FACULDADE DE INFORMÁTICA
PUCRS - Brazil

http://www.pucrs.br/inf/pos/

An Alternative Algorithm to Multiply a Vector by a Kronecker Represented Descriptor

P. Fernandes, R. Presotto, A. Sales, T. Webber

TECHNICAL REPORT SERIES

Number 047
April, 2005

Contact:
paulof@inf.pucrs.br
www.inf.pucrs.br/~paulof

rpresotto@inf.pucrs.br
www.inf.pucrs.br/~rpresotto

asales@inf.pucrs.br
www.inf.pucrs.br/~asales

twebber@inf.pucrs.br
www.inf.pucrs.br/~twebber

Published by PPGCC - FACIN - PUCRS
Av. Ipiranga, 6681
90619-900 Porto Algre - RS - Brasil

1 Introduction

All formalisms used to model complex systems are based on a structured description. This is particularly the case of Markovian performance and reliability evaluation models. A myriad of formalisms is available in the research community, *e.g.*, Stochastic Activity Networks [21], Queueing Networks [12, 8], Stochastic Petri Nets (SPN) [1], Performance Evaluation Process Algebra (PEPA) [14], and Stochastic Automata Networks (SAN) [20]. Among such formalisms we are specially interested in those which use Tensor (or Kronecker) Algebra to represent the infinitesimal generator of the underlying Markov chain. Such tensor represented infinitesimal generator is referred in the literature as *descriptor*. SPN [10, 17] and PEPA [15] can convert its models into descriptors for numerical manipulation. However, in the context of this paper, our attention is focused on the SAN formalism for two major reasons: tensor representation is used by the SAN since its first definition [19]; and, at the authors best knowledge, only SAN uses the state of the art form of Kronecker representation, called *Generalized Tensor Algebra* [6].

The key operation to perform iterative solutions, both stationary and transient, of models described in the SAN formalism is the multiplication of a probability vector by the descriptor [22]. Such operation is performed using the *Shuffle* algorithm [11] which has a quite efficient way to handle tensor structures and takes advantage of several techniques developed to SAN [5], to PEPA [13], and to SPN [7].

The main purpose of this paper is to propose an alternative algorithm to perform the vector-descriptor product, which we call *Slice* algorithm. The main advantage of the Slice algorithm is the possible reduction of computational cost (number of multiplications) for very sparse tensor components. Such reduction is achieved keeping the compact tensor format of the descriptor. In some way, the Slice algorithm can be considered as a trade-off between the sparse matrix approach used for straightforward Markov chains, and the fully tensor approach used by the Shuffle algorithm.

Nevertheless, this paper does not exploit the Slice algorithm possibilities to its limits, since very few considerations are made concerning possible optimizations. In particular, we do not analyse the possible benefits of automata reordering according to functional dependencies, which was deeply studied for the Shuffle algorithm. Also a possible hybrid approach using Shuffle and Slice algorithms are not discussed in detail. Actually, we focus our contribution in the presentation of an original way to handle the vector-descriptor product and we present encouraging measures to develop further studies based on this new approach.

This paper is organized with a brief introduction to the SAN formalism (Section 3) followed by sections describing the Shuffle algorithm principle (Section 4.1) and the proposed Slice algorithm (Section 4.2). Section 5 presents two SAN model examples in order to develop some comparative measures in Section 6. Finally the conclusion points out some future works necessary to raise the Slice algorithm to a similar level of optimization as the level already obtained by the Shuffle algorithm.

2 Kronecker algebra

In this section, the concepts of Classical Tensor Algebra [2, 9] and Generalized Tensor Algebra [19, 11] are presented.

2.1 CTA - Classical Tensor Algebra

Define two matrices A and B as follows:

$$A = \begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix} \qquad B = \begin{pmatrix} b_{00} & b_{01} & b_{02} & b_{03} \\ b_{10} & b_{11} & b_{12} & b_{13} \\ b_{20} & b_{21} & b_{22} & b_{23} \end{pmatrix}$$

The *tensor product* $C = A \otimes B$ is given by

$$C = \begin{pmatrix} a_{00}B & a_{01}B \\ a_{10}B & a_{11}B \end{pmatrix}$$

In general, to define the tensor product of two matrices: A of dimensions ($\rho_1 \times \gamma_1$) and B of dimensions ($\rho_2 \times \gamma_2$); it is convenient to observe that the tensor product matrix has dimensions ($\rho_1\rho_2 \times \gamma_1\gamma_2$) and may be considered as consisting of $\rho_1\gamma_1$ blocks each having dimensions ($\rho_2\gamma_2$), *i.e.*, the dimensions of B. To specify a particular element, it suffices to specify the block in which the element occurs and the position within that block of the element under consideration. Thus, as mentioned previously, element c_{36} ($a_{11}b_{02}$) is in the (1, 1) block and at position (0, 2) of that block. The tensor product $C = A \otimes B$ is defined by assigning to the element of C that is in the (k, l) position of block (i, j), the value $a_{ij}b_{kl}$, *i.e.*:

$$C_{[ik][jl]} = a_{ij}b_{kl}.$$

The *tensor sum* of two *square* matrices A and B is defined in terms of tensor products as:

$$A \oplus B = A \otimes I_{n_B} + I_{n_A} \otimes B$$

where n_A is the order of A; n_B is the order of B; I_{n_i} is the identity matrix of order n_i; and "+" represents the usual operation of matrix addition. Since both sides of this operation (matrix addition) must have identical dimensions, it follows that tensor addition is defined for square matrices only. The value assigned to the element $C_{[ik][jl]}$ of the tensor sum $C = A \oplus B$ is defined as:

$$C_{[ik][jl]} = a_{ij}\delta_{kl} + b_{kl}\delta_{ij},$$

where δ_{ij} is the element of i^{th} row and j^{th} column of an identity matrix defined as:

$$\delta_{ij} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases}$$

Some important properties of tensor product and sum operations defined by Davio [2, 9] are:

- Associativity:
  $A \otimes (B \otimes C) = (A \otimes B) \otimes C$ and $A \oplus (B \oplus C) = (A \oplus B) \oplus C$
- Distributivity over (ordinary matrix) addition:
  $(A + B) \otimes (C + D) = (A \otimes C) + (B \otimes C) + (A \otimes D) + (B \otimes D)$
- Compatibility with (ordinary matrix) multiplication:
  $(A \times B) \otimes (C \times D) = (A \otimes C) \times (B \otimes D)$
- Compatibility over multiplication:
  $A \otimes B = (A \otimes I_{n_B}) \times (I_{n_A} \otimes B)$
- Commutativity of normal factors[1]:
  $(A \otimes I_{n_B}) \times (I_{n_A} \otimes B) = (I_{n_A} \otimes B) \times (A \otimes I_{n_B})$

[1] Although this property could be inferred from the *Compatibility with (ordinary matrix) multiplication*, it was defi ned by Fernandes, Plateau and Stewart [11].

2.2 GTA - Generalized Tensor Algebra

Generalized Tensor Algebra is an extension of Classical Tensor Algebra. The main distinction of GTA with respect to CTA is the addition of the concept of *functional elements*. However, a matrix can be composed of constant elements (belonging to $\mathbb{R}$) or functional elements. A functional element is a function evaluated in $\mathbb{R}$ according to a set of parameters composed of the rows of one or more matrices. Generalized tensor product is denoted by $\underset{g}{\otimes}$. The value assigned to the element $C_{[ik][jl]}$ of the generalized tensor product $C = A(\mathcal{B}) \underset{g}{\otimes} B(\mathcal{A})$ is defined as:

$$C_{[ik][jl]} = a_{ij}(b_k)b_{kl}(a_i).$$

Generalized tensor sum is also analogous to the ordinary tensor sum, and is denoted by $\underset{g}{\oplus}$. The elements of the generalized tensor sum $C = A(\mathcal{B}) \underset{g}{\oplus} B(\mathcal{A})$ are defined as:

$$C_{[ik][jl]} = a_{ij}(b_k)\delta_{kl} + b_{kl}(a_i)\delta_{ij}.$$

GTA properties defined by Fernandes, Plateau and Stewart [11] are:

- Associativity:
  $[\ A(\mathcal{B},\mathcal{C}) \underset{g}{\otimes} B(\mathcal{A},\mathcal{C})\ ] \underset{g}{\otimes} C(\mathcal{A},\mathcal{B}) = A(\mathcal{B},\mathcal{C}) \underset{g}{\otimes} [\ B(\mathcal{A},\mathcal{C}) \underset{g}{\otimes} C(\mathcal{A},\mathcal{B})\ ]$
  $[\ A(\mathcal{B},\mathcal{C}) \underset{g}{\oplus} B(\mathcal{A},\mathcal{C})\ ] \underset{g}{\oplus} C(\mathcal{A},\mathcal{B}) = A(\mathcal{B},\mathcal{C}) \underset{g}{\oplus} [\ B(\mathcal{A},\mathcal{C}) \underset{g}{\oplus} C(\mathcal{A},\mathcal{B})\ ]$
- Distributivity over addition:
  $[A(\mathcal{C},\mathcal{D}) + B(\mathcal{C},\mathcal{D})] \underset{g}{\otimes} [C(\mathcal{A},\mathcal{B}) + D(\mathcal{A},\mathcal{B})] = A(\mathcal{C},\mathcal{D}) \underset{g}{\otimes} C(\mathcal{A},\mathcal{B}) \ + \ A(\mathcal{C},\mathcal{D}) \underset{g}{\otimes} D(\mathcal{A},\mathcal{B}) \ + \ B(\mathcal{C},\mathcal{D}) \underset{g}{\otimes} C(\mathcal{A},\mathcal{B}) \ + \ B(\mathcal{C},\mathcal{D}) \underset{g}{\otimes} D(\mathcal{A},\mathcal{B})$
- Compatibility over multiplication I:
  $A \underset{g}{\otimes} B(\mathcal{A}) = I_{n_A} \underset{g}{\otimes} B(\mathcal{A}) \ \times \ A \underset{g}{\otimes} I_{n_B}$
- Compatibility over multiplication II:
  $A(\mathcal{B}) \underset{g}{\otimes} B = A(\mathcal{B}) \underset{g}{\otimes} I_{n_B} \ \times \ I_{n_A} \underset{g}{\otimes} B$
- Decomposability of Generalized Tensor Product into Ordinary Tensor Product:

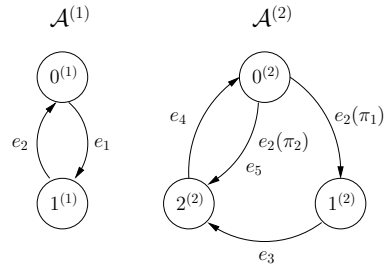
$$A \underset{g}{\otimes} B(\mathcal{A}) = \sum_{k=1}^{n_A} \ell_k(A) \otimes B(a_k)$$

3 Stochastic Automata Networks

The SAN formalism was proposed by Plateau [19] and its basic idea is to represent a whole system by a collection of subsystems with an independent behavior (*local transitions*) and occasional interdependencies (*functional rates* and *synchronizing events*). Each subsystem is described as a stochastic automaton, *i.e.*, an automaton in which the transitions are labeled with probabilistic and timing information. Hence, one can build a continuous-time stochastic process related to SAN, *i.e.*, a SAN model has an equivalent Markov chain model [22, 6].

There are two types of events that change the state of a SAN model: *local events* and *synchronizing events*. Local events change the SAN state changing the state of only one automaton. Synchronizing events, in opposition, can change simultaneously the states of more than one automaton.

The other possibility of interaction among automata is the use of functional rates. Any event occurrence rate may be expressed by a constant value (a positive real number) or a function of the state of other automata. In opposition to synchronizing events, functional rates are one-way interaction among automata, since it affects only the automaton where it appears and not the automata from which it depends.



Type	Event	Rate
loc	e_1	f
syn	e_2	μ
loc	e_3	σ
loc	e_4	δ
loc	e_5	τ

$$f = \left[\left(st\ \mathcal{A}^{(2)} == 0^{(2)}\right) * \lambda\right] + \left[\left(st\ \mathcal{A}^{(2)} == 2^{(2)}\right) * \gamma\right]$$

Figure 1: Example of a SAN model

Figure 1 presents a SAN model with two automata, one synchronizing and four local events. In this example, the rate of the event e_1 is not a constant rate, but a functional rate f described with the SAN notation[2] employed by the PEPS tool [5]. The interpretation of f defines the firing of the transition from $0^{(1)}$ to $1^{(1)}$ state with rate λ if automaton $\mathcal{A}^{(2)}$ is in state $0^{(2)}$, or γ if automaton $\mathcal{A}^{(2)}$ is in state $2^{(2)}$. If automaton $\mathcal{A}^{(2)}$ is in state $1^{(2)}$, the transition from $0^{(1)}$ to $1^{(1)}$ state does not occur (rate equal to 0).

Figure 2 shows the equivalent Markov chain model to the SAN model in Figure 1. It is important to notice that only 5 of the 6 states in this model are reachable. In order to avoid a reducible model, it is necessary to express the reachable global states of a SAN model by means of a (*reachability*) function. For the model in Figure 1, the reachability function must exclude the global state $1^{(1)}1^{(2)}$, thus:

$$Reachability = !\left[\left(st\ \mathcal{A}^{(1)} == 1^{(1)}\right)\ \&\&\ \left(st\ \mathcal{A}^{(2)} == 1^{(2)}\right)\right]$$

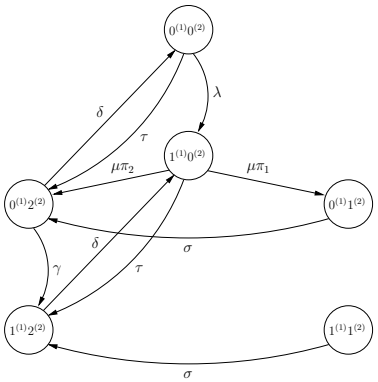


Figure 2: Equivalent Markov chain to the SAN model in Figure 1

The size of a SAN model depends on the number of automata and the size of each automaton. For a model composed by N automata, being each automaton $\mathcal{A}^{(i)}$ composed by n_i local states, the total number of global states is equal to $\prod_{i=1}^{N} n_i$. The straightforward Markov chains approach would

[2]The interpretation of a function can be viewed as the evaluation of an expression of non-typed programming languages, *e.g.*, C language. Each comparison is evaluated to value 1 (*true*) and value 0 (*false*).

represent such model as a sparse matrix of that order. Exploiting the modular structure of a SAN model, and using tensor algebra, the SAN formalism allows a much more efficient management of the memory needs. Thus, the infinitesimal generator of a SAN model is no more described by a unique sparse matrix, but instead, by a *descriptor*. For a model with N automata and E synchronizing events, the descriptor (Q) is an algebraic formula containing $N + 2NE$ matrices:

$$Q = \bigoplus_{i=1}^{N}{}_{g} Q_l^{(i)} + \sum_{j=1}^{E} \left(\bigotimes_{i=1}^{N}{}_{g} Q_{e_j^+}^{(i)} + \bigotimes_{i=1}^{N}{}_{g} Q_{e_j^-}^{(i)} \right) \tag{1}$$

In equation 1, there are N matrices $Q_l^{(i)}$ representing the occurrence of local events of automaton $\mathcal{A}^{(i)}$, NE matrices $Q_{e_j^+}^{(i)}$ representing the occurrence of synchronizing event e in automaton $\mathcal{A}^{(i)}$, and NE analogous matrices representing the diagonal adjustment of synchronizing event e in automaton $\mathcal{A}^{(i)}$ (matrices $Q_{e_j^-}^{(i)}$).

Table 1 details the descriptor Q, which is composed of two separated parts: a tensor sum corresponding to the local events; a sum of tensor products corresponding to the synchronizing events [11]. The tensor sum operation of the local part can be decomposed into the ordinary sum of N normal factors, *i.e.*, a sum of tensor products where all matrices but one are identity matrices[3]. Therefore, in this first part, only the non-identity matrices ($Q_l^{(i)}$) need to be stored.

$\sum$	N		$Q_l^{(1)} \underset{g}{\otimes} I_{n_2} \underset{g}{\otimes} \cdots \underset{g}{\otimes} I_{n_{N-1}} \underset{g}{\otimes} I_{n_N}$ $I_{n_1} \underset{g}{\otimes} Q_l^{(2)} \underset{g}{\otimes} \cdots \underset{g}{\otimes} I_{n_{N-1}} \underset{g}{\otimes} I_{n_N}$ $\vdots$ $I_{n_1} \underset{g}{\otimes} I_{n_2} \underset{g}{\otimes} \cdots \underset{g}{\otimes} Q_l^{(N-1)} \underset{g}{\otimes} I_{n_N}$ $I_{n_1} \underset{g}{\otimes} I_{n_2} \underset{g}{\otimes} \cdots \underset{g}{\otimes} I_{n_{N-1}} \underset{g}{\otimes} Q_l^{(N)}$
	$2E$	e^+	$Q_{e_1^+}^{(1)} \underset{g}{\otimes} Q_{e_1^+}^{(2)} \underset{g}{\otimes} \cdots \underset{g}{\otimes} Q_{e_1^+}^{(N-1)} \underset{g}{\otimes} Q_{e_1^+}^{(N)}$ $\vdots$ $Q_{e_E^+}^{(1)} \underset{g}{\otimes} Q_{e_E^+}^{(2)} \underset{g}{\otimes} \cdots \underset{g}{\otimes} Q_{e_E^+}^{(N-1)} \underset{g}{\otimes} Q_{e_E^+}^{(N)}$
		e^-	$Q_{e_1^-}^{(1)} \underset{g}{\otimes} Q_{e_1^-}^{(2)} \underset{g}{\otimes} \cdots \underset{g}{\otimes} Q_{e_1^-}^{(N-1)} \underset{g}{\otimes} Q_{e_1^-}^{(N)}$ $\vdots$ $Q_{e_E^-}^{(1)} \underset{g}{\otimes} Q_{e_E^-}^{(2)} \underset{g}{\otimes} \cdots \underset{g}{\otimes} Q_{e_E^-}^{(N-1)} \underset{g}{\otimes} Q_{e_E^-}^{(N)}$

Table 1: SAN descriptor

[3] I_{n_i} is an identity matrix of order n_i.

4 Vector-Descriptor Product

The vector-descriptor product operation corresponds to the product of a vector v, as big as the product state space ($\prod_{i=1}^{N} n_i$), by descriptor Q. Since the descriptor is the ordinary sum of $N + 2E$ tensor products, the basic operation of the vector-descriptor product is the multiplication of vector v by a tensor product of N matrices:

$$\sum_{j=1}^{N+2E} \left(v \times \left[\bigotimes_{i=1}^{N}{}_{g} Q_j^{(i)} \right] \right) \tag{2}$$

where $Q_j^{(i)}$ corresponds to I_{n_i}, $Q_l^{(i)}$, $Q_{e^+}^{(i)}$, or $Q_{e^-}^{(i)}$ according to the tensor product term where it appears.

For simplicity in this section, we describe the Shuffle and Slice algorithms for the basic operation vector $\times$ tensor product term omitting the j index from equation 2, *i.e.*:

$$v \times \left[\bigotimes_{i=1}^{N}{}_{g} Q^{(i)} \right] \tag{3}$$

4.1 Shuffle Algorithm

The Shuffle algorithm is described in this section without any considerations about optimizations for the evaluation of functional elements. A thorough study about matrices reordering and generalized tensor algebra properties with this objective can be found in [11]. All those optimizations aim to reduce the overhead of evaluate functional elements, but they do not change the number of multiplications needed by the Shuffle algorithm. Therefore, we ignore the functional elements in the context of this paper, and the basic operation (equation 3) is simplified to consider classical ($\otimes$) and not generalized ($\underset{g}{\otimes}$) tensor products.

The basic principle of the Shuffle algorithm concerns the application of the decomposition of a tensor product in the ordinary product of normal factors property:

$$\begin{aligned} Q^{(1)} \otimes Q^{(2)} \otimes \ldots \otimes Q^{(N-1)} \otimes Q^{(N)} = \quad & (Q^{(1)} \otimes I_{n_2} \otimes \ldots \otimes I_{n_{N-1}} \otimes I_{n_N}) \ \times \\ & (I_{n_1} \otimes Q^{(2)} \otimes \ldots \otimes I_{n_{N-1}} \otimes I_{n_N}) \ \times \\ & \ldots \\ & (I_{n_1} \otimes I_{n_2} \otimes \ldots \otimes Q^{(N-1)} \otimes I_{n_N}) \ \times \\ & (I_{n_1} \otimes I_{n_2} \otimes \ldots \otimes I_{n_{N-1}} \otimes Q^{(N)}) \end{aligned} \tag{4}$$

Rewritten the basic operation (equation 3) according to this property:

$$v \times \left[\prod_{i=1}^{N} I_{nleft_i} \otimes Q^{(i)} \otimes I_{nright_i} \right] \tag{5}$$

where $nleft_i$ corresponds to the product of the order of all matrices before the i^{th} matrix of the tensor product term, *i.e.*, $\prod_{k=1}^{i-1} n_k$ (particular case: $nleft_1 = 1$) and $nright_i$ corresponds to the product of the order of all matrices after the i^{th} matrix of the tensor product term, *i.e.*, $\prod_{k=i+1}^{N} n_k$ (particular case: $nright_N = 1$).

Hence, the Shuffle algorithm consists in multiplying successively a vector by each normal factor. More precisely, vector v is multiplied by the first normal factor, then the resulting vector is multiplied by the next normal factor and so on until the last factor. In fact, the multiplication of a vector v by the i^{th} normal factor corresponds to ***shuffle*** the elements of v in order to assemble $nleft_i \times nright_i$ vectors

of size n_i and multiply them by matrix $Q^{(i)}$. Therefore, assuming that matrix $Q^{(i)}$ is stored as a sparse matrix, the number of multiplications needed to multiply a vector by the i^{th} normal factor is:

$$nleft_i \times nright_i \times nz_i \tag{6}$$

where nz_i corresponds to the number of nonzero elements of the i^{th} matrix of the tensor product term ($Q^{(i)}$). Considering the number of multiplications to all normal factors of a tensor product term, we obtain [11]:

$$\prod_{i=1}^{N} n_i \times \sum_{i=1}^{N} \frac{nz_i}{n_i} \tag{7}$$

4.2 Slice Algorithm

Slice is an alternative algorithm to perform the vector-descriptor product not based only on the decomposition of a tensor product in the ordinary product of normal factors property (equation 4), but also applies a very basic property, the *Additive Decomposition* [11]. This property simply states that a tensor product term can be described by a sum of unitary matrices[4]:

$$Q^{(1)} \otimes Q^{(2)} \otimes \ldots \otimes Q^{(N-1)} \otimes Q^{(N)} = \sum_{i_1=1}^{n_1} \ldots \sum_{i_N=1}^{n_N} \sum_{j_1=1}^{n_1} \ldots \sum_{j_N=1}^{n_N} \left(\hat{q}^{(1)}_{(i_1,j_1)} \otimes \ldots \otimes \hat{q}^{(N)}_{(i_N,j_N)} \right) \tag{8}$$

where $\hat{q}^{(k)}_{(i,j)}$ is an unitary matrix of order n_k in which the element in row i and column j is equal to element i, j of the matrix $Q^{(k)}$.

Obviously, the application of such property over a tensor product with fully dense matrices results in a catastrophic number of $\prod_{i=1}^{N}(n_i)^2$ unitary matrix terms, but the number of terms is considerably reduced for sparse matrices. In fact, there is one unitary matrix to each possible combination of one nonzero element from each matrix. We may define $\theta(1 \ldots N)$ as the set of all possible combinations of nonzero elements of the matrices from $Q^{(1)}$ to $Q^{(N)}$. Therefore, the cardinality of $\theta(1 \ldots N)$, and consequently the number of unitary matrices to decompose a tensor product term, is given by: $\prod_{i=1}^{N} nz_i$.

Generically evaluating the unitary matrices from equation 8, the sole nonzero element appears in the tensor coordinates i_1, j_1 for the outermost block, coordinates i_2, j_2 for the next inner block, and so on until the coordinates i_N, j_N for the innermost block. By the own definition of the tensor product, the value of an element is $\prod_{k=1}^{N} q^{(k)}_{(i_k,j_k)}$. For such unitary matrices, we use the following notation:

$$\hat{Q}^{(1\ldots N)}_{i_1,\ldots,i_N,j_1,\ldots,j_N} = \hat{q}^{(1)}_{(i_1,j_1)} \otimes \ldots \otimes \hat{q}^{(N)}_{(i_N,j_N)} \tag{9}$$

The pure application of the Additive Decomposition property corresponds to generate a single equivalent sparse matrix to the tensor product term. For many cases, it may result in a too large number of elements. It is precisely to cope with this problem that the Shuffle algorithm was proposed. However, the manipulation of considerably sparse tensor product terms like this is somewhat awkward, since a decomposition in N normal factors may be a too large effort to multiply very few resulting elements.

The basic principle of the Slice algorithm is to handle the tensor product term in two distinct parts. The Additive Decomposition property is applied to all first $N-1$ matrices, generating $\prod_{i=1}^{N-1} nz_i$ very sparse terms which are multiplied (tensor product) by the last matrix, *i.e.*:

$$Q^{(1)} \otimes Q^{(2)} \otimes \ldots \otimes Q^{(N-1)} \otimes Q^{(N)} = \sum_{\substack{\forall i_1,\ldots,i_{N-1},j_1,\ldots,j_{N-1} \\ \in \theta(1\ldots N-1)}} \hat{Q}^{(1\ldots N-1)}_{i_1,\ldots,i_{N-1},j_1,\ldots,j_{N-1}} \otimes Q^{(N)} \tag{10}$$

[4] A unitary matrix is a matrix in which there is only one nonzero element.

Therefore, the Slice algorithm consists in dealing with $N-1$ matrices as a very sparse structure, and dealing with the last matrix as the Shuffle approach did. The multiplication of a vector v by the tensor product term (equation 3) using the Slice algorithm can be rewritten as:

$$v \times \left[\sum_{\substack{\forall i_1,\ldots,i_{N-1},j_1,\ldots,j_{N-1} \\ \in \theta(1\ldots N-1)}} \hat{Q}^{(1\ldots N-1)}_{i_1,\ldots,i_{N-1},j_1,\ldots,j_{N-1}} \otimes Q^{(N)} \right] \tag{11}$$

We call each term of the previous equation as *Additive Unitary Normal Factor*, since it is composed of an unitary matrix times a standard normal factor. The decomposition in normal factors applied to each additive unitary normal factor of equation 11 results in:

$$v \times \left(\hat{Q}^{(1\ldots N-1)}_{i_1,\ldots,i_{N-1},j_1,\ldots,j_{N-1}} \otimes I_{n_N} \right) \times \left(I_{nleft_N} \otimes Q^{(N)} \right) \tag{12}$$

It is important to notice that the first multiplication takes only n_N elements of the vector v and it corresponds to the product of this *sliced* vector (called v_s) by the single scalar which is the nonzero element of the matrix $\hat{Q}^{(1\ldots N-1)}_{i_1,\ldots,i_{N-1},j_1,\ldots,j_{N-1}}$. The resulting vector, called v'_s, must then be multiplied only once by the matrix $Q^{(N)}$, since all other positions of the intermediate vector (except those in v'_s) are zero.

The application of the Slice algorithm must generate the nonzero element (c) of matrix $\hat{Q}^{(1\ldots N-1)}_{i_1,\ldots,i_{N-1},j_1,\ldots,j_{N-1}}$. Hence, it must pick a *slice* of the vector v (called v_s) according to the row position of element c, and multiply all elements of v_s by c. In fact, this multiplication by a scalar corresponds to the first multiplication by a normal factor of equation 12. The resulting vector, called v'_s, must be multiplied by the matrix $Q^{(N)}$ (second multiplication in equation 12), accumulating the result (r_s) in the positions of the resulting vector r corresponding to the column position of element c.

The Slice algorithm (Algorithm 1) can be summarize for all Additive Unitary Normal Factors in the operation:

$$r = v \times \left[\sum_{\substack{\forall i_1,\ldots,i_{N-1},j_1,\ldots,j_{N-1} \\ \in \theta(1\ldots N-1)}} \hat{Q}^{(1\ldots N-1)}_{i_1,\ldots,i_{N-1},j_1,\ldots,j_{N-1}} \otimes Q^{(N)} \right]$$

Algorithm 1 Slice Algorithm

1: **for all** $i_1,\ldots,i_{N-1},j_1,\ldots,j_{N-1} \in \theta(1\ldots N-1)$ **do**
2: $\quad c \leftarrow \prod_{k=1}^{N-1} q^{(k)}_{(i_k,j_k)}$
3: $\quad$ slice v_s from v according to $i_1,\ldots,i_{N-1}$
4: $\quad v'_s \leftarrow c \times v_s$
5: $\quad r_s \leftarrow v'_s \times Q^{(N)}$
6: $\quad$ add r_s to r according to $j_1,\ldots,j_{N-1}$
7: **end for**

The computational cost (number of needed multiplications) of the slice algorithm considers the number of unitary matrices ($\prod_{i=1}^{N-1} nz_i$), the cost to generate the nonzero element of each unitary matrix ($N-2$), the cost to multiply it by each element of sliced vector v_s (n_N), and the cost to multiply v_s by the last matrix $Q^{(N)}$ (nz_N), *i.e.*:

$$\prod_{i=1}^{N-1} nz_i \times \left[(N-2) + n_N + nz_N \right] \tag{13}$$

4.3 Diagonal Optimization

An optimization available to the Shuffle algorithm, is called *Diagonal Optimization*. Among all known optimizations in the Shuffle algorithm, this one is presented because it saves considerable time in a sequential implementation by changing the descriptor structure.

The basic principle of this optimization consists in remove all elements of the descriptor matrices that may generate a diagonal element in the corresponding unique (and usually huge) sparse matrix. Therefore, all diagonal elements of the local matrices ($Q_l^{(i)}$) are removed, and all terms concerning diagonal adjustment of the synchronizing events ($Q_{e_j^-}^{(i)}$) are also removed. All diagonal elements of the corresponding unique sparse matrix are previously computed and stored in a state space sized vector D. The resulting descriptor is:

$$Q = \bigoplus_{i=1}^{N}{}_g \bar{Q}_l^{(i)} + \sum_{j=1}^{E} \left(\bigotimes_{i=1}^{N}{}_g Q_{e_j^+}^{(i)} \right) + D \tag{14}$$

where the local matrices $\bar{Q}_l^{(i)}$ corresponds to the original local matrices which all diagonal elements were removed.

This optimization can be applied to both Shuffle and Slice algorithms reducing significatively the number of multiplications to perform, but increasing the memory needs to store the descriptor due to the inclusion of vector D. The computational costs for the algorithms is still calculated according to equations 7 and 13, but now it does not takes into account the product tensor terms for diagonal adjustments, and it considers less nonzero elements in the local matrices. However, it must add as many multiplications as the number of elements in vector D (the state space size). The impact of this optimization can be observed in the numerical analysis section, when the speed up and memory increase of this optimization is analyzed.

5 Examples

In this section, we present two examples described by the SAN formalism in order to develop some comparative measures.

5.1 Mixed Queue Network Model

This example presents a two-classes mixed *Finite Capacity Queueing Networks* (FCQN). For this model (Figure 3), customers of the first class will act as an open system (queue 1 up to queue 5), and the customers of the second class will act as a closed system (queue 1 up to queue 3).

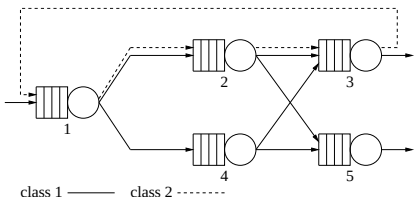
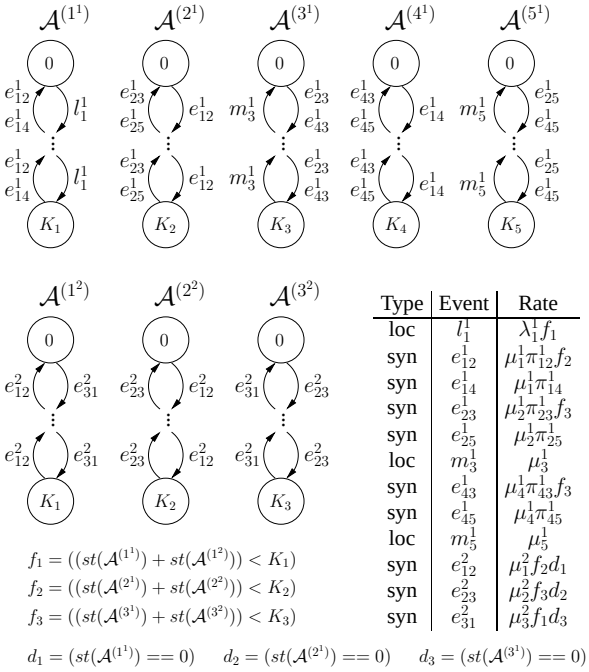


Figure 3: Mixed Queueing Network model

In this model, all queues have only one server available and the customers of class 1 have priority over the customers of class 2.



Type	Event	Rate
loc	l_1^1	$\lambda_1^1 f_1$
syn	e_{12}^1	$\mu_1^1 \pi_{12}^1 f_2$
syn	e_{14}^1	$\mu_1^1 \pi_{14}^1$
syn	e_{23}^1	$\mu_2^1 \pi_{23}^1 f_3$
syn	e_{25}^1	$\mu_2^1 \pi_{25}^1$
loc	m_3^1	μ_3^1
syn	e_{43}^1	$\mu_4^1 \pi_{43}^1 f_3$
syn	e_{45}^1	$\mu_4^1 \pi_{45}^1$
loc	m_5^1	μ_5^1
syn	e_{12}^2	$\mu_1^2 f_2 d_1$
syn	e_{23}^2	$\mu_2^2 f_3 d_2$
syn	e_{31}^2	$\mu_3^2 f_1 d_3$

Figure 4: SAN model equivalent to Figure 3

Figure 4 shows a SAN model equivalent to the model presented in Figure 3. For this SAN model, each queue Q_i^r (queue i of class r) is represented by an automaton $\mathcal{A}^{(i^r)}$, *i.e.*, there are five automata representing the queues of class 1 and three automata representing the queues of class 2.

A local event is used to represent each communication with the exterior - arrival at queue Q_1^1 (event l_1^1) and departure of customers of class 1 from queues Q_3^1 and Q_5^1 (events m_3^1 and m_5^1 respectively). Synchronizing events e_{ij}^r are used to represent the exchange of customers of class r from queue i to queue j. The arrival rate of customers of class r at queue Q_i^r from outside the model is denoted by λ_i^r, as well as μ_i^r indicates the service rate of customers of class r at queue Q_i^r. We denote by π_{ij}^r the probability that a customer of class r will be routed from queue Q_i^r to Q_j^r.

The capacity of queue Q_i^r is indicate by K_i. Function f_i is used to represent the capacity restriction of admission in queue i accepting both classes of customers, and function d_i indicates the priority of customers of a class over other one at queue i.

5.2 Propagation Implementation Model

The Propagation algorithm is based on a classic region growing method for image segmentation [18] which uses pixel homogeneity. However, instead of using pixel homogeneity property, a similar measure based on the matches correlation score is adopted [16]. This propagation strategy could also be justified because the seed pairs are composed by points of interest, which are the local maxima of the textureness. Therefore, these matches neighbors are also strongly textured, which allows good propagation even though they are not local maxima. Such algorithm advances by comparing neighbors pixels through out the source images. The freedom of evolution through out the images assumes that the algorithm knows the entire surface of the images. This can create a situation in which several processors are propagating over the same regions at the same time creating a redundancy of computation.

A solution proposed to the Propagation algorithm is based on a master/slave paradigm [3]. One processor (master) is responsible for distributing the work and centralizing the final results, and the others (slaves) are running the propagation algorithm each one using a subset of the seed pairs and knowing a pair of corresponding slices over the images (coordinates of target slice).

Figure 5 presents a SAN model which contains one automaton *Master*, one automaton *Buffer*, and S automata $Slave^{(i)}$ $(i = 1..S)$.

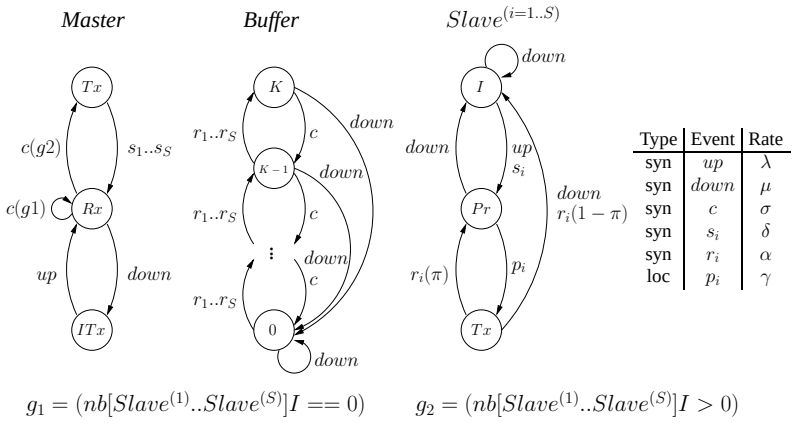


Figure 5: SAN model for the Propagation Implementation

Automaton *Master* is responsible for the distribution of slices (*tasks*) to the S slaves (automaton $Slave^{(i)}$, where $i = 1..S$) and analysis of final matches (*results*) evaluated by them. It consumes such final matches from a repository. Since this SAN model has asynchronous characteristics, the results generated by the slave nodes are stored on this repository - automaton *Buffer*. Thus, master node does not need to synchronize the reception of results sent by slaves.

6 Numerical Analysis

The numerical results for this section were obtained on a 2.8 GHz Pentium IV Xeon under Linux operating system with 2 GBytes of memory. The actual PEPS 2003 implementation [5] was used to obtain the Shuffle algorithm results and a prototype implementation was used to obtain the Slice algorithm results.

6.1 Shuffle and Slice Comparison

Mixed QN												
	Shuffle			*Slice*			*Shuffle-D*			*Slice-D*		
	c.c.	*time*	*mem.*	*c.c.*	*time*	*mem.*	*c.c.*	*time*	*mem.*	*c.c.*	*time*	*mem.*
$K=3$	9.14×10^6	0.16	4	2.59×10^6	0.05	4	4.64×10^6	0.08	529	1.36×10^6	0.03	529
$K=4$	5.53×10^7	0.87	5	1.58×10^7	0.28	5	2.80×10^7	0.44	3,069	8.30×10^6	0.14	3,069
$K=5$	2.40×10^8	3.77	6	6.86×10^7	1.19	6	1.22×10^8	1.91	13,140	3.60×10^7	0.63	13,140
$K=6$	8.30×10^8	12.86	6	2.36×10^8	4.06	6	4.21×10^8	6.52	45,056	1.24×10^8	2.13	45,056
$K=7$	2.43×10^9	47.31	7	6.85×10^8	14.82	7	1.23×10^9	23.98	131,091	3.59×10^8	7.77	131,091
Propagation												
	Shuffle			*Slice*			*Shuffle-D*			*Slice-D*		
	c.c.	*time*	*mem.*	*c.c.*	*time*	*mem.*	*c.c.*	*time*	*mem.*	*c.c.*	*time*	*mem.*
$S=3$	2.43×10^5	<0.01	9	6.59×10^4	<0.01	9	1.27×10^5	<0.01	37	4.17×10^4	<0.01	37
$S=4$	1.10×10^6	0.02	12	2.61×10^5	<0.01	12	5.70×10^5	0.01	95	1.63×10^5	<0.01	95
$S=5$	4.67×10^6	0.09	15	9.97×10^5	0.02	15	2.40×10^6	0.04	257	6.22×10^5	0.01	257
$S=6$	1.88×10^7	0.33	18	3.71×10^6	0.07	18	9.61×10^6	0.17	733	2.31×10^6	0.04	733
$S=7$	7.31×10^7	1.14	21	1.35×10^7	0.23	21	3.72×10^7	0.58	2,144	8.39×10^6	0.14	2,144

Table 2: Shuffle and Slice algorithms comparison

The first set of experiments is conducted for both examples using both algorithms in their standard versions (columns *Shuffle* and *Slice*) and versions with the diagonal optimization (columns *Shuffle-D* and

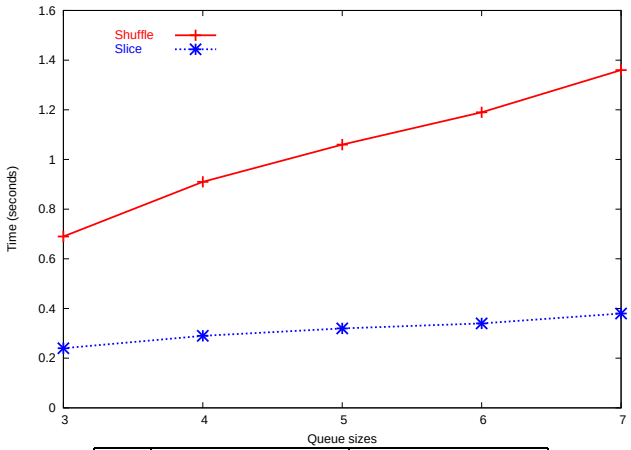
Slice-D). For each option we compute the number of multiplications performed (computational cost - *c.c.*), and the time to execute a complete multiplication in seconds (*time*). For the Mixed Queue Network model (*Mixed QN*) described in Section 5.1, we consider all queues with the same capacity (K) assuming the values $K = 3..7$. For the Propagation Implementation model (*Propagation*) described in Section 5.2, we fixed the buffer size in 40 places and we assign the number of slaves (S) with the values $S = 3..7$. For all models, we also measured the memory needs to store the descriptor in KBytes, which is indicated in column *mem*. Obviously, the memory needs for the Shuffle and Slice approach are equal, since the choice of algorithm does not interfere with the descriptor structure. The memory needs are indicated to show the cost of using the diagonal optimization which is considerably faster, but more memory demanding.

The number of multiplications needed by the Slice algorithm (equation 13) is less significant than the number needed in the Shuffle algorithm (equation 7). Such advantage remains valid, even considering the diagonal optimization in both algorithms, as illustrated in the columns *c.c.* on the right hand side of Table 2. Even though the time spent in Slice algorithm is still better than Shuffle one, the gains are slightly less significant. This happens probably due to a more optimized treatment of the function evaluations in Shuffle algorithm. An analysis of the functional evaluations for the Slice algorithm may reveal further optimizations, but as said in the introduction such analysis is out of the scope of this paper.

It is also important to notice that the standard Slice algorithm execution is faster than the Shuffle algorithm using the diagonal optimization. For example, in the Propagation model with $S = 7$, Slice uses less than 1% of the memory and it is more than 2 times faster than the best Shuffle solution currently available.

6.2 Slice Algorithm Optimizations

The second set of experiments is conducted over the Mixed Queue Network example assuming all queues, but the last one, with the same capacity ($K = 4$). The capacity of the last queue (K_5) is tested with values 3, 4, 5, 6, and 7. For these experiments only results of the standard versions of the Shuffle and Slice algorithm were computed. Figure 6 shows a table with the numeric results obtained for these experiments and a plot of the time spent in both approaches.

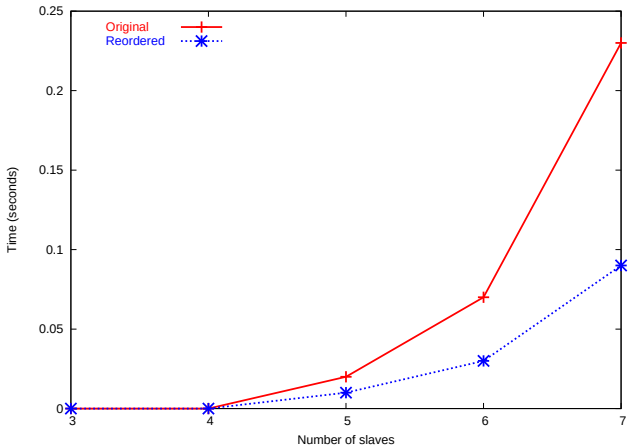


	Shuffle		*Slice*	
K_5	*c.c.*	*time*	*c.c.*	*time*
3	4.42×10^7	0.69	1.38×10^7	0.24
4	5.53×10^7	0.91	1.58×10^7	0.29
5	6.65×10^7	1.06	1.79×10^7	0.32
6	7.76×10^7	1.19	1.99×10^7	0.34
7	8.88×10^7	1.36	2.20×10^7	0.38

Figure 6: Increasing the size of the last queue

Observing equations 7 and 13, it is possible to notice that, unlike the cost of the Shuffle algorithm, the cost of the Slice algorithm is less dependent on the order of the last matrix. This can be verified by the results in Figure 6 since the Slice and Shuffle curves have clearly different behaviors.

The last set of experiments (Figure 7) shows the effect of automata reordering for the Propagation Implementation model. For these experiments, only the results of the Slice algorithm are indicated. The left hand side columns (*Original*) indicate the results obtained for the example with their automata in the order that they were presented in Figure 5, *i.e.*, with the larger automaton (*Buffer*) as the second one. The right hand side columns (*Reordered*) indicate the results obtained putting automaton *Buffer* as the last one. The results show clearly the improvements in the number of multiplications as well as in the time spent. Such encouraging result suggests that many other optimizations could still be found to the Slice algorithm.



	Original		*Reordered*	
S	*c.c.*	*time*	*c.c.*	*time*
3	6.59×10^4	< 0.01	3.67×10^4	< 0.01
4	2.61×10^5	< 0.01	1.37×10^5	< 0.01
5	9.97×10^5	0.02	4.92×10^5	0.01
6	3.71×10^6	0.07	1.73×10^6	0.03
7	1.35×10^7	0.23	5.95×10^6	0.09

Figure 7: Automata reordering

7 Conclusion and Future Works

This paper proposes a different way to perform vector-descriptor product. The new Slice algorithm has shown a better overall performance than the traditional Shuffle algorithm for all examples tested. In fact, the Shuffle algorithm would only be more efficient for quite particular cases in which the descriptor matrices would be nearly full. Even though we could imagine such tensor products (with only nearly full matrices), we were not able to generate a real SAN model with such characteristics. It seems that real case SAN models have naturally sparse matrices. The local part of a descriptor is naturally very sparse due to the tensor sum structure. The synchronizing events are mostly used to describe exceptional behaviors, therefore it lets the synchronizing part of the descriptor also quite sparse.

As a matter of fact, the Slice algorithm offers a good trade-off between the unique sparse matrix approach used for straightforward Markov chains and the pure tensor approach of the Shuffle algorithm. It is much more memory efficient than the unique sparse matrix approach, and it would only be slower than the Shuffle algorithm in hypothetical models with nearly full matrices. However, even for those hypothetical models, the Slice approach may be used for some terms of the descriptor. Such hybrid

approach could analyze which algorithm should be used to each one of the tensor product terms of the descriptor.

Besides the immediate future works to develop further experiments with the Slice algorithm already mentioned in the previous section (*e.g.*, function evaluations), we may also foresee studies concerning parallel implementations. The prototyped parallel implementation of the Shuffle algorithm [4] has already shown consistent gains to solve particularly slow SAN models. Nevertheless, the Shuffle algorithm parallelization suffers an important limitation that consists in the passing of a whole tensor product term to each parallel node. This is a problem since all nodes must compute multiplications of the whole vector v by a tensor product term that usually has nonzero elements in many positions.

The Slice algorithm can offer a more effective parallelization since its Additive Unitary Normal Factors only affect few positions of vector v. A parallel node could receive only similar terms and, therefore, not handle the whole vector v. This can be specially interesting for parallel machines with nodes with few memory resources. Unfortunately, in order to profit from this Slice algorithm characteristic we must give up the diagonal optimization, since it creates a (usually huge) state space sized vector that could hardly be efficiently handle by a parallel implementation.

Concentrating back in the sequential implementation, our first results with the Slice algorithm prototype were very encouraging, but we expect to have many improvements to do before integrate this new algorithm in a new version of the PEPS software tool. As we said before, this paper is just a first step for this new approach and much numerical studies have to be done. However, the current version of the Slice algorithm already shows better results than Shuffle. So even at the worst case scenario, *i.e.*, if no other optimizations could be found, Slice is a more efficient solution to be used in Kronecker structured formalism solvers as PEPS, PEPA Workbench or SMART.

References

[1] M. Ajmone-Marsan, G. Conte, and G. Balbo. A Class of Generalized Stochastic Petri Nets for the Performance Evaluation of Multiprocessor Systems. *ACM Transactions on Computer Systems*, 2(2):93–122, 1984.

[2] V. Amoia, G. De Micheli, and M. Santomauro. Computer-Oriented Formulation of Transition-Rate Matrices via Kronecker Algebra. *IEEE Transactions on Reliability*, R-30(2):123–132, 1981.

[3] L. Baldo, L. Brenner, L. G. Fernandes, P. Fernandes, and A. Sales. Performance Models for Master/Slave Parallel Programs. In J. Bradley and W. Knottenbelt, editors, *First International Workshop on Practical Applications of Stochastic Modelling*, pages 41–60, The Royal Society, London, UK, September 2004.

[4] L. Baldo, L. G. Fernandes, P. Roisenberg, P. Velho, and T. Webber. Parallel PEPS Tool Performance Analysis using Stochastic Automata Networks. In M. Donelutto, D. Laforenza, and M. Vanneschi, editors, *Euro-Par 2004 International Conference on Parallel Processing*, volume 3149 of *Lecture Notes in Computer Science*, pages 214–219, Pisa, Italy, August/September 2004. Springer-Verlag Heidelberg.

[5] A. Benoit, L. Brenner, P. Fernandes, B. Plateau, and W. J. Stewart. The PEPS Software Tool. In *Computer Performance Evaluation / TOOLS 2003*, volume 2794 of *LNCS*, pages 98–115, Urbana, IL, USA, 2003. Springer-Verlag Heidelberg.

[6] L. Brenner, P. Fernandes, and A. Sales. The Need for and the Advantages of Generalized Tensor Algebra for Kronecker Structured Representations. *International Journal of Simulation: Systems, Science & Technology*, 6(3-4):52–60, February 2005.

[7] G. Ciardo, R. L. Jones, A. S. Miner, and R. Siminiceanu. SMART: Stochastic Model Analyzer for Reliability and Timing. In *Tools of Aachen 2001 International Multiconference on Measurement, Modelling and Evaluation of Computer-Communication Systems*, pages 29–34, Aachen, Germany, September 2001.

[8] Y. Dallery, Z. Liu, and D. Towsley. Properties of fork/join queueing networks with blocking under various operating mechanisms. *IEEE Transactions on Robotics and Automation*, 13(4):503–518, 1997.

[9] M. Davio. Kronecker Products and Shuffle Algebra. *IEEE Transactions on Computers*, C-30(2):116–125, 1981.

[10] S. Donatelli. Superposed generalized stochastic Petri nets: definition and efficient solution. In R. Valette, editor, *Proceedings of the* 15^{th} *International Conference on Applications and Theory of Petri Nets*, pages 258–277. Springer-Verlag Heidelberg, 1994.

[11] 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.

[12] E. Gelenbe. G-Networks: Multiple Classes of Positive Customers, Signals, and Product Form Results. In *Performance*, volume 2459 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag Heidelberg, 2002.

[13] S. Gilmore and J. Hillston. The PEPA Workbench: A Tool to Support a Process Algebra-based Approach to Performance Modelling. In *Computer Performance Evaluation*, pages 353–368, 1994.

[14] S. Gilmore, J. Hillston, L. Kloul, and M. Ribaudo. PEPA nets: a structured performance modelling formalism. *Performance Evaluation*, 54(2):79–104, 2003.

[15] J. Hillston and L. Kloul. An Efficient Kronecker Representation for PEPA models. In L. de Alfaro and S. Gilmore, editors, *Proceedings of the first joint PAPM-PROBMIV Workshop)*, pages 120–135, Aachen, Germany, September 2001. Springer-Verlag Heidelberg.

[16] M. Lhuillier. *Modlisation pour la synthse d'images partir d'images*. Thse de doctorat en informatique, Institut National Polytechnique de Grenoble, France, 2000.

[17] A. S. Miner and G. Ciardo. A data structure for the efficient Kronecker solution of GSPNs. In *Proceedings of the* 8^{th} *International Workshop on Petri Nets and Performance Models*, pages 22–31, Zaragoza, Spain, September 1999.

[18] O. Monga. An Optimal Region Growing Algorithm for Image Segmentation. *International Journal of Pattern Recognition and Artificial Intelligence*, 1(3):351–375, 1987.

[19] B. Plateau. On the stochastic structure of parallelism and synchronization models for distributed algorithms. In *Proceedings of the 1985 ACM SIGMETRICS conference on Measurements and Modeling of Computer Systems*, pages 147–154, Austin, Texas, USA, 1985. ACM Press.

[20] B. Plateau and K. Atif. Stochastic Automata Networks for modelling parallel systems. *IEEE Transactions on Software Engineering*, 17(10):1093–1108, 1991.

[21] W. H. Sanders and J. F. Meyer. Stochastic Activity Networks: Formal Definitions and Concepts. In *Lectures on Formal Methods and Performance Analysis : First EEF/Euro Summer School on Trends in Computer Science*, volume 2090 of *Lecture Notes in Computer Science*, pages 315–343, Berg En Dal, The Netherlands, July 2001. Springer-Verlag Heidelberg.

[22] W. J. Stewart. *Introduction to the numerical solution of Markov chains*. Princeton University Press, 1994.