goals of a good decomposition is to have events span few levels, that is, to define sub-models so that the range  $First(e) - Last(e) + 1$  is small with respect to  $K$ . In particular, events  $e$  such that  $First(e) = Last(e) = k$  are said to be *local*, while those where  $First(e) > Last(e)$  are said to be *synchronizing*. Note that the local events at a given level can be merged into a single *macro-event*  $\lambda_k$  without violating the product-form requirement, since we can define

$$\mathcal{N}_{\lambda_k} = \mathcal{N}_{K,\lambda_k} \times \cdots \times \mathcal{N}_{1,\lambda_k}$$

where  $\mathcal{N}_{k,\lambda_k} = \bigcup_{e:First(e)=Last(e)=k} \mathcal{N}_{k,e}$ , where, again, the union is applied to the value of the functions, i.e.,  $\mathcal{N}_{k,\lambda_k}(\mathbf{i}_k) = \bigcup_{e:First(e)=Last(e)=k} \mathcal{N}_{k,e}(\mathbf{i}_k)$ , while  $\mathcal{N}_{l,\lambda_k}(\mathbf{i}_l) = \{\mathbf{i}_l\}$  for  $l \neq k$  and  $\mathbf{i}_l \in \mathcal{S}_l$ . From now on, we assume that the set of events  $\mathcal{E}$  has been redefined to reflect this merging.

Under these conditions, the next-state function  $\mathcal{N}$  is fully captured by  $(L + 1)K$  boolean matrices  $\mathbf{B}_{k,e}$ , where  $L$  is the number of synchronizing events; indeed, in a good decomposition, a majority of these matrices are the identity, so they do not need to be stored explicitly. Also, for any matrix  $\mathbf{B}_{k,e}$  that must be actually stored, we can use a sparse row-wise data structure. Then, given  $\mathbf{i}_k$ , we can retrieve the set  $\mathcal{N}_{k,e}(\mathbf{i}_k)$  in time and memory proportional to  $|\mathcal{N}_{k,e}(\mathbf{i}_k)|$ , which, in practical applications, is much smaller than  $n_k$ . In other words, the storage required to encode  $\mathcal{N}$  is a negligible portion of the overall storage requirements, and the time required to build its encoding is also very small.

**Generating  $\mathcal{S}$  as the fixed-point of  $\mathcal{N}$ .** Using our encoding of the next-state function, we are able to generate enormous state spaces efficiently using a modification of the algorithm shown in Fig. 7. Specifically, instead of computing  $\mathcal{N}(\mathcal{S})$  at each iteration, we compute several “lighter” next-state functions, one per event, exploiting the concept of *event locality* to limit our computation to nodes at the appropriate levels. In other words, we don’t even need to explore the firing an event  $e$  at level  $k$ , if  $k > First(e)$  or  $Last(e) > k$ . Fig. 8 illustrates, as an example, the firing of an event  $e$  dependent on levels 4, 3, and 2 only, starting from the state space shown on the left (nodes encoding  $\emptyset$  and arcs pointing to them are omitted for clarity). Event  $e$  is enabled only when the the sub-models



**Fig. 8.** Firing an event  $e$  dependent only on levels 4, 3, and 2.

4, 3, and 2 are in the local states with index 3, 0, and 0, and its firing changes these local states to 0, 1, and 1, respectively. The state space after this firing is shown on the right. Note that a single firing adds the two states shown in boldface.

Another related improvement is the use of an efficient *iteration strategy*, where we explore the firing of events in a specific order: starting from an initial set of states (in our case,  $\{\mathbf{s}^{initial}\}$ ), we fire exhaustively any event  $e$  for which  $First(e) = 1$  (of course, only the local macro-event  $\lambda_1$  satisfies this requirement), then we fire exhaustively any event  $e$  for which  $First(e) = 2$  (this includes both the local macro-event  $\lambda_2$  and any event synchronizing level 1 and level 2) and, for any new node (hence for the corresponding states) that these firings might create at level 1, we again fire exhaustively any event  $e$  for which  $First(e) = 1$ , and so on. Once we reach level  $K$  and exhaustively fire any event at level  $K$ , plus any event at lower levels on any newly created node, the process is completed: our MDD is *saturated* and it encodes exactly  $\mathcal{S}$ .

We introduced this idea of saturating the nodes of the MDD during fixed-point exploration in [17], and showed how, paired with locality, it can reduce the memory and time requirements to generate the state space of practical models by orders of magnitude. For example, with our implementation in SMART [18], we could generate the state space corresponding to the famous dining philosophers problem, with 1,000 philosophers ( $\mathcal{S}$  contains almost  $10^{627}$  states!), in less than one second on an 800 MHz Pentium workstation. This represents an enormous improvement with respect to more conventional symbolic methods based on a BDD encoding of both the state space and the next state function [37, 38, 39, 43]. Furthermore, the benefits increase with the height of the MDDs (the number of levels  $K$ ), since, when decomposing practical models, the range of levels on which

an event depends is going to be mostly a small constant, regardless of the value of  $K$ .

## 5.2 Implicit Sequential Markov Chain Generation and Solution

Armed with our understanding of the structure imposed on the state space by the logic product-form behavior, we can now discuss an analogous concept for the description of the transition rate matrix of the CTMC underlying a high-level model decomposed into sub-models. It is interesting to note, however, that work on this *Kronecker-based description* predates that on MDDs; in fact, it was our inspiration for it, rather than the other way around.

The use of Kronecker (also called *tensor*) algebra [4, 23, 27] for the description of a transition rate matrix is over twenty years old [2], but its real impact was realized when this approach began being applied to high-level models, first by Plateau and Stewart on *Synchronized Automata Networks* [26, 41, 42], then by Donatelli on the more general *Superposed Stochastic Automata* [25] and by Buchholz and Kemper [7, 8, 9, 12] on several classes of hierarchical formalisms, including queueing networks and their variants. For a brief description of the Kronecker product “ $\otimes$ ” and Kronecker sum “ $\oplus$ ” operators, see the Appendix.

**Kronecker description of the transition rate matrix.** We have seen that the transition rate from state  $\mathbf{i}$  to state  $\mathbf{j}$  can be expressed in a per-event fashion:

$$Rate(\mathbf{i}, \mathbf{j}) = \sum_{e \in \mathcal{E}} Rate_e(\mathbf{i}, \mathbf{j}).$$

To apply our implicit description techniques, we only need to assume a product-form requirement analogous to that of Eq. 3:

$$Rate_e = Rate_{K,e} \cdots Rate_{1,e},$$

where each non-negative function  $Rate_{k,e} : \mathcal{S}_k \times \mathcal{S}_k \rightarrow \mathbb{R}$  expresses the (multiplicative) contribution of sub-model  $k$  to the rate of event  $e$  (of course, dimensionally, only their product is a rate).

As it should be expected, we require that  $Rate_{k,e}(\mathbf{i}_k, \mathbf{j}_k) = 0$  iff  $\mathbf{j}_k \notin \mathcal{N}_{k,e}(\mathbf{i}_k)$  and we say that event  $e$  is (stochastically) independent of level  $k$  if  $Rate_{k,e}$  is the identity, i.e.,  $Rate_{k,e}(\mathbf{i}_k, \mathbf{j}_k) = 0$  when  $\mathbf{i}_k \neq \mathbf{j}_k$ , 1 when  $\mathbf{i}_k = \mathbf{j}_k$ . Note that the logic independence of Eq. 4 is a necessary but not sufficient condition for this new definition of independence, since the local state of a sub-model  $k$  could affect the timing of an event  $e$  but not its enabling or its effect, i.e., we might have  $\mathcal{N}_{k,e}(\mathbf{i}_k) = \{\mathbf{i}_k\}$  for all  $\mathbf{i}_k \in \mathcal{S}_k$ , but the (positive) value of  $Rate_{k,e}(\mathbf{i}_k, \mathbf{i}_k)$  could nevertheless depend on the particular local state  $\mathbf{i}_k$ . Also, we say that  $e$  is a local event for level  $k$  iff  $k$  is the only level for which  $Rate_{k,e}$  is not the identity, and again we merge all such events into a macro event  $\lambda_k$ .

We can then define the matrices  $\mathbf{W}_{k,e}$  as the encoding of  $Rate_{k,e}$ , for any synchronizing event  $e$ , which we assume indexed from 1 to  $L$ , and the matrices

$\mathbf{R}_k$  as the encoding of  $Rate_{k,\lambda_k}$ :

$$\mathbf{W}_{k,e}[\mathbf{i}_k, \mathbf{j}_k] = Rate_{k,e}(\mathbf{i}_k, \mathbf{j}_k) \quad \mathbf{R}_k[\mathbf{i}_k, \mathbf{j}_k] = Rate_{k,\lambda_k}(\mathbf{i}_k, \mathbf{j}_k),$$

and write

$$\mathbf{R} = \widehat{\mathbf{R}}[\mathcal{S}, \mathcal{S}] = \sum_{e \in \mathcal{E}} \widehat{\mathbf{R}}_e[\mathcal{S}, \mathcal{S}] = \left( \bigoplus_{K \geq k \geq 1} \mathbf{R}_k + \sum_{e=1}^L \bigotimes_{K \geq k \geq 1} \mathbf{W}_{k,e} \right) [\mathcal{S}, \mathcal{S}], \quad (5)$$

where we recall that, when indexing the “potential” matrix  $\widehat{\mathbf{R}}$ , a global state  $\mathbf{i}$  is interpreted as its mixed-base integer value when used as a matrix or vector index. An analogous expression exists for  $\mathbf{Q}$ , the infinitesimal generator.

In other words, we can express the (huge) transition rate matrix  $\mathbf{R}$  as the sub-matrix corresponding to the reachable states  $\mathcal{S}$  of a (possibly even larger) matrix that can be expressed through Kronecker operators applied to  $(L+1)K$  small real matrices: a huge memory saving. Furthermore, as in the logic case, many of these matrices will be the identity in practical applications. However, unlike the results we discussed for decision diagrams, iterative numerical methods multiply the implicitly-encoded matrix  $\mathbf{R}$  by “explicit” probability vectors, which are now the main memory bottleneck, so we must examine how this encoding affects the run-time efficiency.

**Potential vs. actual state space, row vs. column access.** One potential source of overhead when using Eq. 5 to encode  $\mathbf{R}$  is that we need to consider a sub-matrix of a Kronecker expression. Initial proposals [25, 41] used algorithms that can (almost) ignore the difference between the potential state space  $\widehat{\mathcal{S}}$  and the actual state space  $\mathcal{S}$ . This is possible if we access  $\mathbf{R}$ , or more precisely  $\widehat{\mathbf{R}}$ , by rows, since, by definition, an unreachable state  $\mathbf{j}$  cannot be reached from a reachable state  $\mathbf{i}$ , that is,  $\widehat{\mathbf{R}}[\mathcal{S}, \widehat{\mathcal{S}} \setminus \mathcal{S}] = \mathbf{0}$ . Unfortunately, the converse is not true, that is,  $\widehat{\mathbf{R}}[\widehat{\mathcal{S}} \setminus \mathcal{S}, \mathcal{S}]$  is not necessarily zero. The Jacobi iteration can be rewritten to enforce access by rows:

```

for each  $\mathbf{i} \in \widehat{\mathcal{S}}$  such that  $\widehat{\boldsymbol{\pi}}^{old}[\mathbf{i}] > 0$  do
  for each  $\mathbf{j} \in \widehat{\mathcal{S}}$  such that  $\widehat{\mathbf{R}}[\mathbf{i}, \mathbf{j}] > 0$  do
     $\widehat{\boldsymbol{\pi}}^{new}[\mathbf{j}] \leftarrow \widehat{\boldsymbol{\pi}}^{new}[\mathbf{j}] + \widehat{\boldsymbol{\pi}}^{old}[\mathbf{i}] \widehat{\mathbf{R}}[\mathbf{i}, \mathbf{j}] \widehat{\mathbf{h}}[\mathbf{j}]$ ;
  end for;
end for;

```

• access row  $\mathbf{i}$  of  $\widehat{\mathbf{R}}$

where  $\widehat{\mathbf{h}}$ ,  $\widehat{\boldsymbol{\pi}}^{old}$ , and  $\widehat{\boldsymbol{\pi}}^{new}$  are vectors of size  $|\widehat{\mathcal{S}}|$ , indexed by the mapping

$$\widehat{\boldsymbol{\psi}} : \widehat{\mathcal{S}} \rightarrow \{0, \dots, |\widehat{\mathcal{S}}| - 1\},$$

which is simply the mixed-based value of the state. This Jacobi iteration moves the probability mass only to states reachable from those having an initial probability mass so, if  $\widehat{\boldsymbol{\pi}}^{old}$  is properly initialized according to the initial state, only

entries of  $\widehat{\boldsymbol{\pi}}^{old}$  corresponding to states in  $\mathcal{S}$  will ever become positive, hence only entries in  $\widehat{\mathbf{R}}[\mathcal{S}, \mathcal{S}]$  are used.

The disadvantages of such a “potential” approach, however, are numerous:  $\widehat{\mathbf{h}}$ ,  $\widehat{\boldsymbol{\pi}}^{old}$ , and  $\widehat{\boldsymbol{\pi}}^{new}$  may require much more storage than their “actual” siblings  $\mathbf{h}$ ,  $\boldsymbol{\pi}^{old}$ , and  $\boldsymbol{\pi}^{new}$ ; to make things worse, the entire vector  $\widehat{\boldsymbol{\pi}}^{new}$  is used as an accumulator, so it should in principle be allocated using a higher-precision floating point type, while, with access by columns, only one scalar accumulator is required; finally, the approach just described can be used for the Jacobi method (or the even slower Power method), but the preferred Gauss-Seidel essentially requires access by columns.

Another potential overhead is the actual computation of the entry  $\widehat{\mathbf{R}}[\mathbf{i}, \mathbf{j}]$  using the Kronecker expression of Eq. 5. With vectors of size  $|\widehat{\mathcal{S}}|$ , the indexing function  $\widehat{\Psi}$  is indeed easy to define and compute, but it can still involve an overhead factor  $O(K)$ ; an analogous overhead is potentially implied also by the multiplications of the  $K$  entries in the appropriate  $\mathbf{W}_{k,e}$  matrices (in most cases, only one event causes an entry of  $\widehat{\mathbf{R}}$  to be positive). Algorithms that attempt to amortize these computations have been introduced [10]; the best ones in practice appear to be based on an interleaving of the row and column indices of the  $K$  matrices, which imply an access pattern that is neither by rows nor by columns: again, they allow us to employ methods such as Power and Jacobi, but they precludes us from using Gauss-Seidel. It should also be mentioned that, in the case where the matrices  $\mathbf{W}_{k,e}$  are quite dense, the *shuffle* algorithm is extremely efficient: for full matrices, the complexity of computing  $\mathbf{x} \cdot \bigotimes_{K \geq k \geq 1} \mathbf{A}_k$  is  $n_{K:1} \cdot \sum_{K \geq k \geq 1} n_k$  instead of  $(n_{K:1})^2$  [26]; however, the matrices  $\mathbf{W}_{k,e}$  are usually *ultra-sparse*, i.e., they often have around one nonzero entry per row on average, so these savings might not be realized in practice [10].

More recent implementations based on the actual state space that access  $\widehat{\mathbf{R}}$  by column have been proposed. These allocate only vectors of size  $|\mathcal{S}|$ , but require a more complex indexing scheme that maps a state  $\mathbf{i}$  to the index  $i = \Psi(\mathbf{i}) \in \{0, \dots, |\mathcal{S}| - 1\}$ . Furthermore, in the case of access by columns, for a given reachable destination state  $\mathbf{j}$ , a source state  $\mathbf{i}$  may be unreachable, in which case  $\Psi(\mathbf{i})$  must return a null value, to signal that the entry  $\widehat{\mathbf{R}}[\mathbf{i}, \mathbf{j}]$  is to be ignored. The lexicographic order mentioned in Sect. 3.1 is an excellent candidate to use for  $\Psi$ , but, if used in a simplistic way, it involves an overhead factor  $O(\log |\mathcal{S}|)$ , since each state  $\mathbf{j}$  must be searched in  $\mathcal{S}$  to determine its index  $\Psi(\mathbf{j})$ . Again, the best algorithm appears to be the one using interleaving of the  $\mathbf{i}$  and  $\mathbf{j}$  indices to maximize the amortization of these searches; when combined with the multi-level data structure of [19] or, even better, the MDDs of [16, 17, 33], the overhead is only  $O(\log n_1)$ . For a thorough discussion of algorithms based on either the potential or the actual state space, see [10].

**Matrix diagrams.** In [20], we introduced an alternative method to encode the transition rate matrix  $\mathbf{R}$  of a model structured into  $K$  sub-models. While related to the expression of Eq. 5, this new method has several advantages over a straightforward Kronecker encoding that uses  $(L + 1)K$  matrices.

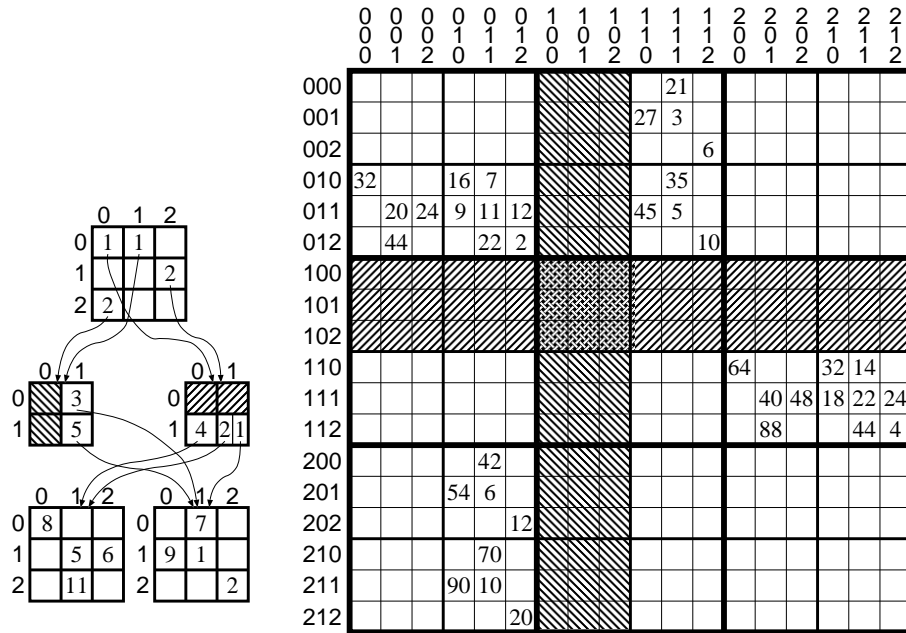


Fig. 9. An example of matrix diagram and the matrix encoded by it.

The definition of matrix diagrams is quite similar to that of MDDs, but there are several fundamental differences. First, the nodes of a matrix diagram are matrices, not arrays, since we are encoding a two-dimensional matrix, not a set. Second, the entries of the matrices are lists of pairs, each pair containing a real number and a pointer to a node at the next level, since we are encoding a real value along a path, not just the existence of a path. Finally, as defined, matrix diagrams are not canonical; this is not a problem since the only operation we need to perform on them when solving the CTMC is a vector-matrix multiplication.

Formally, a (nonzero, reduced) *matrix diagram* is a directed acyclic multi-graph where:

- Nodes are organized into  $K$  levels. We write  $\langle k.p \rangle$  to denote a generic node, where  $k$  is the level and  $p$  is a unique index for the nodes at that level. Level  $K$  contains only a single node  $\langle K.r \rangle$ , the *root*, whereas levels  $K-1$  through 1 contain one or more nodes reachable on a path from the root.
- A node  $\langle k.p \rangle$  is a  $n_k \times n_k$  matrix; for  $K \geq p > 1$ , the entry  $\langle k.p \rangle[i, j]$  is a list of pairs of the form  $(v, q)$ , where  $v$  is a real number and  $\langle k-1.q \rangle$  is a node, while, for  $k = 1$ , the entry is just a real number  $v$ .
- No two elements of a list  $\langle k.p \rangle[i, j]$  can have the same value or pointer.
- No two nodes at the same level *duplicate* (i.e., have the same pattern of entries) each other.

Such a data structure can encode a real matrix as follows: the value of the entry in position  $(\mathbf{i}, \mathbf{j})$  is given by the sum over all paths of the products of the form  $v_K \cdots v_1$ , where  $v_k$  is a real value found in the list in position  $(\mathbf{i}_k, \mathbf{j}_k)$  of the matrix at level  $k$  for the path. For example, consider the matrix diagram shown in Fig. 9 on the left, and the corresponding matrix encoded by it, on the right; the value of the entry in position  $((0, 1, 1), (1, 1, 0))$  is given by the product of the entries found in position  $(0, 1)$  of the one matrix at level 3, in position  $(1, 1)$  of the first matrix at level 2, and in position  $(1, 0)$  of the second matrix at level 1:  $45 = 1 \cdot 5 \cdot 9$  (only one path needs to be considered because each of the lists corresponding to the row and column indices contains only one entry); the value of the entry in position  $((1, 1, 1), (2, 1, 1))$  is instead obtained by summing the products corresponding to two paths:  $22 = 2 \cdot 2 \cdot 5 + 2 \cdot 1 \cdot 1$ .

In [20] we showed several important advantages of matrix diagrams over a traditional matrix-based Kronecker representation:

- A single data structure encodes the entire matrix, unlike the traditional representation which relies on summing, for each event, the Kronecker product of  $K$  matrices. This saves a fair amount of indexing overhead.
- If the row and column index sets of the matrix being represented are strict subsets of the cross-products of the row and column index sets at each level (as it is often the case for  $\mathbf{R}$  vs.  $\widehat{\mathbf{R}}$ ), a matrix diagram can encode exactly the intended matrix. For example, in Fig. 9, any state of the form  $(1, 0, \mathbf{i}_1)$  is unreachable, as indicated by the greyed portions in the large matrix; this is reflected, in the matrix diagram, by the greyed portions in the matrices at level 2: indeed, these can be stored as a  $2 \times 1$  and a  $1 \times 2$  matrix, respectively, provided the information about their row and column index set is preserved. This can save both memory and, especially, execution time, since it allows us to operate with “actual” vectors while having the same low overhead enjoyed when using “potential” vectors.
- Operation caches can be used to avoid recomputing partial multiplications of the real values encountered along a path during an iteration of the numerical solution method.
- Finally, because of the ability to represent the desired matrix and not a supermatrix of it, matrix diagrams allow the numerical methods to perform a by-column access without having to worry about the spurious nonzero entries in  $\widehat{\mathbf{R}}[\widehat{\mathcal{S}} \setminus \mathcal{S}, \mathcal{S}]$ .

Thanks to the above advantages, experimental results show that a Kronecker implementation based on matrix diagrams has a very low overhead both in terms of memory and execution time. For example, in [20], we solved models with tens of millions of states and hundreds of millions of nonzero entries on a simple Pentium workstation, while explicit solutions were restricted to models about one order of magnitude smaller, and traditional matrix-based Kronecker implementations required either two or three times as much time when using Gauss-Seidel (due to the overhead in by-column access), or approximately twice memory when using Jacobi (due to the need to store both the “old” and the “new” iterates).

## 6 Conclusion

Formal mathematical models for both logic and temporal analysis of nondeterministic systems are often the most desirable tools, but their solution can be infeasible due to time and, especially, memory limitations.

Parallel and distributed algorithms will continue to play a role in helping to cope with the state-space explosion problem, but they can at best increase the size of the state spaces that can be tackled linearly in the number of resources (processors, memory) we are willing to employ for the solution process.

A more fundamental paradigm shift is achieved using implicit methods based on decision diagrams, Kronecker algebra, and similar approaches that exploit the complex symmetries often present in systems exhibiting globally-asynchronous locally-synchronous behavior. The success of model checking is proof of their effectiveness for the logic modeling aspects but, for Markov models, these approaches are only now beginning to arise the enjoy widespread acceptance.

Our prediction for the area of logic and stochastic models is that an important future research direction will focus on the use of distributed algorithms based on decision diagrams and implicit approaches in general, combined with approximations that avoid the memory bottleneck due to explicit vectors when computing the numerical solution of the underlying stochastic processes.

## References

- [1] S. C. Allmaier, M. Kowarschik, and G. Horton. State space construction and steady-state solution of GSPNs on a shared-memory multiprocessor. In *Proc. 7th Int. Workshop on Petri Nets and Performance Models (PNPM'97)*, pages 112–121, St. Malo, France, June 1997. IEEE Comp. Soc. Press.
- [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] J. A. Bergstra, A. Ponse, and S. A. Smolka. *Handbook of Process Algebra*. Elsevier Science, 2000.
- [4] J. W. Brewer. Kronecker products and matrix calculus in system theory. *IEEE Trans. Circ. and Syst.*, CAS-25:772–781, Sept. 1978.
- [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. Numerical solution methods based on structured descriptions of Markovian models. In G. Balbo and G. Serazzi, editors, *Computer performance evaluation*, pages 251–267. Elsevier Science Publishers B.V. (North-Holland), 1991.
- [8] P. Buchholz. Hierarchical Markovian models – Symmetries and Reduction. In *Modelling Techniques and Tools for Computer Performance Evaluation*. Elsevier Science Publishers B.V. (North-Holland), 1992.
- [9] P. Buchholz. A class of hierarchical queueing networks and their analysis. *Queueing Systems.*, 15:59–80, 1994.

- [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.*, 12(3):203–222, Summer 2000.
- [11] P. Buchholz, M. Fischer, and P. Kemper. Distributed steady state analysis using Kronecker algebra. In B. Plateau, W. J. Stewart, and M. Silva, editors, *Numerical Solution of Markov Chains*, pages 76–95. Prentice Hall, Zaragoza, Zaragoza, Spain, Sept. 1999.
- [12] P. Buchholz and P. Kemper. Numerical analysis of stochastic marked graphs. In *Proc. 6th Int. Workshop on Petri Nets and Performance Models (PNPM'95)*, pages 32–41, Durham, NC, Oct. 1995. IEEE Comp. Soc. Press.
- [13] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking:  $10^{20}$  states and beyond. In *Proceedings, Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 428–439, Philadelphia, Pennsylvania, 4–7 June 1990. IEEE Computer Society Press.
- [14] S. Caselli, G. Conte, and P. Marenzoni. Parallel state space exploration for GSPN models. In G. De Michelis and M. Diaz, editors, *Application and Theory of Petri Nets 1995 (Proc. 16th Int. Conf. on Applications and Theory of Petri Nets, Turin, Italy)*, Lecture Notes in Computer Science 935, pages 181–200. Springer-Verlag, June 1995.
- [15] G. Ciardo, J. Gluckman, and D. Nicol. Distributed state-space generation of discrete-state stochastic models. *INFORMS J. Comp.*, 10(1):82–93, 1998.
- [16] G. Ciardo, G. Luetzgen, and R. Siminiceanu. Efficient symbolic state-space construction for asynchronous systems. In M. Nielsen and D. Simpson, editors, *Application and Theory of Petri Nets 2000 (Proc. 21th Int. Conf. on Applications and Theory of Petri Nets, Aarhus, Denmark)*, Lecture Notes in Computer Science 1825, pages 103–122. Springer-Verlag, June 2000.
- [17] G. Ciardo, G. Luetzgen, 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)*. Springer-Verlag, Apr. 2001. To appear.
- [18] G. Ciardo and A. S. Miner. SMART: Simulation and Markovian Analyzer for Reliability and Timing. In *Proc. IEEE International Computer Performance and Dependability Symposium (IPDS'96)*, page 60, Urbana-Champaign, IL, USA, Sept. 1996. IEEE Comp. Soc. Press.
- [19] 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.
- [20] 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.
- [21] 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.
- [22] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.
- [23] M. Davio. Kronecker products and shuffle algebra. *IEEE Trans. Comp.*, C-30:116–125, Feb. 1981.

- [24] E. W. Dijkstra, W. H. Feijen, and A. Van Gasteren. Derivation of a termination detection algorithm for a distributed computation. *Inf. Proc. Letters*, 16:217–219, June 1983.
- [25] S. Donatelli. Superposed Stochastic Automata: a class of stochastic Petri nets with parallel solution and distributed state space. *Perf. Eval.*, 18:21–26, 1993.
- [26] P. Fernandes, B. Plateau, and W. J. Stewart. Efficient descriptor-vector multiplication in stochastic automata networks. *Journal of the ACM*, 45(3):381–414, 1998.
- [27] A. Graham. *Kronecker Products and Matrix Calculus: with Applications*. Halsted Press / John Wiley, New York, 1981.
- [28] B. R. 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.
- [29] T. Kam, T. Villa, R. Brayton, and A. Sangiovanni-Vincentelli. Multi-valued decision diagrams: theory and applications. *Multiple-Valued Logic*, 4(1–2):9–62, 1998.
- [30] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL: Status and developments. In O. Grumberg, editor, *9th Int. Conf. on Computer Aided Verification (CAV '97)*, volume 1254 of *Lecture Notes in Computer Science*, pages 456–459. Springer-Verlag, June 1997.
- [31] P. Marenzoni, S. Caselli, and G. Conte. Analysis of large GSPN models: a distributed solution tool. In *Proc. 7th Int. Workshop on Petri Nets and Performance Models (PNPM'97)*, pages 122–131, St. Malo, France, June 1997. IEEE Comp. Soc. Press.
- [32] V. Migallón, J. Penadés, and D. B. Szyld. Experimental study of parallel iterative solutions of Markov chains with block partitions. In B. Plateau, W. J. Stewart, and M. Silva, editors, *Numerical Solution of Markov Chains*, pages 96–110. Prentice Hall, Zaragoza, Zaragoza, Spain, Sept. 1999.
- [33] 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.
- [34] T. Murata. Petri Nets: properties, analysis and applications. *Proc. of the IEEE*, 77(4):541–579, Apr. 1989.
- [35] D. Nicol and G. Ciardo. Automated parallelization of discrete state-space generation. *J. Par. and Distr. Comp.*, 47:153–167, 1997.
- [36] D. M. Nicol. Non-committal barrier synchronization. *Parallel Computing*, 21:529–549, 1995.
- [37] E. Pastor and J. Cortadella. Efficient encoding schemes for symbolic analysis of Petri nets. In *Proc. Design Automation and Test in Europe*, Feb. 1998.
- [38] 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.
- [39] 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.

- [40] S. Pissanetzky. *Sparse Matrix Technology*. Academic Press, 1984.
- [41] 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.
- [42] W. J. Stewart, K. Atif, and B. Plateau. The numerical solution of stochastic automata networks. *Eur. J. of Oper. Res.*, 86:503–525, 1995.
- [43] K. Varpaaniemi, J. Halme, K. Hiekkänen, and T. Pyssysalo. PROD reference manual. Technical Report B13, Helsinki Univ. of Technology, 1995.

## A A brief introduction to Kronecker operators

Given  $K$  real matrices  $\mathbf{A}_k \in \mathbb{R}^{n_k \times n_k}$ , for  $K \geq k \geq 1$ , their *Kronecker product*

$$\mathbf{A} = \mathbf{A}_K \otimes \mathbf{A}_{K-1} \otimes \cdots \otimes \mathbf{A}_1 = \bigotimes_{K \geq k \geq 1} \mathbf{A}_k \in \mathbb{R}^{n_{K:1} \times n_{K:1}}$$

is defined by

$$\mathbf{A}[\mathbf{i}, \mathbf{j}] = \mathbf{A}_K[\mathbf{i}_K, \mathbf{j}_K] \cdots \mathbf{A}_1[\mathbf{i}_1, \mathbf{j}_1],$$

where we use the mixed-base indexing scheme

$$\mathbf{i} \equiv (\mathbf{i}_K, \dots, \mathbf{i}_1) = (\dots((\mathbf{i}_K) \cdot n_{K-1} + \mathbf{i}_{K-1}) \cdot n_{K-2} \cdots) \cdot n_1 + \mathbf{i}_1 = \sum_{K \geq k \geq 1} \mathbf{i}_k \cdot n_{k-1:1},$$

while the *Kronecker sum* of the same matrices is simply

$$\bigoplus_{K \geq k \geq 1} \mathbf{A}_k = \sum_{K \geq k \geq 1} \mathbf{I}_{n_{K:k+1}} \otimes \mathbf{A}_k \otimes \mathbf{I}_{n_{k-1:1}} \in \mathbb{R}^{n_{K:1} \times n_{K:1}}.$$

For example, given  $\mathbf{A} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$  and  $\mathbf{B} = \begin{bmatrix} r & s & t \\ u & v & w \\ x & y & z \end{bmatrix}$ , we have

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} a\mathbf{B} & b\mathbf{B} \\ c\mathbf{B} & d\mathbf{B} \end{bmatrix} = \begin{bmatrix} ar & as & at & | & br & bs & bt \\ au & av & aw & | & bu & bv & bw \\ ax & ay & az & | & bx & by & bz \\ \hline cr & cs & ct & | & dr & ds & dt \\ cu & cv & cw & | & du & dv & dw \\ cx & cy & cz & | & dx & dy & dz \end{bmatrix} \quad \text{and}$$

$$\mathbf{A} \oplus \mathbf{B} = \begin{bmatrix} a & | & b \\ a & | & b \\ \hline c & | & d \\ c & | & d \\ c & | & d \end{bmatrix} + \begin{bmatrix} r & s & t & | & & & \\ u & v & w & | & & & \\ x & y & z & | & & & \\ \hline & & & | & r & s & t \\ & & & | & u & v & w \\ & & & | & x & y & z \end{bmatrix} = \begin{bmatrix} a+r & s & t & | & b & & \\ u & a+v & w & | & & b & \\ x & y & a+z & | & & & b \\ \hline c & & & | & d+r & s & t \\ & c & & | & u & d+v & w \\ & & c & | & x & y & d+z \end{bmatrix}.$$

Intuitively, the Kronecker product expresses *contemporaneous* or *synchronized* actions: if  $\mathbf{A}$  and  $\mathbf{B}$  are the transition probability matrices of two independent DTMCs,  $\mathbf{A} \otimes \mathbf{B}$  is the transition probability matrix of their composition, while the Kronecker sum expresses *asynchronous* behavior: if  $\mathbf{A}$  and  $\mathbf{B}$  are the infinitesimal generator matrices of two independent CTMCs,  $\mathbf{A} \oplus \mathbf{B}$  is the infinitesimal generator matrix of their composition.

Note that the definition of Kronecker product (but not that of Kronecker sum) can be extended to rectangular matrices, but we only need it for square matrices in our case.