REVUE FRANÇAISE D'AUTOMATIQUE, D'INFORMATIQUE ET DE RECHERCHE OPÉRATIONNELLE. RECHERCHE OPÉRATIONNELLE

PAULO FERNANDES
BRIGITTE PLATEAU
WILLIAM J. STEWART

Optimizing tensor product computations in stochastic automata networks

Revue française d'automatique, d'informatique et de recherche opérationnelle. Recherche opérationnelle, tome 32, n° 3 (1998), p. 325-351.

<http://www.numdam.org/item?id=RO_1998__32_3_325_0>

Recherche opérationnelle/Operations Research
(vol. 32, n° 3, 1998, pp. 325-351)

OPTIMIZING TENSOR PRODUCT COMPUTATIONS IN STOCHASTIC AUTOMATA NETWORKS

by Paulo FERNANDES ([1]), Brigitte PLATEAU ([1]) and William J. STEWART ([2])

Abstract. – *In this paper we consider some numerical issues in computing solutions to networks of stochastic automata (SAN). In particular our concern is with keeping the amount of computation per iteration to a minimum, since iterative methods appear to be the most effective in determining numerical solutions. In a previous paper we presented complexity results concerning the vector-descriptor multiplication phase of the analysis. In this paper our concern is with optimizations related to the implementation of this algorithm. We also consider the possible benefits of grouping automata in a SAN with many small automata, to create an equivalent SAN having a smaller number of larger automata.* © Elsevier, Paris

Keywords: Markov chains, Stochastic automata networks, Generalized tensor algebra, Vector-descriptor multiplication.

Résumé. – *Dans cet article, nous étudions les algorithmes numériques pour la résolution de modèles de réseaux d'automates stochastiques. Plus particulièrement, nous montrons comment il est possible de réduire le coût de calcul d'une itération, puisque les schémas itératifs apparaissent les plus efficaces dans ce contexte. Dans un article précédent, nous avons donné des résultats de complexité pour la multiplication vecteur-descripteur. Dans cet article, nous nous intéressons aux optimisations liées à l'implantation de l'algorithme. De plus, nous montrons l'intérêt de grouper les automates en un réseau équivalent qui comporte moins d'automates, ceux-ci étant de taille plus grande.* © Elsevier, Paris

Mots clés : Chaîne de Markov, Réseaux d'automates stochastiques, Produit tensoriel généralisé, multiplication vecteur-descripteur.

1. INTRODUCTION

The use of *Stochastic Automata Networks* (SANs) is becoming increasingly important in performance modelling issues related to parallel and distributed computer systems. Models that are based on Markov chains allow for considerable complexity, but in practice they often suffer from difficulties that are well-documented. The size of the state space generated may become

([1]) IMAG–LMC, 100 rue des Mathématiques, 38041 Grenoble Cedex, France.
([2]) Department of Computer Science, North Carolina State University, Raleigh, N.C. 27695-8206, USA.
E-mail: Paulo.Fernandes@imag.fr, Brigitte.Plateau@imag.fr, billy@csc.ncsu.edu

Recherche opérationnelle/Operations Research, 0399-0559/98/03/

so large that it effectively prohibits the computation of a solution. This is true whether the Markov chain results from a stochastic Petri net formalism, or from a straightforward Markov chain analyzer [18, 12].

In many instances, the SAN formalism is an appropriate choice. Parallel and distributed systems are often viewed as collections of components that operate more or less independently, requiring only infrequent interaction such as synchronizing their actions, or operating at different rates depending on the state of parts of the overall system. This is exactly the viewpoint adopted by SANs [15, 20]. The components are modelled as individual stochastic automata that interact with each other. Furthermore, the state space explosion problem associated with Markov chain models is mitigated by the fact that the state transition matrix is not stored, nor even generated. Instead, it is represented by a number of much smaller matrices, one for each of the stochastic automata that constitute the system, and from these all relevant information may be determined without explicitly forming the global matrix. The implication is that a considerable saving in memory is effected by storing the matrix in this fashion [5, 15].

Other modelling approaches such as Petri Nets [6] and Stochastic Process Algebras [10, 11] can benefit from this tensor representation of the transition matrix. But as far as we know, these approaches do not exploit the functional extension of tensor algebra that is extensively used in SANs. For Superposed Petri Nets, in [21], it has been shown that the structured tensor representation can also be efficiently used to reduce the storage space of the iteration vectors.

There are two overriding concerns in the application of any Markovian modelling methodology, viz., memory requirements and computation time. Since these are frequently functions of the number of states, a first approach is to develop techniques that minimize the number of states in the model. In SANs, it is possible to make use of symmetries as well as lumping and various superpositioning of the automata to reduce the computational burden, [1, 4, 17]. Furthermore, in [9], structural properties of the Markov chain graph (specificially the occurrence of cycles) are used to compute steady state solutions.

For very large problems, it is well-known that only iterative methods are viable. We have shown in [20] that projection methods (Arnoldi, GMRES) with preconditioning can be used with a gain of performance compared to the standard power method. In this paper we concentrate on procedures that allow us to keep the amount of computation per iteration, which is basically the cost of the matrix-vector multiplication, to a minimum. In a previous paper [8], we proved a theorem concerning the complexity of a matrix-vector

multiplication when the matrix is stored as a compact SAN descriptor and has functional rates. This algorithm is an improvement of the one in [15] when functional rates must be handled.

The objective of this paper is to analyze the cost of implementing this new algorithm and to compare it to more usual sparse methods. In this way we extend the results reported in [7]. We show in these experiments that the actual performance of the implementation is model dependant. Nevertheless, it is shown that some optimizations can always be used to improve the performance of the basic algorithm: They concern the ordering of matrix multiplications (small matrices used to build the transition matrix in a structured way) and the benefits of reducing the number of automata of a model using an automatic grouping procedure. In this grouping procedure, a SAN with many small automata is transformed into an equivalent SAN with less larger automata and we examine the memory/performance trade-offs.

2. THE SAN DESCRIPTOR AND EXAMPLES

There are basically two ways in which stochastic automata interact [14]: The rate at which a transition may occur in one automaton may be a *function* of the state of other automata. Such transitions are called *functional* transitions. A transition in one automaton may *force* a transition to occur in one or more other automata. We refer to such transitions collectively under the name of *synchronizing* transitions. Synchronizing transitions may also be functional. In any given automaton, transitions that are not synchronizing transitions are said to be *local* transitions. As a general rule, it is shown in [13], that stochastic automata networks may always be treated by separating out the local transitions and the synchronizing events. A system containing N stochastic automata with E synchronizing events may be written as

$$Q = \sum_{j=1}^{2E+N} \otimes_{g,i=1}^{N} Q_j^{(i)}. \tag{1}$$

This formula is referred to as the *descriptor* of the stochastic automata network. The subscript g denotes a generalization of the tensor product concept to matrices with functional entries [14].

Two simple examples [14] will be used to illustrate the performance of the numerical algorithms. We do not wish to claim that these examples cover the entire spectrum of SAN models and often realistic models are more complex; instead they were chosen because they incorporate features that typically lead to numerical difficulties.

A Model of Resource Sharing : In this model, N distinguishable processes share a certain resource, and at most P of them can concurrently use this resource. Each of these processes alternates between a *sleeping* state and a resource *using* state. We shall let $\lambda^{(i)}$ be the rate at which process i awakes from the sleeping state wishing to access the resource, and $\mu^{(i)}$, the rate at which this same process releases the resource when it has possession of it. In our SAN representation, each process is modelled by a two state automaton $\mathcal{A}^{(i)}$. We shall let $s\mathcal{A}^{(i)}$ denote the current state of automaton $\mathcal{A}^{(i)}$. Also, we introduce the function $f = \delta\left(\sum_{i=1}^{N} \delta(s\mathcal{A}^{(i)} = using) < P\right)$, where $\delta(b)$ is an integer function that has the value 1 if the boolean b is true, and the value 0 otherwise. The local transition matrix for automaton $\mathcal{A}^{(i)}$ is

$$Q_l^{(i)} = \begin{pmatrix} -\lambda^{(i)} f & \lambda^{(i)} f \\ \mu^{(i)} & -\mu^{(i)} \end{pmatrix},$$

and the overall descriptor for the model is, according to equation (1)

$$Q = \bigoplus_{g\ i=1}^{N} Q_l^{(i)} = \sum_{i=1}^{N} I_2 \otimes_g \cdots \otimes_g I_2 \otimes_g Q_l^{(i)} \otimes_g I_2 \otimes_g \cdots \otimes_g I_2,$$

where Id_d denotes the identity matrix of order d.

The SAN product state space for this model is of size 2^N. Notice that when $P = 1$, the reachable state space is of size $N + 1$, which is considerably smaller than the product state space, while when $P = N$ the reachable state space is the entire product state space. Other values of P give rise to intermediate cases.

A Queueing Network with Blocking and Priority Service : The second model we shall use is an open queueing network of three finite capacity queues and two customer classes. Class 1 customers arrive from the exterior to queue 1 according to a Poisson process with rate λ_1. Arriving customers are lost if they arrive and find the buffer full. Similarly, class 2 customers arrive from outside the network to queue 2, also according to a Poisson process, but this time at rate λ_2 and they also are lost if the buffer at queue 2 is full. The servers at queues 1 and 2 provide exponential service at rates μ_1 and μ_2 respectively. Customers that have been served at either of these queues try to join queue 3. If queue 3 is full, class 1 customers are blocked (blocking after service) and the server at queue 1 must halt. This server cannot begin to serve another customer until a slot becomes available in the buffer of queue 3 and the blocked customer is transferred. On the other hand, when a (class 2) customer has been served at queue 2 and finds the

buffer at queue 3 full, that customer is lost. Queue 3 provides exponential service at rate μ_{3_1} to class 1 customers and at rate μ_{3_2} to class 2 customers. It is the only queue to serve both classes. In this queue, class 1 customers have preemptive priority over class 2 customers. Customers departing after service at queue 3 leave the network. We shall let $C_k - 1$, $k = 1, 2, 3$ denote the finite buffer capacity at queue k.

Queues 1 and 2 can each be represented by a single automaton ($\mathcal{A}^{(1)}$ and $\mathcal{A}^{(2)}$ respectively) with a one-to-one correspondance between the number of customers in the queue and the state of the associated automaton. Queue 3 requires two automata for its representation; the first, $\mathcal{A}^{(3_1)}$, provides the number of class 1 customers and the second, $\mathcal{A}^{(3_2)}$, the number of class 2 customers present in queue 3.

This SAN has two synchronizing events: the first corresponds to the transfer of a class 1 customer from queue 1 to queue 3 and the second, the transfer of a class 2 customer from queue 2 to queue 3. We shall denote these synchronizing events as s_1 and s_2 respectively. In addition to these synchronizing events, this SAN requires two functions. They are $f = \delta(s\mathcal{A}^{(3_1)} + s\mathcal{A}^{(3_2)} < C_3 - 1)$ and $g = \delta(s\mathcal{A}^{(3_1)} = 0)$. The function f has the value 0 when queue 3 is full and the value 1 otherwise, while the function g has the value 0 when a class 1 customer is present in queue 3, thereby preventing a class 2 customer in this queue from receiving service. It has the value 1 otherwise. The various matrices for this model can be found in [14] and the descriptor follows the general format (eq. (1))

$$Q = \bigoplus_g Q_l^{(i)} + \bigotimes_g Q_{s_1}^{(i)} + \bigotimes_g \bar{Q}_{s_1}^{(i)} + \bigotimes_g Q_{s_2}^{(i)} + \bigotimes_g \bar{Q}_{s_2}^{(i)},$$

The reachable state space of the SAN is of size $C_1 \times C_2 \times C_3(C_3 + 1)/2$ whereas the complete SAN product state space has size $C_1 \times C_2 \times {C_3}^2$.

3. ALGORITHM ANALYSIS

We present without proof, the algorithm [8] concerning vector-descriptor multiplication. We use the notation $B[\mathcal{A}]$ to indicate that the matrix B may contain transitions that are a function of the state of the automaton $\mathcal{A}$, and more generally, $A^{(m)}[\mathcal{A}^{(1)}, \mathcal{A}^{(2)}, \ldots, \mathcal{A}^{(m-1)}]$ to denote that the matrix $A^{(m)}$ may contain elements that are a function of the state variable of one or more of the automata $\mathcal{A}^{(1)}, \mathcal{A}^{(2)}, \ldots, \mathcal{A}^{(m-1)}$. We assume, without loss of generality that the state space of automaton i is $1, \ldots, n_i$. Precisely, we

consider the algorithm to perform the multiplication

$$x\Big(A^{(1)} \otimes_g A^{(2)}[\mathcal{A}^{(1)}] \otimes_g A^{(3)}[\mathcal{A}^{(1)}, \mathcal{A}^{(2)}] \otimes_g \cdots \otimes_g A^{(N)}[\mathcal{A}^{(1)}, \ldots, \mathcal{A}^{(N-1)}]\Big) =$$

$$\begin{aligned} x(& I_{1:N-1} \otimes_g A^{(N)}[\mathcal{A}^{(1)}, \ldots, \mathcal{A}^{(N-1)}] \\ & \times I_{1:N-2} \otimes_g A^{(N-1)}[\mathcal{A}^{(1)}, \ldots, \mathcal{A}^{(N-2)}] \otimes_g I_{N:N} \\ & \times \cdots \times I_{1:1} \otimes_g A^{(2)}[\mathcal{A}^{(1)}] \otimes_g I_{3:N} \times A^{(1)} \otimes_g I_{2:N}) \end{aligned} \tag{2}$$

where $I_{i:j}$, with $i \leq j$ denotes the identity matrix of size $\prod_{k=i}^{j} n_k$. This algorithm, presented in Figure 1, involves at most of the order of $\prod_{i=1}^{N} n_i \times \sum_{i=1}^{N} n_i$ multiplications (the worst case being when the matrices have only nonzero elements) and this number is a bound on the number of function evaluations. This bound is reached if all matrix entries are functional and if the arguments of the functions are all the automata.

The analysis is a straightforward translation of equation (2). Each tensor product term is decomposed into N normal factors [1] which are processed iteratively by the algorithm. It is useful to consider the code as consisting of the three parts illustrated in Figure 1. Part 1 corresponds to the vector-matrix multiplication, when the matrix has constant entries (i.e., after all functions have been evaluated). Part 2 corresponds to fetching the subvectors to be multiplied and the loop management. Note that subvectors that have to be fetched are composed of non contiguous entries in x. Part 3 occurs only in models with functional transition rates and corresponds to the transformation of a vector index into a SAN state, which is subsequently used as an argument for the functions. Then the functions are evaluated. This is performed only if the matrix $A^{(i)}$ in the innermost loop has functional entries. Part 1 constitutes the essential multiplication phase of the algorithm. Complexity results show that taking advantage of the tensor structure may lead to less operations than a regular global operator. Parts 2 and 3 are the overhead due to the tensor representation and the use of functions. One of the *objectives of this*

(1) A normal factor is a term of the type

$$I_{1:i-1} \otimes_g A^{(i)}[\mathcal{A}^{(1)}, \ldots, \mathcal{A}^{(N)}] \otimes_g I_{i:N}$$

2. Initialize: $nleft = n_1 n_2 \cdots n_{N-1}$; $nright = 1$.
2. For $i = N, \ldots, 2, 1$ do
2. • $base = 0$; $jump = n_i \times nright$;
2. • For $k = 1, 2, \ldots, nleft$ do

---

3. o For $j = 1, 2, \ldots, i-1$ do
3. * $k_j = \left(\left[(k-1) / \prod_{l=j+1}^{i-1} n_l \right] mod \left(\prod_{l=j}^{i-1} n_l \right) \right) + 1$
3. o Evaluate $A^{(i)}[\mathcal{A}^{(1)}, \ldots, \mathcal{A}^{(i-1)}]$ for $k_1, \ldots, k_{i-1}$

---

2. o For $j = 1, 2, \ldots, nright$ do
2. * $index = base + j$;
2. * For $l = 1, 2, \ldots, n_i$ do
2. · $z_l = x_{index}$; $index = index + nright$;

---

1. * Multiply: $z' = z \times A^{(i)}[k_1, \ldots, k_{i-1}]$

---

2. * $index = base + j$;
2. * For $l = 1, 2, \ldots, n_i$ do
2. · $x_{index} = z'_l$; $index = index + nright$;
2. o $base = base + jump$;
2. • $nleft = nleft / n_{i-1}$;
2. • $nright = nright \times n_i$;

Figure 1. – Basic Multiplication Algorithm.

paper is to give an empirical evaluation of this overhead, to propose some optimizations, and to compare this method with the usual sparse matrix method (multiplying a vector and a matrix in a Harwell Boeing format).

Remark 1: In equation (2), only one automata can depend on the $(N-1)$ other automata and so on. One automaton must be independent of all the others. This provides a means by which the individual factors on the right-hand side of equation (2) *must* be ranked; i.e., according to the automata on which they *may* depend. A given automaton may actually depend on a subset of the automata in its parameter list.

What is not immediately apparent from the algorithm as described is the fact that in *ordinary tensor products, (OTP),* the normal factors commute and the order in which the innermost multiplication is carried out may be arbitrary. In *generalized tensor products, (GTP),* however this freedom of choice does not exist; the order is prescribed. As indicated by the right hand side of equation 2, each automaton $\mathcal{A}^{(i)}$ is represented by a normal factor which *must* be processed before any factor of its arguments $[\mathcal{A}^{(1)}, \ldots, \mathcal{A}^{(i-1)}]$ is processed. Let us call this order a *legal processing* order. This order depends

on the set of dependencies of the tensor product [2] and always exists if the set of dependencies is acyclic. In the sequel, this order is denoted γ, and γ_i is the index of the automaton in the i-th position after re-ordering. Note that there may be more than one legal processing order for a term.

Remark 2: Now consider an arbitrary permutation σ (σ_i denotes the initial index of the automaton in the i-th position after permutation). In [8], we show how to perform this simple transformation where the position of each automata in a list describing the SAN is changed. Let us call this order *a positioning order*. In this transformation the multiplication $x \ A^{(1)}[\mathcal{A}^{(1)}, \ldots, \mathcal{A}^{(N)}] \otimes_g \cdots \otimes_g A^{(N)}[\mathcal{A}^{(1)}, \ldots, \mathcal{A}^{(N)}]$ becomes

$$x \ P^{\sigma} \ A^{(\sigma_1)}[\mathcal{A}^{(1)}, \ldots, \mathcal{A}^{(N)}] \otimes_g \cdots \otimes_g A^{(\sigma_N)}[\mathcal{A}^{(1)}, \ldots, \mathcal{A}^{(N)}] \ (P^{\sigma})^T. \quad (3)$$

Here P^{σ} is a permutation matrix defined by σ, and $(P^{\sigma})^T$ is its transpose. In what follows, we denote $x^{\sigma} = xP^{\sigma}$. In this context the left-hand side of (2) becomes

$$x^{\sigma}\Big(A^{(\sigma_1)}[\mathcal{A}^{(1)}, \ldots, \mathcal{A}^{(N)}] \otimes_g \cdots \otimes_g A^{(\sigma_N)}[\mathcal{A}^{(1)}, \ldots, \mathcal{A}^{(N)}]\Big).$$

No matter which permutation is chosen, the right-hand side remains essentially identical in the sense that the terms are always ordered with the same legal processing order γ. The only change results from the manner in which each *normal factor* is written. Each must have the form $I_{\sigma_1:\sigma_{i-1}} \otimes_g A^{(\sigma_i)}[\mathcal{A}^{(1)}, \ldots, \mathcal{A}^{(N)}] \otimes_g I_{\sigma_{i+1}:\sigma_N}$, where $I_{\sigma_l:\sigma_k}$ denotes the identity matrix of size $\prod_{i=l}^{k} n_{\sigma_i}$.

Remark 3: Now consider a tensor product term and a legal processing order γ. The normal factor decomposition is

$$A^{(1)}[\mathcal{A}^{(1)}, \ldots, \mathcal{A}^{(N)}] \otimes_g \cdots \otimes_g A^{(N)}[\mathcal{A}^{(1)}, \ldots, \mathcal{A}^{(N)}] =$$
$$\prod_{i=1}^{N} I_{1:\gamma_i-1} \otimes_g A^{(\gamma_i)}[\mathcal{A}^{(1)}, \ldots, \mathcal{A}^{(N)}] \otimes_g I_{\gamma_i+1:N}$$

It is possible now to permute each normal factor of this decomposition. Denote $\epsilon^{(i)}$ the permutation [3] $1, \ldots, N \ \rightarrow \ 1, \ldots, \gamma_i - 1, \gamma_i + 1, \ldots, N, \gamma_i,$

([2]) A set of dependencies can be modeled with a directed graph, where the nodes are the automata and there is an arc from i to j if and only if the state of automaton i is an argument of a functional transition rate in automaton j in the tensor term. We only consider acyclic sets of dependencies here. It is shown in [8] that if there is a cyclic set of dependencies in a term, this term can be exactly decomposed into a number of non-cyclic terms.

[3] The property is true for any permutation, but will be used for an $\epsilon^{(i)}$ as defined next.

which places automaton γ_i in the last position. This leads to the equivalent expression

$$\prod_{i=1}^{N} P^{\epsilon^{(i)}} I_{1:\gamma_i-1} \otimes_g I_{\gamma_i+1:N} \otimes_g A^{(\gamma_i)}[\mathcal{A}^{(1)}, \ldots, \mathcal{A}^{(N)}] \quad (P^{\epsilon^{(i)}})^T. \quad (4)$$

In what follows, we show how to use the expressions (3) and (4) to reduce the number of function evaluations in the algorithm implementation.

4. REDUCING THE NUMBER OF FUNCTION EVALUATIONS

In a descriptor, there are generally terms with and without functions and the function arguments have various patterns. When a term has no function, PEPS uses the algorithm of Figure 1 with part 3 removed. When a term has functions, it incurs an overhead. This cost mainly comes from: (a) the computation of individual automata states (the arguments) from a global state index, (b) the function evaluation itself knowing the individual automata states, (c) the number of these evaluations.

The cost of the function evaluations (b) is clearly model dependant. The cost of the computation of individual automata states from a global state index (a) and the number of times it is done (c) are also model dependant, in the sense that it depends on the set of dependencies. Nevertheless, for a given set of dependencies, the number of these evaluations and their cost may be reduced.

To estimate this overhead, we experimented with three versions of the multiplication algorithm: the first version is straightforward and does not use any positioning order, the second and third versions use permutations as indicated in remarks 2 and 3 respectively in order to reduce the cost of the function evaluations.

Basic algorithm (A): In this algorithm, the positioning order is the automata index order $1, \ldots, N$. A legal processing order is γ (γ_i is the index l of the matrix $A^{(l)}$ that is i^{th} in processing order), and a general algorithm to perform the following multiplication is given in Figure 2.

$$x\Big(A^{(1)}[\mathcal{A}^{(1)}, \ldots, \mathcal{A}^{(N)}] \otimes_g \cdots \otimes_g A^{(N)}[\mathcal{A}^{(1)}, \ldots, \mathcal{A}^{(N)}]\Big) =$$
$$x \prod_{i=1}^{N} I_{1:\gamma_i-1} \otimes_g A^{(\gamma_i)}[\mathcal{A}^{(1)}, \ldots, \mathcal{A}^{(N)}] \otimes_g I_{\gamma_i+1:N}$$

In this algorithm (and the versions that follow), the respective order of the loops on k and j could be exchanged. Without any specific information on

```
2.   For i = γ1,γ2,...,γN do
2.     • nleft = ∏_{l=1}^{γi-1} nl;
2.     • nright = ∏_{l=γi+1}^{N} nl;
2.     • jump = ∏_{l=γi}^{N} nl;
2.     • base = 0
2.     • For k = 1,2,...,nleft do
2.       o For j = 1,2,...,nright do
2.         * index = base + j;
2.         * For l = 1,2,...,nγi do
2.           · zl = x_index;  index = index + nright;
-------------------------------------------------------
3.         * Automata-state (k1,...,kγi-1,kγi+1,...,kN)
3.         * Evaluate A^(γi)[A^(1),...,A^(N)] for k1,...,kγi-1,kγi+1,...,kN
-------------------------------------------------------
1.         * Multiply: z' = z × A^(γi)[k1,...,ki-1,ki,...,kN]
-------------------------------------------------------
2.         * index = base + j;
2.         * For l = 1,2,...,nγi do
2.           · x_index = zl';  index = index + nright;
2.       o base = base + jump;
```

Figure 2. - Basic algorithm (A)

the positioning order of the automata, the function call "Automata-state", which computes the individual state of each automata from an index into the global state vector, must be performed before any multiplication with $A^{(\gamma_i)}[k_1, \ldots, k_{\gamma_i-1}, k_{\gamma_i}, \ldots, k_N]$. This means that the individual states of all automata are computed, even if only a subset is required for the function evaluation. This version has no permutation cost, but may be expensive in terms of function evaluations. We show next that the cost of part 3 can be reduced by choosing an appropriate positioning order. When a normal factor within this term has no function, part 3 is removed for this factor.

Algorithm using two permutations (B): One idea to reduce the number of function evaluations is to use an appropriate positioning order and the expression of the problem given in equation (3). In the algorithm of Figure 2, this means moving Part 3 from the loop in j to the loop in k. For that purpose, we need to choose a positioning order σ such that *the automata parameters are ranked before the functions*. This is always possible as we deal with tensor terms with acyclic dependencies. As a consequence, a trivial legal processing order is the reverse of the positioning order. Note that the processing order does not have any influence on the function evaluation performance issue. The algorithm to perform the following multiplication

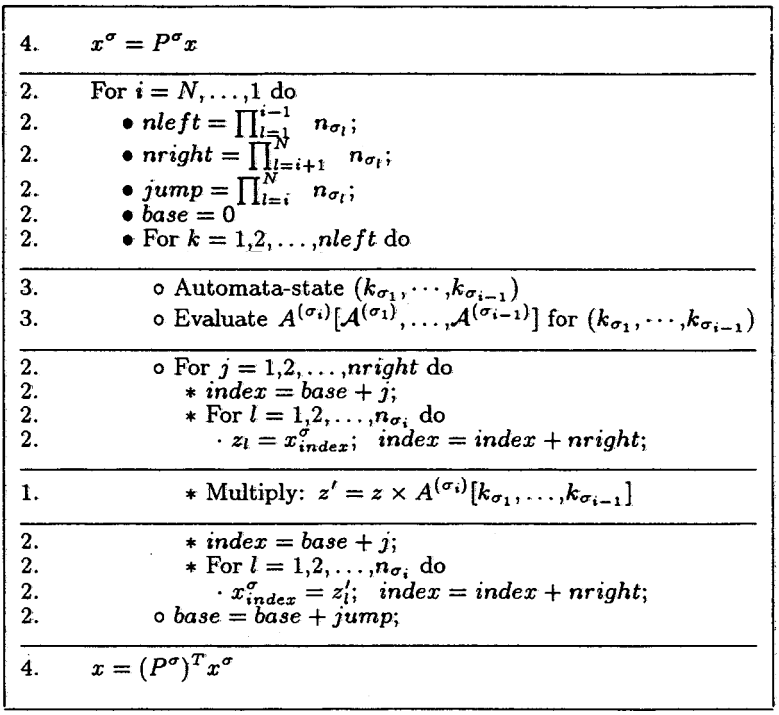

Figure 3. – Algorithm using two permutations (B).

is given in Figure 3.

$$x^\sigma \Big(A^{(\sigma_1)}[\mathcal{A}^{(1)},\ldots,\mathcal{A}^{(N)}] \otimes_g A^{(\sigma_2)}[\mathcal{A}^{(1)},\ldots,\mathcal{A}^{(N)}] \otimes_g \cdots \otimes_g A^{(\sigma_N)}[\mathcal{A}^{(1)},\ldots,\mathcal{A}^{(N)}] \Big)$$

In this version, the function "Automata-state" may be called from the loop in k, and the matrix $A^{(\sigma_i)}[k_1,\ldots,k_{\sigma_{i-1}}]$ is a constant for all the included j-loops. This optimization always saves computation, by reducing (a) and (c): the set of parameters is better identified so the cost of a call to "Automata-state" is smaller, and the number of evaluations usually significantly decreased, which is the main source of performance gain. When a normal factor within this term is constant, part 3 is omitted for this factor.

It is possible to try to reduce further the number of calls to "Automata-state" and to "Evaluate". For a set of functional automata F, this global

number of calls is $\sum_{i \in F} \prod_{l=1}^{i-1} n_{\sigma_l}$ and we now suggest a heuristic which can decrease this number[4].

Defining this order requires some notation: Let F be the set of functional automata, P the set of automata which are parameters of the functions and not functional themselves and C be what remains, i.e., the set of automata with constant entries and which are not parameters of functional automata. Consider an iterative process to determine a positioning order for the automata. If a subset S of automata has been ranked, an automata i in F and not yet ranked is *eligible* if and only if all its parameters are either already ranked or in P. Let S_i be the set of its parameters already ranked, and $T_{S,i}$ the others (thus in P). In this situation, the weight of an eligible automata is defined as $w_{i,S} = \prod_{l \in T_{S,i}} n_{\sigma_l}$. Given two eligible automata i and j in F, i is "smaller" than j if and only if $n_i < n_j \quad or \quad \left(n_i = n_j \quad and \quad w_{i,S} < w_{j,S}\right)$.

Heuristic

* Initially, $S = \emptyset$, and F, P and C are defined as above

* While there are eligible automata in F, choose any "smallest" eligible automata i, and rank first its arguments $T_{S,i}$, in any order, then rank i, i.e., $S \leftarrow S,\ T_{S,i},\ i,\ F \leftarrow F - \{i\},\quad P \leftarrow P - \{T_{S,i}\}$.

* Lastly, rank the automata in C in any order.

The basic idea is that, given a set of integers $n_1, \ldots, n_N$, the non decreasing ranking $n_1 \leq \ldots \leq n_N$ minimizes the expression $\sum_{i=1}^{N} \prod_{l=1}^{i-1} n_l$.

In our context, this order does not respect the constraint "parameters before functions" and the summation is not on $[1...N]$ but on F. Our intuition relies on the adaption of this idea. It is trivially true that the algorithm gives the minimum cost if F consists of a single element, on any set of N automata and for any set of dependencies. Assume now that the algorithm has ranked automata $S = \sigma_1, \ldots, \sigma_l$. This ranking has the partial cost C_S. Adding the next function f to S gives the partial cost $C = C_S + prod_{1:l} \times w_{f,S}$ where $prod_{1:l} = \prod_{i=1}^{l} n_{\sigma_i}$, and the final cost is of the form $C = C_S + prod_{1:l} \times w_{f,S} + prod_{1:l} \times w_{f,S} \times n_f \times C_{additional}$ where $C_{additional}$ will depend on the decisions taken later. A good guess is to make n_f minimum as it is multiplied by a large number, and for equal sizes, $w_{f,S}$ minimum.

[4] We conjecture that the optimization of this function is an NP-hard problem.

This heuristic is used for this version of the algorithm to compute the positioning order (and the processing order) of a tensor term. It is an attempt to reduce the number of function evaluations using only two permutations per tensor term.

Algorithm using $N + 1$ permutations (C): In the preceding case, our concern was in finding a single positioning order which reduces the number and cost of function evaluations. Of course, this does not necessarily find the minimum number and cost of function evaluations for a single normal factor of the tensor term. To perform this optimization, we need, in general, one permutation per normal factor. To do this, we use expression (4).

If we follow the same reasoning as in the previous case, the optimum is reached by a permutation which, for each nornal factor, ranks first the parameters of the functional term, then the functional term, and finally the rest. If we do that, part 2 of the algorithm would be as before, fetching subvectors with non-contiguous components in x. Instead, we use a permutation $\epsilon^{(i)}$ which ranks first the parameters of matrix $A^{(\gamma_i)}$ from rank 1 to rank $p^{(\gamma_i)}$ and γ_i in the last position. Thus, not only do we minimize the number of function evaluations but also, in part 2, contiguous subvectors are fetched. This may decrease the cost of the algorithm in terms of memory management and is a little less expensive in term of computation time. The algorithm is given in Figure 4.

Notice that the loop management is slightly different and the subvectors to be multiplied are composed of contiguous entries in x. The permutation $(P^{\epsilon^{(i)}})^T \; P^{\epsilon^{(i+1)}}$ is performed in one step and the last permutation P^{ϵ} recovers the initial ordering of the automata. If $A^{(\gamma_i)}$ has no function, $\epsilon^{(i)}$ is the identity permutation (and is skipped) and a regular loop management like that of Figure 2 is used.

In this version of the algorithm, the cost of the function evaluations is minimized. The price to pay is the $N + 1$ (at most) permutations per tensor term.

Heuristic for choosing an algorithm: The choice of the multiplication procedure may be different for each tensor term. In some cases, the best method may be chosen a priori, while in some others, experiments may decide if the reduction of the function evaluation cost is worth the price of the permutations. We would like to suggest the following rules:

* If the term is constant, the algorithm of Figure 2 with part 3 omitted, is used.

4. $x^{\epsilon} = P^{\epsilon^{(1)}} x$

2. For $i = \gamma_1, \gamma_2, \ldots, \gamma_N$ do
2. • $nleft = \prod_{l=1}^{p(\gamma_i)} n_{\epsilon_l^{(i)}}$;
2. • $nright = \prod_{l=p(\gamma_i)+1}^{N-1} n_{\epsilon_l^{(i)}}$;
2. • $base = 0$
2. • For $k = 1,2,\ldots,nleft$ do

3. • Automata-state $(k_1, \ldots, k_{p(\gamma_i)})$
3. • Evaluate $A^{(i)}[A^{(1)}, \ldots, A^{(N)}]$ for $k_1, \ldots, k_{p(\gamma_i)}$

2. o For $j = 1,2,\ldots,nright$ do
2. * $index = base + (j-1)n_{\gamma_i}$;
2. * $z = x^{\epsilon}[index, \ldots, index + n_{\gamma_i}]$;

1. * Multiply: $z' = z \times A^{(\gamma_i)}[k_1, \ldots, k_{p(\gamma_i)}]$

2. * $x^{\epsilon}[index, \ldots, index + n_{\gamma_i}] = z'_l$;
2. o $base = base + nright$

4. o if $i \neq \gamma_N$ then $x^{\epsilon} = (P^{\epsilon^{(i)}})^T P^{\epsilon^{(i+1)}} x^{\epsilon}$
4. $x = P^{\epsilon} x^{\epsilon}$

Figure 4. – Algorithm with $N + 1$ permutations (C).

* If the number of function evaluation cannot be reduced using permutations, use algorithm (A). The cases where the number of function evaluations cannot be reduced are for example: if the ranking $1, \ldots, N$ is the best or if there is a single functional normal factor whose parameters are all the other automata. This is the case in the Mutex example where there is only one matrix with functional entries per tensor term and these functions have all the other automata as arguments: the number of function evaluations cannot be reduced, as the functional matrix has to be placed in the last position, anyway.

* If the number of function evaluations is the same using $N+1$ permutations or 2 permutations, then use algorithm (B). This is the case if the positioning order computed in method B is optimal, that is to say such that for all automata with functions (in this term), it is preceded only by its parameters.

* For a tensor product reduced to a single functional normal factor and that we are not in the case of item 2, algorithms (B) and (C) have two permutations, but with different positioning order. Experiments show (as noted before) that (C) is better (although sometimes only slightly).

* If the tensor product has more than one function, version (A), (B) or (C) are viable. The best one will be obtained as a trade-off betwen the permutation cost and the function evaluation cost.

In PEPS, a procedure is available, which determines experimentally, the best procedure to use. The next section shows comparative experiments with these 3 methods. The time to compute this heuristic and the various orderings is negligible compared to the time of one iteration. These steps manipulate small data structures, typically of the size of the number of automata.

4.1. Implementation Details

The various versions of vector multiplication with a generalized tensor product have been implemented in the software package PEPS, version 3.0 [16]. This version of PEPS is written in C++. Experiments have lead us to introduce some basic optimizations in the algorithm skeleton presented above.

Sparsity : Notice that the number of multiplications in the innermost loop of the algorithm may be reduced by taking advantage of the fact that the small block matrices are generally sparse. Previous complexity results were computed under the assumption that the matrices are full. In the implementation presented here, sparsity of individual small matrices is exploited by an appropriate multiplication algorithm (part 1). The same is true for certain highly structured block matrices such as those that are diagonal, contain a single row or column and so on.

Particular normal factors : In the decomposition of a tensor product into normal factors,

$$I_{1:i-1} \otimes_g A^{(i)}[\mathcal{A}^{(1)}, \ldots, \mathcal{A}^{(N)}] \otimes_g I_{i+1:m},$$

there might be factors where $A^{(i)}[\mathcal{A}^{(1)}, \ldots, \mathcal{A}^{(N)}]$ is simply an identity matrix. In this case, the processing of the factor is simply skipped. For example, tensor products which comes from the local transition matrix (see the Mutex example) are reduced to a single normal factor.

Diagonal elements : The descriptor formulation (1) as the summation of $N + 2E$ terms (each term processed using the algorithm above), implies that the diagonal elements of the global transition matrix are obtained as a sum of $N + 2E$ multiplications, when executing the algorithm $N + 2E$ times. It is always better, in terms of computation time, to handle only the nondiagonal elements of the descriptor by this algorithm. The diagonal entries of the descriptor are pre-calculated once and stored in a vector. This

means that the descriptor is now in the form $D + \sum_{j=1}^{N+E} \otimes_{g,i=1}^{N} Q_j^{(i)}$. The vector-matrix multiplication includes a dot product with the diagonal entries. This requires additional memory, which may however be reduced using the techniques introduced in [21].

Automata state evaluation : In order to evaluate the functions, the algorithm must compute the state of each individual automata using a global index in the global state vector. This function, "Automata-state", is a simple base decomposition as shown in [Davio] and can be implemented as such, using integer division and remainder (as in part 3 of Figure 1). A less expensive implementation is obtained, for any positioning order σ, by keeping track (in an array) of the current state of the automata $(\sigma_1, \sigma_2, \ldots, \sigma_N)$. At each loop iteration, when k is incremented by 1, the current state array $(\sigma_1, \sigma_2, \ldots, \sigma_N)$ is changed by the *next* state. The *next* state is defined by the lexical order on automata states, when automata are ranked by σ. The change in the state array amounts to a few (maximum N) additions or settings to 0 and tests. This means that the function "Automata-state" has a relatively cheap iterative implementation, for any positioning order σ, and for all subsets of automata $(\sigma_1, \sigma_2, \ldots, \sigma_i)$.

Permutations : Permutations, when required, are implemented in PEPS as follows: given a copy of the vector x^ϵ according to the automata positioning order ϵ, we want the permuted copy x^σ according to σ. The basic steps of the permutation algorithm are, starting from index 0 in x^ϵ and x^σ, and visiting all entries of x^ϵ in sequence (index i^ϵ), the current state array being $k_{\epsilon_1}, \ldots, k_{\epsilon_N}$:

* for each increment of i^ϵ, compute the "next" state array of $k_{\epsilon_1}, \ldots, k_{\epsilon_N}$. Accordingly, compute the new index i^σ from its previous value, knowing that $i^\sigma = \sum_{j=1}^{N} (\prod_{l=\sigma_{i+1}}^{\sigma_N} n_{\sigma_l}) k_{\sigma_i}$. As the "next" function for $k_{\epsilon_1}, \ldots, k_{\epsilon_N}$ amounts to a few increments or setting to 0, this is indeed also true for computing the "next" i^σ.
* copy $x_{i_\epsilon}^\epsilon$ into $x_{i_\sigma}^\sigma$.

Number of vector copies: We consider now the cost, in terms of number of vector copies, within an iterative scheme of the basic step $\pi_{n+1} = \pi_n D + \pi_n \sum_{j=1}^{N+E} \otimes_{g,i=1}^{N} Q_j^{(i)}$. Permanent copies of π_{n+1} and π_n are always required. In Method B, an additional vector is needed if there is a tensor product term which cannot be reduced to a normal factor. Indeed, for a normal factor, the input can be directly π_n and the result can be directly added into π_{n+1}. In Method C, when a tensor term is not reduced to a normal factor, two additional vectors are needed. One is required as in Method B

to store the temporary results of each multiplication by a normal factor, the other to perform each intermediate permutation.

4.2. Implementation Measurements

All numerical experiments were conducted on an IBM RS6000/370 workstation running AIX version 3.2.5. For each model, 20 iterations of each vector-descriptor multiplications were performed, and the CPU time in seconds for one iteration and for each part is reported here and compared with the sparse method. The sparse method is inspired from the version in MARCA [19]. The models are:

(1) The Mutex example with numerical values $\lambda^{(i)} = 1.0$ and $\mu^{(i)} = 0.4$ for all i. The values of N and P vary and are reported in the column "Models".

(2) The queueing network example with numerical values $\lambda_1 = 1.0$, $\lambda_2 = 1.0$, $\mu_1 = 3.0$, $\mu_2 = 2.0$ and $\mu_{3_1} = \mu_{3_2} = 2.0$. The values of C_1, C_2 and C_3 are respectively 10, 40 and 50.

In order to cover various cases of functional dependencies, two variations are added to the second model. In the first variation, the rate μ_2 becomes a function $f\mu_2 = \frac{\mu_2}{1+s\mathcal{A}^{(3_2)}}$. This means that queue 2 is slowed down if there are many type 2 customers in queue 3. In the second variation, the service rate of queue 2 is the same function, but in a tabulated form $g\mu_2 = \mu_2 \sum_{i=0}^{C_3-1} \frac{1}{1+i}\delta(s\mathcal{A}^{(3_2)} = i)$. This was to introduce a more complex formula to test the capabilities of the methods when handling functions. Table 1 shows the results for the mutex example [5].

First we should compare the performance of PEPS with the sparse method: PEPS has roughly the same performance when the number of resources is equal to the number of processes. The SAN model has no functions and the state space is the product of the local spaces. In other cases, when the SAN has functions, the cost of the function evaluations and of the permutations is the main overhead, the other parts remaining unchanged. Version (A) of the algorithm is the best in this case, because the descriptor is a sum of tensor product which are reduced to normal factors, and because the functions have all automata as parameters. This function evaluation cost cannot be decreased. Note that version (C), which has the same permutation cost as (B), has a lower cost on Part 2 (loop management). Finally notice that PEPS

[5] The value 0.0 in the table indicates that the actual value is smaller than 0.05

TABLE 1
Resource Sharing Model.

Models	Method	total	mult(1)	loops(2)	funcs(3)	perm(4)	diag	sparse
16-16	A	1.6	1.0	0.5	0.0	0.0	0.0	3.9
	B	1.6	1.0	0.5	0.0	0.0	0.0	3.9
	C	1.6	1.0	0.5	0.0	0.0	0.0	3.9
16-15	A	17.2	1.0	0.5	15.7	0.0	0.0	3.9
	B	23.3	1.0	0.5	15.7	6.1	0.0	3.9
	C	23.1	1.0	0.3	15.7	6.1	0.0	3.9
16-08	A	17.2	1.0	0.5	15.7	0.0	0.0	1.0
	B	23.3	1.0	0.5	15.7	6.1	0.0	1.0
	C	23.1	1.0	0.5	15.7	6.1	0.0	1.0
16-01	A	17.2	1.0	0.5	15.7	0.0	0.0	0.0
	B	23.3	1.0	0.5	15.7	6.1	0.0	0.0
	C	23.1	1.0	0.3	15.7	6.1	0.0	0.0

TABLE 2
Queueing Network Model.

Model	Method	total	mult(1)	loops(2)	funcs(3)	perm(4)	diag	sparse
μ	A	64.8	5.1	2.0	57.5	0.0	0.0	15.7
	B	21.1	5.1	2.0	0.1	13.7	0.0	15.7
	C	18.7	5.1	1.9	0.1	11.6	0.0	15.7
$f\mu_2$	A	79.6	5.1	2.0	72.3	0.0	0.0	15.7
	B	22.6	5.1	2.0	1.6	13.7	0.0	15.7
	C	20.0	5.1	1.9	0.2	12.6	0.0	15.7
$g\mu_2$	A	259.4	5.1	2.0	252.1	0.0	0.0	15.7
	B	40.6	5.1	2.0	19.6	13.7	0.0	15.7
	C	20.4	5.1	1.9	0.6	12.6	0.0	15.7

requires exactly the same amount of time even though the reachable state space ranges from large to small.

Table 2 shows the results for the queueing network example. The descriptor is composed of 3 tensor terms without functions, and 3 tensor terms with a single functional factor when μ_2 is constant. When it is replaced by a function, the tensor term of synchronizing event s_2 has two functions. In this example, version A has the worst performance. In the first experiment, methods B and C perform the same number of permutations (but differently) and have the same number of function evaluations. In the second experiment, method C optimizes the number of function evaluations and performs

more permutations, but the difference is not really significant. In the third experiment, the function evaluation cost is larger and C is the best method.

This concludes the first set of experiments in which our goal was to test the benefits of reducing the number of function evaluations. We note from these experiments that the overhead due to functions (parts 3 and 4) of the algorithm is high compared to part 1 and part 2. In general the time taken by parts 1 and 2 is of the same order as the time it takes to perform the sparse algorithm (remember that the complexity result proves that part 1 has favorable complexity compared to a regular matrix vector multiply and that we use a sparse format for the function "Multiply"). One way to further reduce the number of function evaluations is to reduce the number of automata in a model. Note that reducing the SAN to one automata is equivalent to solving the model as a regular sparse matrix.

5. GROUPING OF AUTOMATA

The objective in this section is to show how we can reduce a SAN to an "equivalent" SAN with fewer automata. The equivalence notion is with respect to the underlying Markov chain and is defined below. Our approach is based on simple algebraic transformations of the descriptor and not on the automata network. Consider a SAN containing N stochastic automata $A_1, \ldots, A_N$ of size $n_1, \ldots, n_N$ respectively, E synchronizing events $s_1, \ldots, s_E$, and functional transition rates. Its descriptor may be written as

$$Q = \sum_{j=1}^{N+2E} \bigotimes_{g,i=1}^{N} Q_j^{(i)}.$$

Let $1, \ldots, N$ be partitioned into B groups named $b_1, \ldots, b_B$, and, without loss of generality, assume that $b_1 = [c_1 = 1, \ldots, c_2]$, $b_2 = [c_2 + 1, \ldots, c_3]$, etc, for some increasing sequence of c_i, $c_{B+1} = N$. The descriptor can be rewritten, using the associativity of the generalized tensor product, as

$$Q = \sum_{j=1}^{2E+N} \bigotimes_{g,k=1}^{B} \Big(\bigotimes_{g,j=c_k+1}^{c_{k+1}} Q_j^{(i)} \Big).$$

The matrices $R_j^{(k)} = \bigotimes_{g,j=c_k+1}^{c_{k+1}} Q_j^{(i)}$, for $j \in 1, \ldots, 2E + N$, are, by definition, the transition matrices of a grouped automaton, named G_k of size

$h_k = \prod_{i=c_k+1}^{c_{k+1}} n_i$. Hence the descriptor may be rewritten as

$$Q = \sum_{j=1}^{2E+N} \bigotimes_{g,k=1}^{B} R_j^{(k)}.$$

This formulation is the basis of the grouping process.

First simplification: Removal of synchronizing events and functional terms

Assume that one of the synchronizing event, say s_1, is such that it synchronizes automata within a group, say b_1. Indeed, this synchronized event becomes internal to group b_1 and may be treated as a transition that is local to G_1. In this case, the value of $R_l^{(1)}$ may be changed in order to simplify the formula for the descriptor. Replacing $R_l^{(1)} + R_{s_1}^{(1)} + \bar{R}_{s_1}^{(1)}$ by $R_l^{(1)}$, the descriptor may be rewritten as

$$\bigoplus_{g,k=1}^{B} R_l^{(k)} + \sum_{j=2}^{E} \left(\bigotimes_{g,i=1}^{B} R_{s_j}^{(i)} + \bigotimes_{g,i=1}^{B} \bar{R}_{s_j}^{(i)} \right).$$

The descriptor is thus reduced (two terms having disappeared). This procedure can be applied in all situations where a synchronized event becomes "internal" to a group.

Following this same line of thought, assume that the local transition matrix of G_1 is a tensor sum of matrices that are functions of the states of automata in b_1 itself. Then the functions in $Q_l^{(i)}$ of $R_l^{(1)} = \bigoplus_{g,i=c_1}^{c_2} Q_l^{(i)}$ are evaluated when performing the generalized tensor operator and $R_l^{(1)}$ is a constant matrix. This is true for all situation of this type, for local matrices and synchronized terms. However, if $R_{s_j}^{(1)}$ is the tensor product of matrices that are functions of the states of automata, some of which are in b_1 and some of which are not in b_1, then performing the generalized tensor product $R_{s_j}^{(1)} = \bigotimes_{g,i=c_1}^{c_2} Q_{s_j}^{(i)}$ allows us to partially evaluate the functions for the arguments in b_1. Other arguments cannot be evaluated. These must be evaluated later when performing the computation $\bigotimes_{g,i=1}^{B} R_{s_j}^{(i)}$ and may in fact, result in an increased number of function evaluations. Some numerical effects of this phenomenon are illustrated in the next section.

Second simplification: Reduction of the reachable state space

In the process of grouping, a situation might arise in which a grouped automata G_i has a reachable state space smaller than the product state

space $[1, \ldots, \prod_{i=c_j+1}^{c_{j+1}} n_i]$. This happens after simplifications one and/or two have been performed. For example, functions may evaluate to zero, or synchronizing events may disable certain transitions. In this case, a reachability analysis is performed in order to compute the reachable state space. This analysis is conducted on the matrix $R_l^{(i)} + \sum_{j=1}^{F} R_{s_j}^{(i)} + \bar{R}_{s_j}^{(i)}$ where F is the set of remaining synchronizing events for the SAN $G_1, \ldots, G_B$. In practice, in the SAN methodology, the global reachable state space is known in advance and the reachable state space of a group may be computed with a simple projection.

6. THE NUMERICAL BENEFITS OF GROUPING

Let us now observe the effectiveness of grouping automata on the two examples discussed in Section 2. In particular, we would like to observe the effect on the time required to perform one multiplication (in seconds) of the descriptor by a vector and on the amount of main memory (in Megabytes) needed to store the descriptor itself. Intuitively we would expect the computation time to diminish and the memory requirements to augment as the number of automata is decreased. The experiments in this section quantify these effects. We first consider the resource sharing model with parameter values $N = 12, 16$ and 20 for both $P = 1$ and $P = N - 1$.

The models were grouped in various ways, varying from the original (non-grouped) case in which $B = N$ to a purely sparse matrix approach in which $B = 1$, where B of course, denotes the number of automata that remain after the grouping process. In the examples with a single resource ($P = 1$) we differentiate between two distinct cases; the first when the automata are grouped but the state space in each of the larger automata that result from the grouping is not reduced and the second when the state space of the resulting automata is reduced (by the elimination of non-reachable states from each new block of automata). The latter is indicated by the label ($\Re$) in the tables. The elimination of non-reachable states only affects the states inside a grouped automaton and not all the states of the global model. This reduction is not possible in models with $N - 1$ resources.

The results presented in Table 3 illustrate the substantial gain in CPU time as the number of automata is reduced, and this with relatively little impact on memory requirements. Furthermore, this is seen to be true even when the state space within the grouped automata are not reduced. A more complete set of results for the reduced case is graphically displayed in Figure 5. However, no results are presented for the two cases $N = 20$; $P = 8$, 19; $B = 1$

TABLE 3
Resource Sharing Model.

Models	$P = 1$		$P = 1$ ($\mathfrak{R}$)		$P = N - 1$	
	CPU	Mem	CPU	Mem	CPU	Mem
$N = 12$ $B = 12$	0.8	33	0.8	33	0.8	33
$N = 12$ $B = 6$	0.5	33	0.1	6	0.5	33
$N = 12$ $B = 4$	0.3	33	0.0	2	0.2	34
$N = 12$ $B = 2$	0.2	42	0.0	1	0.1	50
$N = 12$ $B = 1$	0.0	0	0.0	0	0.0	1 087
$N = 16$ $B = 16$	20.8	513	20.8	513	20.8	513
$N = 16$ $B = 8$	10.4	513	1.6	52	12.1	514
$N = 16$ $B = 4$	2.9	516	0.1	5	3.3	518
$N = 16$ $B = 2$	0.7	564	0.0	1	1.1	593
$N = 16$ $B = 1$	0.0	1	0.0	1	0.4	22 527
$N = 20$ $B = 20$	496.7	8193	496.7	8193	496.7	8 193
$N = 20$ $B = 10$	293.2	8194	20.7	462	283.0	8 194
$N = 20$ $B = 5$	67.3	8197	0.6	25	73.8	8 200
$N = 20$ $B = 2$	11.7	8440	0.0	2	19.8	8 640
$N = 20$ $B = 1$	0.0	1	0.0	1	———	———

since the amount of memory needed exceeded that available on our machine. Estimates are presented by a dotted line. When reading these graphs, be aware that the scale varies with increasing values of N.

These graphs display both CPU curves and memory curves. Consider first the memory curves. Two contrasting effects are at work here. First there is the reduction in the reachable state space which entails a subsequent reduction in the size of the probability vectors and hence an overall reduction in the amount of memory needed. On the other hand, the size of the matrices representing the grouped automata is increased thereby increasing the amount of memory needed. This latter effect becomes more important with increasing value of P, and indeed becomes the dominant effect, as may be observed from Figure 5. As for the CPU curves, observe that these always decrease with decreased number of blocks of automata. This is a combined effect of a reduction in the reachable state space, and the number of functions that need to be evaluated. Notice also that the reduction in the number of functions evaluated only occurs when the number of automata in a group is greater than the number of resources. Finally notice that although the gains obtained in models with very few reachable states (those cases in which $P = 1$) are impressive, we must keep in mind the fact that the SAN approach is not

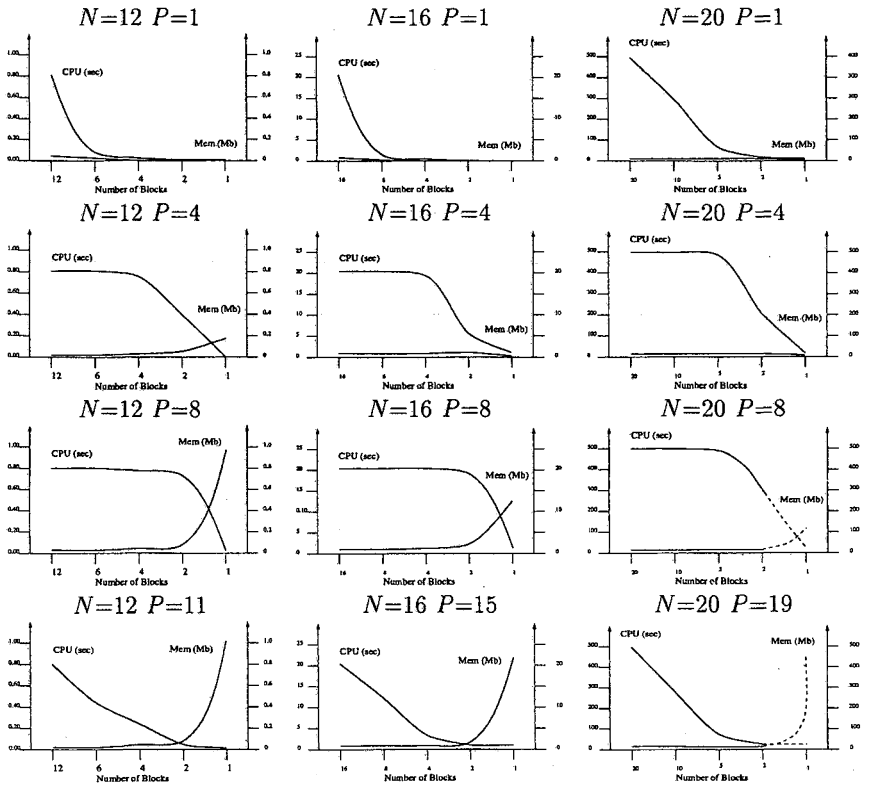

Figure 5. – Reduced Models: Computation Time and Memory requirements.

realistic in these cases. It is much better to generate the small number of states using a standard sparse matrix approach [18, 19].

It may be argued that the best approach (at least for this particular example) is to combine all the automata into just two groups, for this allows us to avoid the state space explosion problem with just a minimal increase in CPU time over a purely sparse approach.

Let us now turn our attention to the queueing network model. We analyzed two models with parameters C_1, $C_2 = 5, 10$ and 20 and $C_3 = 10, 20, 30$ and 50. With these models, experiments were conducted using two different kinds of grouping: a grouping of the automata according to customer class (A_1 and A_{3_1}) and (A_2 and A_{3_2}) and a grouping of the automata according to queue (A_1 and A_2) and (A_{3_1} and A_{3_2}). The second grouping allows for the possibility of a reduction in the state space of the joint automata, (A_{3_1} and A_{3_2}), since the priority queue is now represented by a single automaton.

The results obtained are presented in Table 4 and Figure 6. The last column of Table 4 shows the results for 1 group, thus the sparse method.

TABLE 4
Queueing Network Model.

Models			$(1)(2)(3_1)(3_2)$		$(1,3_1)(2,3_2)$		$(1,2)(3_1,3_2)$		$(1,2)(3_1,3_2)\mathfrak{R}$		$(1,2,3_1,3_2)\mathfrak{R}$	
C_1	C_2	C_3	CPU	Mem	CPU	Mem	CPU	Mem	CPU	Mem	CPU	Mem
5	5	10	0.2	21	0.7	27	0.0	32	0.0	16	0.0	87
5	5	20	0.2	82	0.7	94	0.1	118	0.0	67	0.0	376
10	10	10	0.2	81	0.9	94	0.1	101	0.0	63	0.0	395
10	10	20	0.7	317	3.2	346	0.2	364	0.2	201	0.1	1464
10	10	30	1.6	709	10.2	754	0.6	801	0.5	429	0.4	3 096
10	10	50	4.5	1 963	29.3	2 039	1.6	2 200	1.4	1 153	1.2	8 156
20	20	50	17.4	7 824	88.4	7 989	12.7	8 106	6.6	4 186	———	———

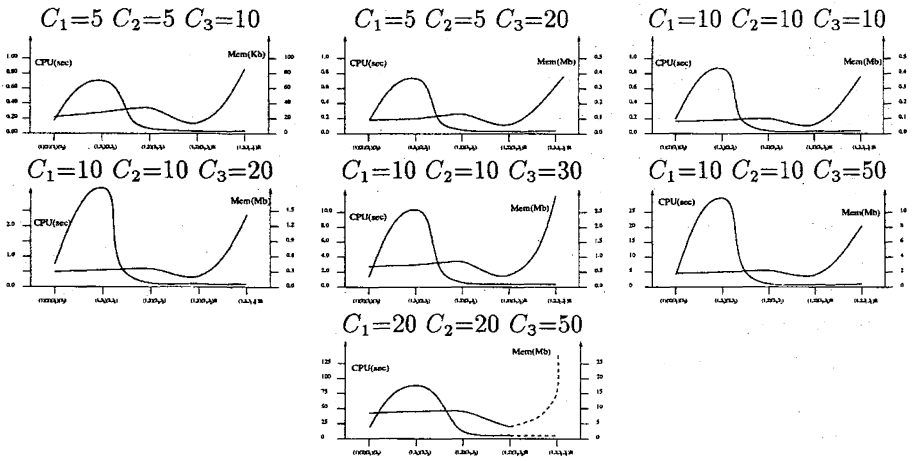

Figure 6. – Reduced Models: Computation Time and Memory requirements.

The gains with this example are not as impressive as they were with the resource sharing model, but it should be remembered that there are relatively few automata (just four) in this example. Notice also that the CPU times obtained with the first grouping are *worse* than in the non-grouped case. This is a result of the fact that this model incorporates synchronizing events combined with functions that cannot be removed using simplification 2. The first grouping eliminates these events, but this results in an increase in

the number of functions that must be evaluated. In all models that possess synchronizing events, a grouping procedure that includes a subset of these events must incorporate the evaluation of tensor products (and not only tensor sums). The evaluation of tensor products can increase the complexity by increasing the number of nonzero elements to be multiplied by the vector. Especially if the tensor products contain functional elements, the number of functions to be evaluated will increase. This effect is apparent in the table of results. Elimination of synchronizing events may prove useful in certain very large problems since descriptors that include synchronizing events require an additional probability vector of size equal to the global number of states.

The CPU time gains obtained from the second grouping result from the elimination of functional elements from the grouped descriptors. All functions depend on the states of automata A_{3_1} and A_{3_2}. Additionally, the elimination of non-reachable states reduces the CPU time and also saves memory. Observe that the curves for memory requirements display an important point of inflexion in the grouping $(1,2)(3_1,3_2)\mathfrak{R}$. This behaviour is due to the reduction in the size of the state space that is possible at this point and at the global grouping $((1,2,3_1,3_2)\mathfrak{R})$. Notice also that the CPU time gains after this point are relatively small. This second phenomena is due to the absence of additional function elimination from the case $(1,2)(3_1,3_2)\mathfrak{R}$ to the case $(1,2,3_1,3_2)\mathfrak{R}$. We were unable to generate the results for the final two entries of this table because of the excessive amount of memory needed.

The experiments clearly show that the benefits that accrue from grouping are non-negligible, so long as the number of function evaluations do not rise drastically as a result. In fact, it seems that function evaluation should be the main concern in choosing which automata to group together. Indirectly, functions also play an important role in identifying non-reachable states, the elimination of which permit important reductions in CPU time and memory. The cost of the grouping itself is model dependant. Basically, for the Mutex example, it corresponds to 10% of the cost of one multiplication of the grouped model ($B = 2$) and for the queueing example to 20%. The major part of this cost is the generation of new functions, if any.

Some rules of thumb for grouping states :

* Memory availability is the primary factor that limits grouping.
Subject to this constraint, the smaller the number of groups the better.
* For any synchronizing event with functional rates or routing probabilities, group functional automata with their arguments.

* If the number of groups can be reduced still further, group other (not in a synchronizing event) functional automata with their arguments.
* If the number of groups can be reduced even more, group automata that are synchronized by the same synchronizing event(s).

7. CONCLUSIONS

To conclude, the implementation of the vector-descriptor multiplication that we suggest should conform to the following steps:
* Group the automata of the SAN according to the rules given above, with a few possible alternatives.
* Test the performance of the vector descriptor multiplication for each alternative and choose the best.
* Use this grouping in an iterative solution procedure.
In the current version of PEPS, the grouping decisions are to be suggested by the user. PEPS computes automatically the descriptor of the grouped SAN, reduces the state space accordingly and decides on which multiplication procedure to use for each term of the descriptor. Before solving the model for many parameters values, the user should test the efficiency of the vector descriptor multiplication for various groupings.

ACKNOWLEDGEMENTS

Paulo Fernandes' research is supported by the (CNRS – INRIA – INPG – UJF) joint project *Apache*, CAPES-COFECUB Agreement (Project 140/93) and PUC/RS, Brazil. Brigitte Plateau's research is supported by the (CNRS – INRIA – INPG – UJF) joint project *Apache*. William J. Stewart's research is supported in part by NSF (DDM-8906248 and CCR-9413309).

REFERENCES

1. K. Atif, Modélisation du Parallélisme et de la Synchronisation. Thèse de Docteur de l'Institut National Polytechnique de Grenoble, 24 September 1992, Grenoble, France.
2. pF. Baccelli, A. Jean-Marie and I. Mitrani, Eds., Quantitative Methods in Parallel Systems, Part I: Stochastic Process Algebras; *Basic Research Series, Springer*, 1995.
3. P. Buchholz, Equivalence Relations for Stochastic Automata Networks. *Computations with Markov Chains; Proceedings of the 2nd International Meeting on the Numerical Solution of Markov Chains*, W.J. Stewart, Ed., Kluwer International Publishers, Boston, 1995.
4. P. Buchholz, Hierarchical Markovian Models – Symmetries and Aggregation; *Modelling Techniques and Tools for Computer Performance Evaluation*, Ed. R. Pooley, J.Hillston, Edinburgh, Scotland, 1992, pp. 234–246.

5. M. DAVIO, Kronecker Products and Shuffle Algebra. *IEEE Trans. Comput, C-30*, No. 2, 1981, pp. 1099–1109.
6. S. DONATELLI, Superposed Stochastic Automata: A Class of Stochastic Petri Nets with Parallel Solution and Distributed State Space. *Performance Evaluation, 18*, 1993, pp. 21–36.
7. P. FERNANDES, B. PLATEAU and W. J. STEWART, Numerical Issues for Stochastic Automata Networks. Proceeding of the Fourth Process Algebras and Performance Modelling Workshop. Edited by Marina Ribaudo, Published by CLUT, Torino, July 1996.
8. P. FERNANDES, B. PLATEAU and W. J. STEWART, Efficient Vector-Descriptor Multiplications in Stochastic Automata Networks. INRIA Report # 2935. Anonymous ftp *ftp ftp.inria.fr/INRIA/Publication/RR.*
9. J.-M. FOURNEAU and F. QUESSETTE, Graphs and Stochastic Automata Networks. *Computations with Markov Chains; Proceedings of the 2nd International Meeting on the Numerical Solution of Markov Chains*, W.J. Stewart, Ed., Kluwer Int. Publishers, Boston, 1995.
10. H. HERMANNS and M. RETTELBACH, Syntax, Semantics, Equivalences, and Axioms for MTIPP. *Proc. of the 2nd Workshop on Process Algebras and Performance Modelling*, U. Herzog, M. Rettelbach, Ed., Arbeitsberichte, Band 27, No. 4, Erlangen, 1994.
11. J. HILLSTON, Computational Markovian Modelling using a Process Algebra. *Computations with Markov Chains; Proceedings of the 2nd International Meeting on the Numerical Solution of Markov Chains*, W.J. Stewart, Ed., Kluwer Int. Publishers, Boston, 1995.
12. P. KEMPER, Closing the Gap between Classical and Tensor Based Iteration Techniques. *Computations with Markov Chains; Proc. of the 2nd International Meeting on the Numerical Solution of Markov Chains*, W.J. Stewart, Ed., Kluwer Int. Publishers, Boston, 1995.
13. B. PLATEAU, On the Stochastic Structure of Parallelism and Synchronization Models for Distributed Algorithms. *Proc. ACM Sigmetrics Conference on Measurement and Modelling of Computer Systems*, Austin, Texas, August 1985.
14. B. PLATEAU and K. ATIF, Stochastic Automata Network for Modelling Parallel Systems. *IEEE Trans. on Software Engineering, 17*, No. 10, 1991, pp. 1093–1108.
15. B. PLATEAU and J. M. FOURNEAU, A Methodology for Solving Markov Models of Parallel Systems. *Journal of Parallel and Distributed Computing. 12*, 1991, pp. 370–387.
16. B. PLATEAU, J. M. FOURNEAU and K. H. LEE, PEPS: A Package for Solving Complex Markov Models of Parallel Systems. In R. Puigjaner, D. Potier, Eds., *Modelling Techniques and Tools for Computer Performance Evaluation*, Spain, September 1988.
17. M. SIEGLE, On Efficient Markov Modelling. *In Proc. QMIPS Workshop on Stochastic Petri Nets*, pp. 213–225, Sophia-Antipolis, France, November 1992.
18. W. J. STEWART, *An Introduction to the Numerical Solution of Markov Chains*, Princeton University Press, New Jersey, 1994.
19. W. J. STEWART, Marca: Markov Chain Analyzer. *IEEE Computer Repository* No. R76 232, 1976. Also IRISA Publication Interne No. 45, Université de Rennes, France.
20. W. J. STEWART, K. ATIF and B. PLATEAU, The Numerical Solution of Stochastic Automata Networks. *European Journal of Operations Research, 86*, No. 3, 1995, pp. 503–525.
21. P. KEMPER, Numerical analysis of superposed GSPNs. *IEEE Trans. on Software Engineering*, Vol. 22(9), Sept. 96.