FACULDADE DE INFORMÁTICA
PUCRS - Brazil

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

The Hybrid Algorithm to Vector-Descriptor Product

Ricardo M. Czekster, Paulo Fernandes, Thais Webber

TECHNICAL REPORT SERIES

Number 052
Junho, 2006

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

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

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

Modeling large complex systems always requires a structured description. The state space explosion itself exists only because we can describe a very large system, *i.e.*, a system with a huge state space, by describing a structure simple enough to be understood by us, humans. Therefore, it is admissible to rely on a structured description to all problems sufficiently large to be called that way. This is evident observing a stochastic Petri net (SPN) [1] or a stochastic automata network (SAN) [17], and even when you look deeply how large straightforward Markov chains are defined, *e.g.*, the balls and buckets concept in MARCA [18]. Using a less procedural approach, process algebras [12] or graph grammars [9] also use a structured, and very modular, description.

The basic principle that recommends the use of Kronecker (tensor) representation for infinitesimal generators or probability matrices is to take advantage of all the structural information already used in the original description of the Markovian models. Such principle appears since the first definitions of stochastic automata networks, but recently it has also been used in other stochastic formalisms [8, 12]. However, the use of Kronecker representations for the infinitesimal generator is not always helpful. It undeniably reduces the memory requirements [14, 5], but it often increases the CPU time needed to achieve stationary or transient solution. One of the major problems with structured representations is the insertion of unreachable states, but to cope with that, very efficient approaches deal with the determination of reachable set [6, 15]. It remains an open problem the efficient solution of large and complex models where all, or almost all, states are reachable.

The numerical algorithms known to compute exact solution of large reachable state space, non-product form, Kronecker represented models are usually iterative and they are based on the vector-descriptor[1] multiplication. This operation can be done by using clever data structures, *e.g.*, matrix diagrams [13], or using a sum of tensor product of standard matrix structures [10]. Despite the algorithmic differences, both approaches can be summarized in finding an efficiently way to multiply a (usually huge) vector by a non-trivial structure. Old stochastic Petri net solutions [1] translate the model representation into a singular sparse matrix. Obviously, this sparse approach cannot be employed to deal with really large models (*e.g.*more than 500 thousand states), since it usually requires the storage of a too large sparse matrix (*e.g.*more than 4 million nonzero elements). The usual stochastic automata network solution, called *Shuffle algorithm*, deals with permutations of matrices and tensors [10]. The Shuffle algorithm can be applied to virtually any structured model, but it suffers with a high CPU cost to compute solutions.

The purpose of this paper is to propose a combined solution to achieve the gains of the Sparse approach for the descriptor parts that are not too big to store, and to keep the absolute necessary memory savings of the Shuffle algorithm for the descriptor parts that needs it. The proposed algorithm is called *Hybrid* and it basically splits every tensor product term of the descriptor in two sets of matrices: the set of matrices to be handled almost as in the Sparse approach; and the matrices to be handled using the Shuffle algorithm. Even though our solution could be employed to descriptors generated by any structured formalism, in the context of this paper we will limit ourselves to the analysis of descriptors generated by SAN models.

The next section of this paper introduces the basics of tensor algebra and the usual structure of a descriptor generated by structured formalisms. Section 3 presents the Sparse, Shuffle and Slice algorithms of vector-descriptor multiplication. Section 4 proposes the Hybrid algorithm and Section 5 proposes two intuitive optimizations, and analyzes numerical issues according to a prototype experiment. Finally, the conclusion presents some future works to study possible improvements of the Hybrid algorithm.

[1] A Kronecker represented infinitesimal generator is usually called a *descriptor* [16, 8].

2 Kronecker Algebra

This section presents a description of some basic concepts needed to understand this paper context. It starts with a brief introduction to the concepts of Classical Tensor Algebra[2] (CTA) [2, 7, 10] and the tensor structure, called Markovian descriptor, used to represent the infinitesimal generator of a (structurally defined) model is described. In the context of this paper we describe specifically a descriptor generated from a SAN model. However, any other structured formalisms with a tensor representation, *e.g.*, SPN or PEPA, could be employed without any loss of generality.

2.1 Classical Tensor Algebra

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 the resulting tensor product $C = (C = A \otimes B)$ as a matrix with dimensions $(\rho_1\rho_2 \times \gamma_1\gamma_2)$. However, the tensor C is a four dimension tensor, which can be flattened (*i.e.* put in a two-dimension format) in a single matrix C consisting of $\rho_1\gamma_1$ blocks each having dimensions $(\rho_2\gamma_2)$. 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, the matrix C element c_{36} (which corresponds to tensor C element $C_{[1,0][1,2]}$ is in the (1, 1) block and at position (0, 2) of that block and has the numeric value $(a_{11}\ b_{02})$. Algebraically, the tensor C elements are defined by:

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

Defining two matrices A and B as follows:

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

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

$$C = \left(\begin{array}{c|c|c|c} a_{00}B & a_{01}B & a_{02}B & a_{03}B \\ \hline a_{10}B & a_{11}B & a_{12}B & a_{13}B \\ \hline a_{20}B & a_{21}B & a_{22}B & a_{23}B \end{array}\right)$$

In this paper context, a particularly important type of tensor product is the tensor product where one of the matrices is an identity matrix of order n (I_n). These particular tensor products are called *normal factors* and they can be composed by matrices only on diagonal blocks:

$$I_3 \otimes B \quad = \quad \left(\begin{array}{cc|cc|cc} b_{00} & b_{01} & 0 & 0 & 0 & 0 \\ b_{10} & b_{11} & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & b_{00} & b_{01} & 0 & 0 \\ 0 & 0 & b_{10} & b_{11} & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & b_{00} & b_{01} \\ 0 & 0 & 0 & 0 & b_{10} & b_{11} \end{array}\right)$$

[2] An extension of CTA, called Generalized Tensor Algebra [10] is used to represent structured models. However in the context of this paper we will focus our attention in the CTA.

or diagonal matrices in every block:

$$B \otimes I_3 \quad = \quad \left(\begin{array}{ccc|ccc} b_{00} & 0 & 0 & b_{01} & 0 & 0 \\ 0 & b_{00} & 0 & 0 & b_{01} & 0 \\ 0 & 0 & b_{00} & 0 & 0 & b_{01} \\ \hline b_{10} & 0 & 0 & b_{11} & 0 & 0 \\ 0 & b_{10} & 0 & 0 & b_{11} & 0 \\ 0 & 0 & b_{10} & 0 & 0 & b_{11} \end{array}\right)$$

The *tensor sum* of two square matrices A and B is defined in terms of tensor products as the sum of normal factors of matrices A and B, *i.e.*:

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

where n_A and n_B are respectively the orders of the matrices A and B. Since both sides of the usual matrix addition operation must have identical dimensions, it follows that tensor sum is defined for square matrices only. The algebraic definition of the tensor sum $C = A \oplus B$ are defined as:

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

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

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

Some important properties of the Classical Tensor Product and sum operations defined by Davio [2, 7] 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[3]:
  $(A \otimes I_{n_B}) \times (I_{n_A} \otimes B) = (I_{n_A} \otimes B) \times (A \otimes I_{n_B})$

Due to the Associativity property, the normal factor definition may be generalized to a tensor product of a suite of matrices, where all matrices but one are identities. This very useful normal factor definition can be applied to express more general forms of:

- Tensor sum definition as the sum of normal factors for all matrices:
  $$\begin{aligned} A \oplus B \oplus C = \quad & (A \otimes I_{n_B} \otimes I_{n_C}) + \\ & (I_{n_A} \otimes B \otimes I_{n_C}) + \\ & (I_{n_A} \otimes I_{n_B} \otimes C) \end{aligned}$$

[3]Although this property could be inferred from the *Compatibility with (ordinary matrix) multiplication*, it was defined by Fernandes, Plateau and Stewart [10].

- Compatibility over multiplication property as the product of normal factors for all matrices:
  $A \otimes B \otimes C = (A \otimes I_{n_B} \otimes I_{n_C}) \times (I_{n_A} \otimes B \otimes I_{n_C}) \times (I_{n_A} \otimes I_{n_B} \otimes C)$

2.2 Markovian Descriptor

The use of tensor algebra to represent large complex models in a structured description undeniably reduces the needs of memory. The SAN formalism, for example, takes advantage of this approach to represent the infinitesimal generator of the model. In fact, instead of defining one single, and usually huge, matrix of order equal to the product state space of the model, an equivalent algebraic formula is defined. For a SAN model with N automata and E synchronized events, this formula (Equation 1) is composed by a sum of $N + 2E$ tensor products with N matrices each:

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

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 $Q_{e_j^-}^{(i)}$).

As mentioned in the introduction, using another structured formalism, *e.g.*, superposed generalized stochastic Petri nets [8], an exactly equivalent definition of the descriptor could be employed by replacing the number of automata (N) by the number of modular subnets, and number of the synchronizing events (E) by the number of synchronized transitions. In a general case, the Markovian descriptor will be composed by $N \times (N + 2E)$ matrices as described in Table 1. So, the descriptor Q is composed of two separated parts: a tensor sum of matrices corresponding to the local events; and a sum of tensor products corresponding to the synchronizing events [10]. The tensor sum operation related to the local part of the descriptor can be decomposed into the ordinary sum of N tensor products called normal factors, *i.e.*, a sum of tensor products where all matrices but one are identity matrices. In Table 1, I_{n_i} is an identity matrix of order n_i, and n_i is the state space size of the i^{th} automaton, *i.e.*, the order of the matrix $Q^{(i)}$.

3 Vector-Descriptor Product

The basic operation to the iterative numerical solution of descriptor is the vector-descriptor product [10]. The vector-descriptor product operation corresponds to the product of a probability vector v, as big as the product state space, 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:

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

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 (see Table 1).

To simplify the algorithm descriptions in this paper, we consider only the basic operation *vector×tensor product term* omitting the j index from Expression 2, *i.e.*:

$\sum$	N		$\begin{array}{ccccccccc} Q_l^{(1)} & \otimes & I_{n_2} & \otimes & \cdots & \otimes & I_{n_{N-1}} & \otimes & I_{n_N} \\ I_{n_1} & \otimes & Q_l^{(2)} & \otimes & \cdots & \otimes & I_{n_{N-1}} & \otimes & I_{n_N} \\ & & & & \vdots & & & & \\ I_{n_1} & \otimes & I_{n_2} & \otimes & \cdots & \otimes & Q_l^{(N-1)} & \otimes & I_{n_N} \\ I_{n_1} & \otimes & I_{n_2} & \otimes & \cdots & \otimes & I_{n_{N-1}} & \otimes & Q_l^{(N)} \end{array}$
	$2E$	e^+	$\begin{array}{ccccccccc} Q_{e_1^+}^{(1)} & \otimes & Q_{e_1^+}^{(2)} & \otimes & \cdots & \otimes & Q_{e_1^+}^{(N-1)} & \otimes & Q_{e_1^+}^{(N)} \\ & & & & \vdots & & & & \\ Q_{e_E^+}^{(1)} & \otimes & Q_{e_E^+}^{(2)} & \otimes & \cdots & \otimes & Q_{e_E^+}^{(N-1)} & \otimes & Q_{e_E^+}^{(N)} \end{array}$
		e^-	$\begin{array}{ccccccccc} Q_{e_1^-}^{(1)} & \otimes & Q_{e_1^-}^{(2)} & \otimes & \cdots & \otimes & Q_{e_1^-}^{(N-1)} & \otimes & Q_{e_1^-}^{(N)} \\ & & & & \vdots & & & & \\ Q_{e_E^-}^{(1)} & \otimes & Q_{e_E^-}^{(2)} & \otimes & \cdots & \otimes & Q_{e_E^-}^{(N-1)} & \otimes & Q_{e_E^-}^{(N)} \end{array}$

Table 1: SAN Descriptor

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

The next three sections (Sections 3.1, 3.2 and 3.3) describe the known algorithms to perform this basic operation of multiplying a vector by a tensor product.

3.1 Sparse Algorithm

Considering a tensor product term of N matrices $Q^{(i)}$, each one of order n_i, and nz_i nonzero elements, the Sparse algorithm will evaluate the tensor product term generating one matrix of order $\prod_{i=1}^{N} n_i$, stored according to a sparse scheme. Then the vector v must be multiplied by the sparse matrix using an algorithm suitable to the sparse scheme used. For the most known sparse storage methods the computational cost in number of floating point multiplications of each vector-sparse matrix product can be optimal, *i.e.*, it will be equal to the number of nonzero elements of the resulting sparse matrix: $\prod_{i=1}^{N} nz_i$.

Each matrix entry is generated with $N-1$ scalar multiplications, combining one nonzero element of each matrix of the tensor product term. The computational cost of computing these entries is given by: $(N-1) \times \prod_{i=1}^{N} nz_i$.

Hence, the complete number of floating point multiplications of the Sparse algorithm will be the sum of both the cost of generating the entries and multiplying a vector by them, *i.e.*:

$$N \times \prod_{i=1}^{N} nz_i \tag{4}$$

3.2 Shuffle Algorithm

In this section, we briefly describe the Shuffle algorithm without any considerations about functional elements optimizations. Matrices reordering and generalized tensor algebra properties for the Shuffle

algorithm were already studied in [10]. All those optimizations are very important to reduce the overhead of evaluating functional elements, but such considerations are out of the scope of this paper. Nevertheless, the existence of functional elements in the descriptor, and consequently the use of generalized tensor products, do not imply in any loss of generality for the algorithms and methods described in this paper. On the contrary, a future analysis of the proposed Hybrid algorithm can be enhanced with considerations about the impact of functional elements (with their particular dependencies) in the descriptor.

Algorithm 1 Shuffle Algorithm - $r = v \times \otimes_{i=1}^{N} Q^{(i)}$

```
1:  for all i = 1, 2, ..., N do
2:      base = 0
3:      for all m = 0, 1, 2, ..., nleft_i − 1 do
4:          for all j = 0, 1, 2, ..., nright_i − 1 do
5:              index = base + j
6:              for all l = 0, 1, 2, ..., n_i − 1 do
7:                  z_in[l] = v[index]
8:                  index = index + nright_i
9:              end for
10:             multiply  z_out = z_in × Q^(i)
11:             index = base + j
12:             for all l = 0, 1, 2, ..., n_i − 1 do
13:                 v[index] = z_out[l]
14:                 index = index + nright_i
15:             end for
16:         end for
17:         base = base + ( nright_i × n_i)
18:     end for
19: end for
20: r = v
```

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:

$$\begin{aligned} Q^{(1)} \otimes Q^{(2)} \otimes \ldots \otimes Q^{(N-1)} \otimes Q^{(N)} = \\ (Q^{(1)} \otimes I_{n_2} \otimes \ldots \otimes I_{n_{N-1}} \otimes I_{n_N}) \ \times \\ (I_{n_1} \otimes Q^{(2)} \otimes \ldots \otimes I_{n_{N-1}} \otimes I_{n_N}) \ \times \\ \ldots \\ (I_{n_1} \otimes I_{n_2} \otimes \ldots \otimes Q^{(N-1)} \otimes I_{n_N}) \ \times \\ (I_{n_1} \otimes I_{n_2} \otimes \ldots \otimes I_{n_{N-1}} \otimes Q^{(N)}) \end{aligned} \tag{5}$$

Consequently, we can rewrite the basic operation, Expression (3), according to this property, *i.e.*:

$$v \times \left[\prod_{i=1}^{N} I_{nleft_i} \otimes Q^{(i)} \otimes I_{nright_i} \right] \tag{6}$$

where $nleft_i$ corresponds to the product of the order of all matrices before the i^{th} matrix of the tensor product term, $\prod_{k=1}^{i-1} n_k$ (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, $\prod_{k=i+1}^{N} n_k$ (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.

The Shuffle algorithm for a tensor product term with N matrices successively multiplies each of the N normal factors. The multiplication of the last normal factor ($I_{nleft_N} \otimes Q^{(N)}$) corresponds performing $nleft_N$ multiplications of a small (n_N) vector, called z_{in} in the Algorithm 1, by the last matrix ($Q^{(N)}$), and storing the result in another small vector, called z_{out}. Figure 1 illustrates this process.

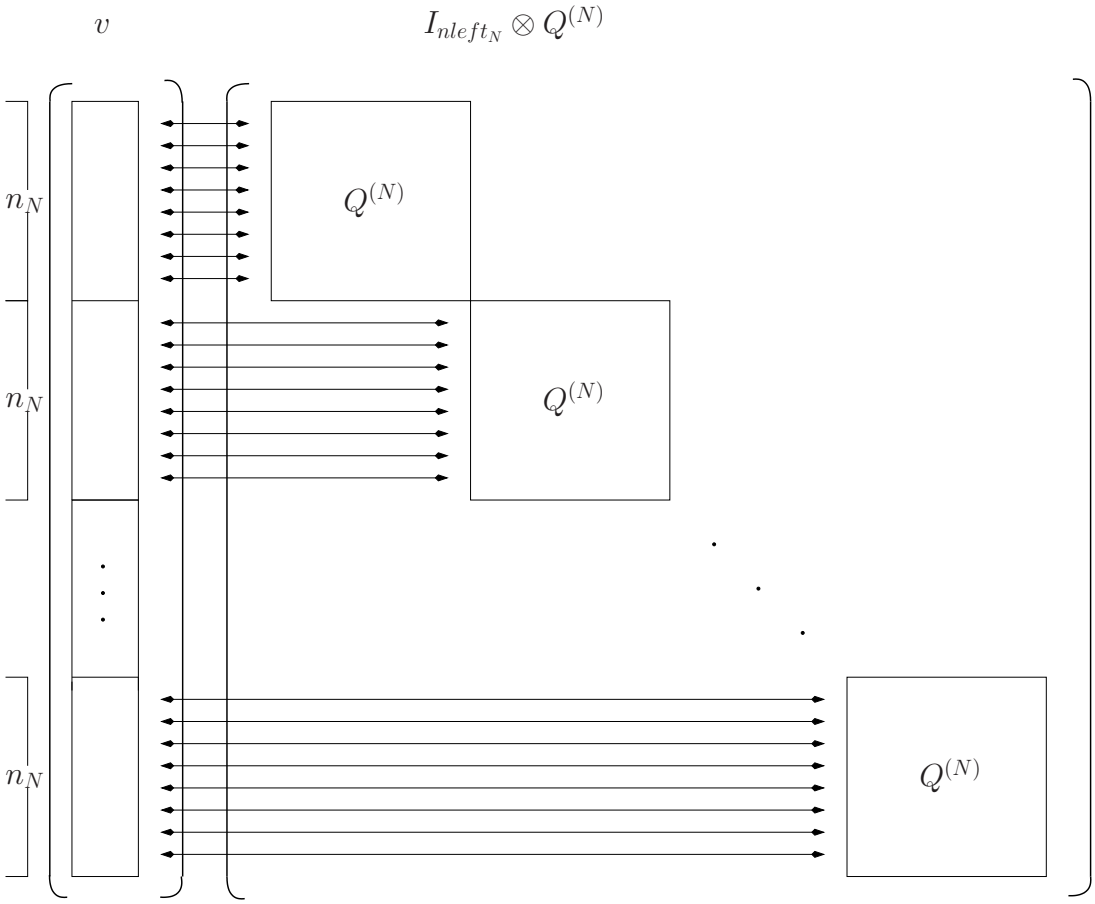


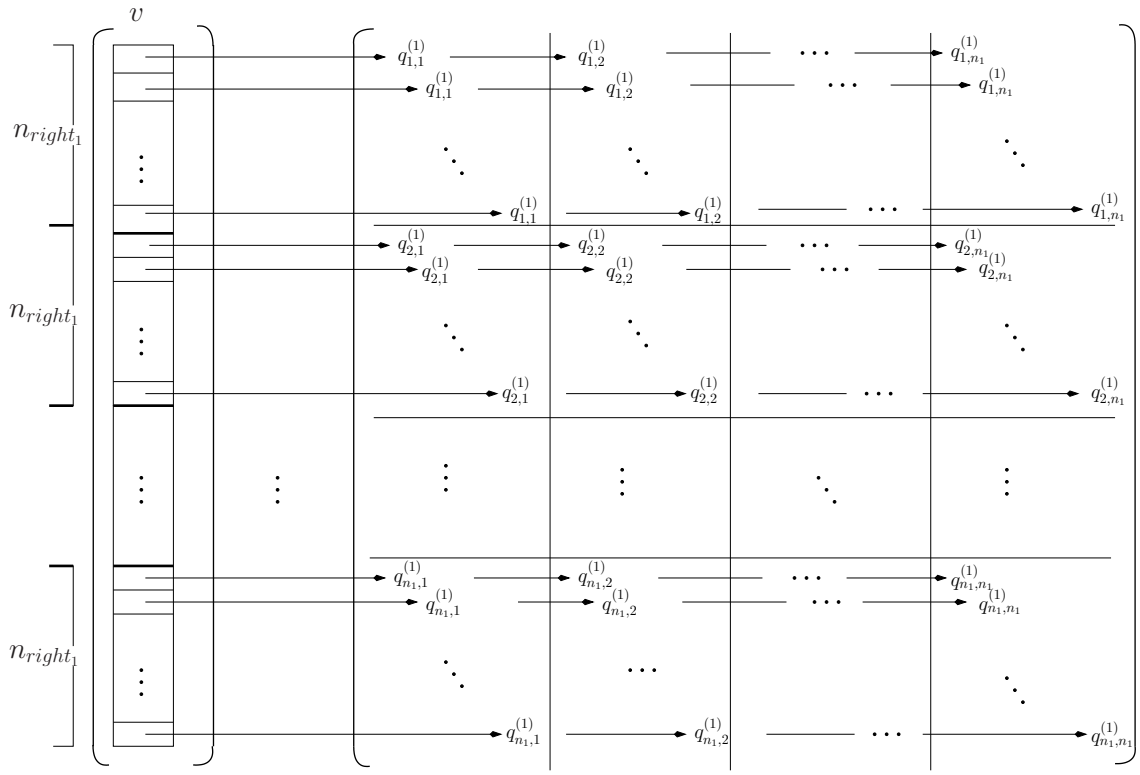
Figure 1: Multiplication of $v \times I_{nleft_N} \otimes Q^{(N)}$

Observing Figure 1, we may notice that the z_{in} vectors are easily located in the product state space sized vector as contiguous slices. This contiguousness is due to the fact that we are multiplying the last normal factor ($I_{nleft_N} \otimes Q^{(N)}$), *i.e.*, the normal factor that corresponds to $Q^{(N)}$ matrices in the diagonal blocks. A more complicated process to locate the z_{in} and z_{out} vectors is required to multiply the other normal factors ($I_{nleft_i} \otimes Q^{(i)} \otimes I_{nright_i}$), *i.e.*, the normal factors where the most inner blocks are diagonal matrices (see normal factors examples in tensor products definition - Section 2).

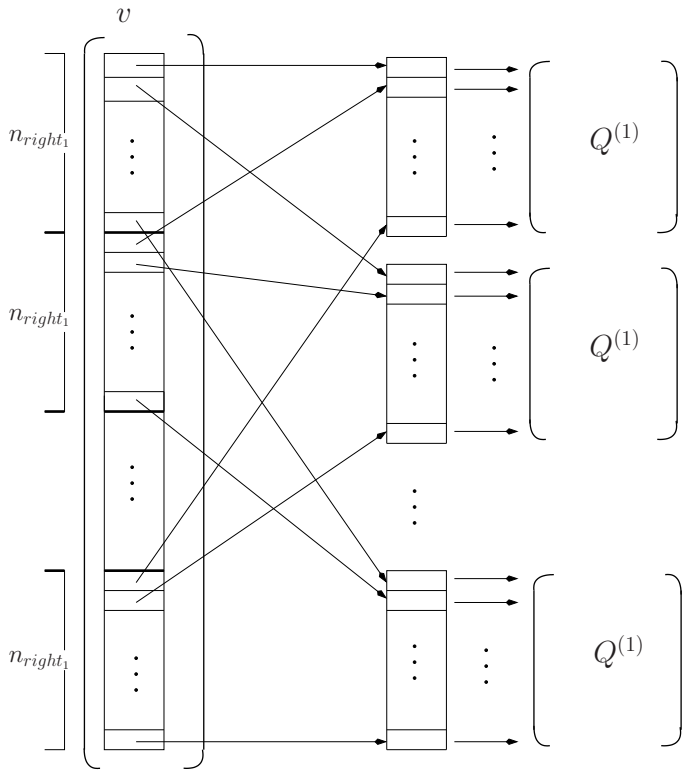
For example, the first normal factor ($Q^{(1)} \otimes I_{nright_1}$), *i.e.*, the normal factor composed by $(n_1)^2$ blocks containing diagonal matrices with the same element of matrix $Q^{(1)}$, can be treated by assembling the z_{in} vectors picking vector elements with a n_{right_1} interval. Figure 2 illustrates this process in two steps: the location of the elements in the input vector v (a) and the assembling of the z_{in} vectors (b). In this figure, we denote as $q_{i,j}^{(k)}$ the element of matrix $Q^{(k)}$ in row i and column j.

Generalizing this process for all normal factors, the multiplication of a vector v by the i^{th} normal factor consists in *shuffling* the elements of v in order to assemble $nleft_i \times nright_i$ vectors of size n_i and multiply them by matrix $Q^{(i)}$.

Hence, assuming that matrix $Q^{(i)}$ is stored as a sparse matrix, the number of needed multiplications to multiply a vector by the i^{th} normal factor is: $nleft_i \times nright_i \times nz_i$, where nz_i corresponds to the



(a)



(b)

Figure 2: Multiplication of $v \times Q^{(1)} \otimes I_{nright_1}$

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, the number of needed floating point multiplications (Shuffle computational cost) to perform the basic operation (multiplication of a vector by a tensor product) is [10]:

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

3.3 Slice Algorithm

The Slice algorithm [11] was proposed as an alternative to the Shuffle algorithm. It is based on the *Additive Decomposition* property [10], instead of the decomposition of a tensor product in the ordinary product of normal factors property (Equation 5). The Additive Decomposition property states that a tensor product term can be described by a sum of unitary matrices, *i.e.*, a sum of matrices in which there is only one nonzero element computed as the product of one nonzero element of each matrix, *i.e.*:

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

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 $q^{(k)}_{i,j}$.

Algorithm 2 Slice Algorithm - $r = v \times \otimes_{i=1}^{N} Q^{(i)}$

1: **for all** $i_1, \ldots, i_{N-1}, j_1, \ldots, j_{N-1} \in \theta(1 \ldots N-1)$ **do**
2: $e = 1$
3: $base_{in} = base_{out} = 0$
4: **for all** $k = 1, 2, \ldots, N-1$ **do**
5: $e = e \times q^{(k)}_{(i_k,j_k)}$
6: $base_{in} = base_{in} + (i_k \times nright_k)$
7: $base_{out} = base_{out} + (j_k \times nright_k)$
8: **end for**
9: **for all** $l = 0, 1, 2, \ldots, n_N - 1$ **do**
10: $v_{in}[l] = v[base_{in} + l] \times e$
11: **end for**
12: $multiply \;\; v_{out} = v'_{in} \times Q^{(N)}$
13: **for all** $l = 0, 1, 2, \ldots, n_N - 1$ **do**
14: $r[base_{out} + l] = r[base_{out} + l] + v_{out}[l]$
15: **end for**
16: **end for**

Defining $\theta(1 \ldots N)$ as the set of all possible combinations of nonzero elements of the matrices from $Q^{(1)}$ to $Q^{(N)}$, 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. For such unitary matrices, we use the following notation:

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

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 tensor product multiplied by the last matrix ($Q^{(N)}$). 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 Slice algorithm consists in computing the element e of an Additive Unitary Normal Factor (*AUNF*), finding the slice of the input vector v according to the coordinate of the matrices elements used to compute element e, and then multiplying it all by the last matrix ($Q^{(N)}$) of the tensor product. This operation is repeated for each *AUNF*, *i.e.*, for each possible combination of nonzero elements of the first $N-1$ matrices of the tensor product.

The number of needed floating point multiplications (Slice computational cost) 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 the sliced vector v_s (n_N), and the cost to multiply v_s by the last matrix $Q^{(N)}$ (nz_N), *i.e.*[11]:

$$\prod_{i=1}^{N-1} nz_i \times \left[(N-2) + n_N + nz_N \right] \tag{9}$$

4 Hybrid Algorithm

The Slice algorithm has shown a better overall performance than the traditional Shuffle algorithm for the most of real case SAN examples [11]. 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 are naturally sparse. The local part of a descriptor is intrinsically very sparse due to the tensor sum structure. The synchronizing events are mostly used to describe exceptional behaviors, therefore it lets this part of the descriptor also quite sparse. As a matter of fact, the Slice algorithm offers a trade-off between the unique sparse matrix approach used for straightforward Markov chains and the pure tensor approach of the Shuffle algorithm.

The Slice algorithm handles the first $N-1$ matrices with a Sparse-like approach, and the last one with a Shuffle-like approach. In fact, for many SAN models, actual tensor terms found in the synchronizing events parts fits Slice algorithm very well because only one matrix may not be sparse (the matrix holding the rates of the event itself). However, some SAN models present more complex events where several automata are synchronized. Such cases will generate tensor product terms with (possibly) many rather dense matrices. To handle those tensor product terms, it would be desirable to have an algorithm considering the matrices in two different groups: the first one with the sparse matrices; and the second one with the matrices with a larger number of nonzero elements. A Sparse-like approach could be applied to the first group of matrices generating *AUNF*s. Each one of those *AUNF*s should be *tensorly* multiplied by the second group of matrices using a Shuffle-like approach. Considering that, the Hybrid algorithm is a generalization of the Slice algorithm, where all matrices but the last are handled with a Sparse-like approach, and the last matrix is handled with a Shuffle-like approach. In fact, the Hybrid consists in:

- the definition of two sets of matrices;
- the re-ordering of the matrices putting the first group in the left hand side, and the second group in the right side;
- handle the first group of matrices with a Sparse-like approach, *i.e.*, generating *AUNF*s;
- handle the second group of matrices with a Shuffle-like approach.

Algorithm 3 Hybrid Algorithm - $r = v \times \otimes_{i=1}^{N} Q^{(i)}$

for all $i_1, \ldots, i_\sigma, j_1, \ldots, j_\sigma \in \theta(1 \ldots \sigma)$ **do**
  $e = 1$
  $base_{in} = base_{out} = 0$
  for all $k = 1, 2, \ldots, \sigma$ **do**
    $e = e \times q^{(k)}_{(i_k, j_k)}$
    $base_{in} = base_{in} + (i_k \times nright_k)$
    $base_{out} = base_{out} + (j_k \times nright_k)$
  end for
  for all $l = 0, 1, 2, \ldots, nright_\sigma - 1$ **do**
    $v_{in}[l] = v[base_{in} + l] \times e$
  end for
  for all $i = \sigma + 1 \ldots, N$ **do**
    $base = 0$
    for all $m = 0, 1, 2, \ldots, nleft_i - 1$ **do**
      for all $j = 0, 1, 2, \ldots, nright_i - 1$ **do**
        $index = base + j$
        for all $l = 0, 1, 2, \ldots, n_i - 1$ **do**
          $z_{in}[l] = v_{in}[index]$
          $index = index + nright_i$
        end for
        $multiply \;\; z_{out} = z_{in} \times Q^{(i)}$
        $index = base + j$
        for all $l = 0, 1, 2, \ldots, n_i - 1$ **do**
          $v_{in}[index] = z_{out}[l]$
          $index = index + nright_i$
        end for
      end for
      $base = base + (nright_i \times n_i)$
    end for
  end for
  for all $l = 0, 1, 2, \ldots, nright_\sigma - 1$ **do**
    $r[base_{out} + l] = r[base_{out} + l] + v_{in}[l]$
  end for
end for

Therefore, the application of the Hybrid algorithm is very dependant of the choice of matrices in each group. It is out of the scope of this paper to present a deep analysis of the impact of choosing this groups. In fact, the proposal of an heuristic to choose which matrices should be placed in each group seems to be quite complex and should probably take into account much more than just the level of sparseness of the matrices, but also consider, for example, the permutation costs and possibly many other characteristics.

In the context of this paper, we are concerned only by the general characteristics of the Hybrid algorithm assuming that the matrices of a tensor product are presented in a pre-defined order and one matrix ($Q^{(\sigma)}$) is indicated to be the last matrix to be handled by the Sparse-like approach. Table 2 presents the general idea of the Hybrid algorithm graphically. In this table, it is possible to observe that the Sparse ($\sigma = N$), the Slice ($\sigma = N-1$) and the Shuffle ($\sigma = 0$) are also particular cases of the Hybrid algorithm.

Algorithm 3 defines formally the steps of the Hybrid approach. It consists in the computation of the element e of each *AUNF* by multiplying one nonzero element of each matrix of the first set of matrices (from $Q^{(1)}$ to $Q^{(\sigma)}$). According to the elements row indices used to generate element e, a contiguous slice of input vector v is taken (v_{in}). Vector v_{in} is multiplied by element e and the resulting vector (v'_{in}) is used as input vector to the Shuffle-like multiplication by the tensor product of the matrices in the second set of matrices (from $Q^{(\sigma+1)}$ to $Q^{(N)}$).

σ	Tensor Product Term	Algorithm
0	$\overset{\sigma}{\downarrow}$ $Q^{(1)} \otimes Q^{(2)} \otimes \ldots \otimes Q^{(N-3)} \otimes Q^{(N-2)} \otimes Q^{(N-1)} \otimes Q^{(N)}$ $\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{shuffle}$	Shuffle
1	$\overset{\sigma}{\downarrow}$ $Q^{(1)} \otimes Q^{(2)} \otimes \ldots \otimes Q^{(N-3)} \otimes Q^{(N-2)} \otimes Q^{(N-1)} \otimes Q^{(N)}$ $\underbrace{\qquad}_{sparse}$ $\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{shuffle}$	
2	$Q^{(1)} \otimes \overset{\sigma}{\downarrow} Q^{(2)} \otimes \ldots \otimes Q^{(N-3)} \otimes Q^{(N-2)} \otimes Q^{(N-1)} \otimes Q^{(N)}$ $\underbrace{\qquad\qquad}_{sparse}$ $\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{shuffle}$	
$\vdots$	$\vdots$	
N-2	$Q^{(1)} \otimes Q^{(2)} \otimes \ldots \otimes Q^{(N-3)} \otimes \overset{\sigma}{\downarrow} Q^{(N-2)} \otimes Q^{(N-1)} \otimes Q^{(N)}$ $\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{sparse}$ $\underbrace{\qquad\qquad\qquad}_{shuffle}$	
N-1	$Q^{(1)} \otimes Q^{(2)} \otimes \ldots \otimes Q^{(N-3)} \otimes Q^{(N-2)} \otimes \overset{\sigma}{\downarrow} Q^{(N-1)} \otimes Q^{(N)}$ $\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{sparse}$ $\underbrace{\qquad}_{shuffle}$	Slice
N	$Q^{(1)} \otimes Q^{(2)} \otimes \ldots \otimes Q^{(N-3)} \otimes Q^{(N-2)} \otimes Q^{(N-1)} \otimes \overset{\sigma}{\downarrow} Q^{(N)}$ $\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{sparse}$	Sparse

Table 2: Hybrid approach as a generalization of traditional algorithms

The computational cost (number of floating point multiplications) of the Hybrid algorithm is computed taking into account the number of multiplication performed to generate the nonzero element of each *AUNF* ($\sigma-1$), plus the number of multiplications of e by the elements of vector v_{in} ($\prod_{i=\sigma+1}^{N} n_i$), plus the cost to multiply the input vector v'_{in} by the tensor product of the last matrices ($\prod_{i=\sigma+1}^{N} n_i \sum_{i=\sigma+1}^{N} \frac{nz_i}{n_i}$). Hence, the computational cost to multiply one single *AUNF* is:

$$\left((\sigma-1) + \prod_{i=\sigma+1}^{N} n_i\right) + \left(\prod_{i=\sigma+1}^{N} n_i \sum_{i=\sigma+1}^{N} \frac{nz_i}{n_i}\right)$$

The total Hybrid computational cost is simply the number of unitary matrices ($\prod_{i=1}^{\sigma} nz_i$) times the cost to handle each *AUNF*, *i.e.*:

$$\prod_{i=1}^{\sigma} nz_i \quad \times \quad \left[\left((\sigma-1)+\prod_{i=\sigma+1}^{N} n_i\right)+\left(\prod_{i=\sigma+1}^{N} n_i \sum_{i=\sigma+1}^{N} \frac{nz_i}{n_i}\right)\right] \tag{10}$$

5 Numerical Analysis

For the numerical analysis in this paper we will consider five examples of tensor products with the matrices sizes (n_i) and numbers of nonzero elements (nz_i) stated in Table 3. When a matrix of an example is an identity, an indication of its type and size (I_j) replaces the n_i and nz_i information, *e.g.*, I_8 indicates an identity matrix of order 8.

Matrices	Tensor Product Examples									
	ex_1		ex_2		ex_3		ex_4		ex_5	
	n_1	nz_1	n_2	nz_2	n_3	nz_3	n_4	nz_4	n_5	nz_5
$Q^{(1)}$	3	4	4	3	3	1	6	1	3	1
$Q^{(2)}$	I_8		8	8	6	4	3	1	3	1
$Q^{(3)}$	I_6		I_3		5	4	3	1	4	2
$Q^{(4)}$	I_6		I_3		I_3		4	2	8	5
$Q^{(5)}$	I_5		I_4		I_4		8	5	4	3
$Q^{(6)}$	I_5		I_5		I_4		4	3	5	4
$Q^{(7)}$	I_4		I_5		I_5		5	4	6	5
$Q^{(8)}$	I_4		I_6		I_6		I_5		5	5
$Q^{(9)}$	I_3		I_6		I_8		I_6		6	6

Table 3: Tensor product examples

The examples in Table 3 were taken from actual tensor product terms corresponding to events of a real SAN model. They represent different possible situations of real events:

- Example ex_1 corresponds to a local event, then all matrices but one are identities;
- Example ex_2 corresponds to a synchronizing event which synchronizes just two automata (this is the most common type of synchronizing event in SAN models);
- Example ex_3 corresponds to a synchronizing event which synchronizes three automata;
- Example ex_4 corresponds to a synchronizing event which synchronizes seven, *i.e.*, almost all, automata; and
- Example ex_5 corresponds to a synchronizing event which synchronizes all nine automata.

These examples have their matrices ordered according to the authors best knowledge. We would like to stress that is not the purpose of this paper to discuss a good heuristic to find the best matrices order. Our goal is to verify some numerical issues about the Hybrid algorithm in comparison with the other known algorithms (Sparse, Shuffle, and Slice).

	ex_1		ex_2		ex_3		ex_4		ex_5	
σ	Eq. 10	Eq. 12	Eq. 10	Eq. 12	Eq. 10	Eq. 12	Eq. 10	Eq. 12	Eq. 10	Eq. 12
0	9,676,800	**1,382,400**	9,072,000	1,814,400	8,087,040	1,866,240	5,711,040	3,637,440	6,402,240	6,402,240
1	12,441,600	1,382,400	6,998,400	1,555,200	2,926,080	852,480	1,095,840	750,240	2,364,480	2,364,480
2	11,059,200	1,382,430	6,220,820	**777,624**	1,797,120	414,724	346,081	230,881	749,761	749,761
3	9,677,180	1,382,780	5,443,340	777,744	1,290,270	**184,352**	108,962	70,562	346,084	346,084
4	8,297,860	1,385,860	4,666,250	778,248	1,106,060	184,464	49,686	30,486	193,830	193,830
5	6,935,040	1,405,440	3,891,460	781,056	922,368	185,088	27,340	15,340	125,220	125,220
6	5,673,600	1,526,400	3,132,000	799,200	741,120	188,160	17,250	8,250	83,400	83,400
7	4,838,400	2,073,600	2,462,400	907,200	576,000	207,360	11,520	**4,320**	57,600	57,600
8	5,990,400	4,608,000	2,462,400	1,684,800	529,920	345,600	11,400	7,800	57,000	**57,000**
9	12,441,600	12,441,600	6,998,400	6,998,400	1,658,880	1,658,880	32,400	32,400	162,000	162,000

Table 4: Computational Cost according to Expressions 10 and 12

5.1 Handling Identity Matrices

The Shuffle algorithm computational cost presented in [10] (Expression 7) considers the processing of all multiplications including the normal factors composed only by identities. However, practical implementations of the Shuffle algorithm [4] skip the identity matrices, dealing only with the normal factors where the (supposed) non-identity matrix is actually a non-identity. In Example ex_1 for instance, the Shuffle algorithm would decompose the term $\bigotimes_{i=1}^{9} Q^{(i)}$ using decomposition in product of normal factors property (see Equation 5) in one interesting normal factor ($Q^{(1)} \otimes I_{nright_1}$) and eight normal factors that will correspond to $\prod_{i=1}^{9} n_i$ sized identity matrices. Obviously, just the first normal factor should be multiplied by the vector v and all other eight normal factors could be skipped. In the case of Example ex_1, it would correspond to reduce the number of floating point multiplications from 9,676,800 (computed according to Expression 7) to just 1,382,400 multiplications. Stating this optimization in a formula, Expression 7 can be replaced by:

$$\sum_{\substack{i=1,\\ \text{where } Q^{(i)} \neq I_{n_i}}}^{N} nleft_i \times nright_i \times nz_i \tag{11}$$

This pure numerical analysis, suggests the same skipping-identities optimization to the Shuffle-like part of the Hybrid algorithm. It will correspond to make a slight change in the Hybrid algorithm (Algorithm 3) with the inclusion of a test (an **if**) to only execute the operations inside the **for** loop in line 12 (commands from lines 13 to 29) if the matrix indexed by variable i of the algorithm ($Q^{(i)}$) is not an identity matrix. Considering this optimization, the Hybrid algorithm computational cost (Expression 10) will be replaced by:

$$\begin{aligned} &\textstyle\prod_{i=1}^{\sigma} nz_i \times \Big[\Big((\sigma-1) + \prod_{i=\sigma+1}^{N} n_i\Big) + \\ &\Big(\sum_{i=\sigma+1}^{N} \Big[\prod_{j=\sigma+1}^{i-1} n_j\Big] \times \Big[\prod_{j=i+1}^{N} n_j\Big] \times nz_i\Big)\Big] \end{aligned} \tag{12}$$

Table 4 presents the reduction in terms of computational cost for this Identities optimization in all possible versions of the Hybrid algorithm for the five tensor product examples. It is important to remind that Hybrid option $\sigma = 0$, $\sigma = 8$, and $\sigma = 9$ corresponds respectively to previous algorithms Shuffle, Slice and Sparse.

First of all, we may observe that a best theoretical performance was achieved with Hybrid options that were not available in previous algorithms for three examples, namely: $\sigma = 2$ for example ex_2; $\sigma = 3$ for example ex_3; and $\sigma = 7$ for example ex_4. For Example ex_1, the Shuffle algorithm ($\sigma = 0$) and for Example ex_5, the Slice algorithm ($\sigma = 8$) already have the best performances. The second logic observation from this table was the quite impressive gains with the Identities optimization for examples

with a large number of identities (ex_1 and ex_2). For models with a fewer number of identities, these gains decrease progressively until Example ex_5 which was not affected at all since it has no identity matrices.

5.2 Precomputing nonzero *AUNF* elements

The second intuitive optimization is also based on a theoretical observation of the Hybrid algorithm. Usually the tensor product terms of a SAN model are very sparse (a few thousands nonzero elements). The only cases where a more significant number of nonzero elements is found are those when we are dealing with a term with many identity matrices. This can be verified in Table 5 by the number of *AUNF*s generated in any of the possible versions of the Hybrid algorithm for the five examples presented (column *AUNF*s). It is important to recall that to each *AUNF* a scalar is computed as the product of an element of each matrix. Examples ex_1 and ex_2 have around a million *AUNF*s (1,382,400 and 777,600 respectively), but in Example ex_3 this number drops significantly to 184,320 and in Examples ex_4 and ex_5 we observe just 3,600 and 18,000 *AUNF*s respectively.

Luckily, for the examples with a large number of nonzero elements, the best Hybrid options are far from the pure Sparse approach ($\sigma = 0$, 2 and 3 for Examples ex_1, $ex2$ and ex_3 respectively - see Table 4). Therefore, the precomputation of these nonzero elements, and their storage, could be considered as a valid option. It results in another reduction of the computational cost of the Hybrid algorithm (Expression 12) to remove the part corresponding to computing the nonzero elements of each *AUNF* ($\sigma - 1$ multiplications - line 5 of Algorithm 3). Hence, Expression 12 will be replaced by:

$$\begin{aligned} &\prod_{i=1}^{\sigma} nz_i \times \left[\left(\prod_{i=\sigma+1}^{N} n_i \right) + \right. \\ &\left. \left(\sum_{i=\sigma+1}^{N} \left[\prod_{j=\sigma+1}^{i-1} n_j \right] \times \left[\prod_{j=i+1}^{N} n_j \right] \times nz_i \right) \right] \end{aligned} \tag{13}$$

Observing Table 5 our first impression is that there are remarkable computational cost gains by using the Elements Precomputation optimization. However, those gains are somewhat deceiving if we focus on the best solution (indicated in boldface for each example). Example ex_1 has no gains, Examples ex_2 and ex_3 have irrelevant gains, and only ex_4 presented some perceptible gains (reduction from 4,320 to 3,600 multiplications). Example ex_5 presented impressive gains, but this is not surprising since the best Hybrid option for this example was the pure Sparse approach which, obviously, is much more effective if the nonzero elements do not have to be recomputed at each vector multiplication. The upside of this analysis is the fact that the memory increment (column *mem.* in Table 5) necessary to this optimization is irrelevant to the most effective Hybrid choices.

5.3 Practical Analysis

The numerical results for this section were obtained running a prototype of the Hybrid algorithm with both optimizations presented (Sections 5.1 and 5.2) on a 2.2 GHz Pentium IV Xeon under Linux operating system with 2 GBytes of memory. The Hybrid algorithm prototype was compiled using `g++` compiler with no compiling optimizations. Table 6 shows the Computational cost (Eq. 13) and the CPU user time spent in seconds (column *time*) to perform one vector-descriptor multiplication considering all Hybrid algorithm options ($\sigma = 0..9$) to the five examples presented. For each possible option of each example, 5,000 multiplications were performed in order to establish the mean values presented.

Observing the results of Table 6, we notice that the computational cost does not precisely indicates which option will be the best choice to each example. In fact, only for Examples ex_3 and ex_4 there was a correlation between the CPU times and the computational cost. However, the Hybrid intermediary options were faster for Examples ex_1 and ex_5, where respectively the Shuffle ($\sigma = 0$) and Sparse ($\sigma = 9$) solution were recommended by computational cost. This fact demands further analysis, since many factors could be affecting the results. For example, the number of floating point additions, or fixed point multiplications are quite different according to the Hybrid choice of σ, and the number of those

σ	Eq. 12	Eq. 13	*AUNF*'s	*mem.*
		ex_1		
0	1,382,400	**1,382,400**	-	**-**
1	1,382,400	1,382,400	4	0.1Kb
2	1,382,430	1,382,400	32	1.0Kb
3	1,382,780	1,382,400	192	6.0Kb
4	1,385,860	1,382,400	1,152	36.0Kb
5	1,405,440	1,382,400	5,760	0.2Mb
6	1,526,400	1,382,400	28,800	0.9Mb
7	2,073,600	1,382,400	115,200	3.5Mb
8	4,608,000	1,382,400	460,800	14.0Mb
9	12,441,600	1,382,400	1,382,400	42.2Mb
		ex_2		
0	1,814,400	1,814,400	-	-
1	1,555,200	1,555,200	3	0.1Kb
2	777,624	**777,600**	24	**0.8Kb**
3	777,744	777,600	72	2.3Kb
4	778,248	777,600	216	6.8Kb
5	781,056	777,600	864	27.0Kb
6	799,200	777,600	4,320	0.1Mb
7	907,200	777,600	21,600	0.7Mb
8	1,684,800	777,600	129,600	4.0Mb
9	6,998,400	777,600	777,600	23.7Mb
		ex_3		
0	1,866,240	1,866,240	-	-
1	852,480	852,480	1	0.0Kb
2	414,724	414,720	4	0.1Kb
3	184,352	**184,320**	16	**0.5Kb**
4	184,464	184,320	48	1.5Kb
5	185,088	184,320	192	6.0Kb
6	188,160	184,320	768	24.0Kb
7	207,360	184,320	3,840	0.1Mb
8	345,600	184,320	23,040	0.7Mb
9	1,658,880	184,320	184,320	5.6Mb
		ex_4		
0	3,637,440	3,637,440	-	-
1	750,240	750,240	1	0.0Kb
2	230,881	230,880	1	0.0Kb
3	70,562	70,560	1	0.0Kb
4	30,486	30,480	2	0.0Kb
5	15,340	15,300	10	0.3Kb
6	8,250	8,100	30	1.0Kb
7	4,320	**3,600**	120	**3.8Kb**
8	7,800	3,600	600	18.8Kb
9	32,400	3,600	3,600	0.1Mb
		ex_5		
0	6,402,240	6,402,240	-	-
1	6,402,240	6,402,240	1	0.0Kb
2	749,761	749,760	1	0.0Kb
3	346,084	346,080	2	0.0Kb
4	193,830	193,800	10	0.3Kb
5	125,220	125,100	30	1.0Kb
6	83,400	82,800	120	3.8Kb
7	57,600	54,000	600	18.8Kb
8	57,000	36,000	3,000	0.1Mb
9	162,000	**18,000**	18,000	**0.6Mb**

Table 5: Computational Cost without (Eq. 12) and with (Eq. 13) Elements Precomputation optimization

	ex_1		ex_2		ex_3		ex_4		ex_5	
σ	Eq. 13	*time*	Eq. 13	*time*	Eq. 13	*time*	Eq. 13	*time*	Eq. 13	*time*
0	**1,382,400**	0.27	1,814,400	0.37	1,866,240	0.42	3,637,440	0.78	6,402,240	1.08
1	1,382,400	0.28	1,555,200	0.19	852,480	0.16	750,240	0.17	6,402,240	0.37
2	1,382,400	0.25	**777,600**	0.18	414,720	0.10	230,880	0.10	749,760	0.16
3	1,382,400	**0.23**	777,600	0.15	**184,320**	**0.09**	70,560	**0.07**	346,080	0.10
4	1,382,400	**0.23**	777,600	**0.14**	184,320	**0.09**	30,480	**0.07**	193,800	0.08
5	1,382,400	**0.23**	777,600	0.16	184,320	**0.09**	15,300	**0.07**	125,100	0.08
6	1,382,400	0.26	777,600	0.16	184,320	**0.09**	8,100	**0.07**	82,800	0.07
7	1,382,400	0.33	777,600	0.17	184,320	**0.09**	**3,600**	**0.07**	54,000	**0.06**
8	1,382,400	0.55	777,600	0.24	184,320	0.10	3,600	**0.07**	36,000	0.07
9	1,382,400	12.33	777,600	6.91	184,320	1.69	3,600	0.10	**18,000**	0.22

Table 6: Execution times using all optimizations

operations may be almost as relevant as the number of floating point multiplications. Also, some studies about memory localization of data, suitability to pipeline acceleration devices and other CPU features could be done in order to explain the discrepancy found in Table 6.

6 Conclusion

The main purpose of this paper is the proposition of a new vector-descriptor algorithm and the analysis of some of its numerical aspects. Initial theoretical results presented in Sections 5.1 and 5.2 suggest the best efficiency of the Hybrid algorithm. This was observed to tensor product terms that do not have too many identity matrices or no identities at all. Our experience with structured models suggests that these tensor product terms, with a reasonable number of identity matrices, are the most commonly encountered ones. Even more encouraging results were found with a practical analysis of the algorithm prototype (Section 5.3).

To verify the results of this prototype to a full SAN model, we solved a SAN real model originally presented in [3]. This model has nine automata and nineteen synchronizing events, resulting in a Markovian descriptor with forty-seven tensor terms. The experience carried out with the Hybrid prototype has considerably improved the solution time with an irrelevant increase in memory costs. Numerically speaking, this model was solved in less than 30% of the time usually spent by the application of the Shuffle algorithm. The choice of matrices ordination and σ for each tensor product term was hand-made by the authors. However, the research for an heuristic to automatically choose the best order and σ to each tensor product demands a much more thorough study of the Hybrid algorithm.

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. D. 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. *Electronic Notes In Theoretical Computer Science*, 128(4):101–121, April 2005.

[4] 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.

[5] 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.

[6] P. Buchholz and P. Kemper. Hierarchical reachability graph generation for Petri nets. *Formal Methods in Systems Design*, 21(3):281–315, 2002.

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

[8] 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.

[9] F. L. Dotti and L. Ribeiro. Specification of Mobile Code Systems using Graph Grammars. In *Formal Methods for Open Object-Based Distributed Systems IV*, pages 45–63, Stanford, USA, 2000. Kluwer Academic Publishers.

[10] 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.

[11] P. Fernandes, R. Presotto, A. Sales, and T. Webber. An Alternative Algorithm to Multiply a Vector by a Kronecker Represented Descriptor. In 21^{st} *UK Performance Engineering Workshop (Accepted)*, pages 57–67, Newcastle, UK, June 2005.

[12] 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.

[13] A. S. Miner. Efficient solution of GSPNs using Canonical Matrix Diagrams. In 9^{th} *International Workshop on Petri Nets and Performance Models (PNPM'01)*, pages 101–110, Aachen, Germany, September 2001. IEEE Computer Society Press.

[14] 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.

[15] A. S. Miner and G. Ciardo. Efficient Reachability Set Generation and Storage Using Decision Diagrams. In *Proceedings of the* 20^{th} *International Conference on Applications and Theory of Petri Nets*, volume 1639 of *LNCS*, pages 6–25, Williamsburg, VA, USA, June 1999. Springer-Verlag Heidelberg.

[16] 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.

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

[18] W. J. Stewart. *MARCA: Markov chain analyzer. A software package for Markov modeling*, pages 37–62. Numerical Solution of Markov Chains. M. Dekker Inc., New York, 1991.