Available online at www.sciencedirect.com
SCIENCE DIRECT®
Performance Evaluation 58 (2004) 367–390
PERFORMANCE EVALUATION
An International Journal
www.elsevier.com/locate/peva

On the benefits of using functional transitions and Kronecker algebra

Anne Benoit [a,*], Paulo Fernandes [b], Brigitte Plateau [a], William J. Stewart [c]

[a] *IMAG-ID, 51 av. Jean Kuntzmann, 38330 Montbonnot, France*
[b] *Department of Computer Science, PUCRS-PPGCC, Av. Ipiranga, 6681 Porto Alegre, Brazil*
[c] *Department of Computer Science, North Carolina State University, Raleigh, NC 27695-8206, USA*

Received 10 February 2001; received in revised form 8 April 2004
Available online 20 July 2004

Abstract

Much attention has been paid recently to the use of Kronecker or tensor product modelling techniques for evaluating the performance of parallel and distributed systems. While this approach facilitates the description of such systems and mimimizes memory requirements, it has suffered in the past from the fact that computation times have been excessively long. In this paper we propose a suite of modelling strategems and numerical procedures that go a long way to alleviating this drawback. Of particular note are the benefits obtained by using functional transitions that are implemented via a generalized tensor algebra. Examples are presented which illustrate the reduction in computation time as each suggested improvement is deployed.

Keywords: Markov chains; Stochastic automata networks; Generalized tensor algebra; Vector–descriptor multiplication

1. Introduction

Stochastic processes in general, and Markov chains in particular, constitute a modeling paradigm that has broad applicability to many branches of science and engineering. Markov chain models have been proposed as an effective tool for analyzing a great variety of systems. However, their use has not been as widely accepted as one would have hoped or expected. This is an unfortunate situation since the theory and application of Markov chain techniques constitute a unifying theme in the application of mathematics to many problems in biology, engineering, economics, physical science, and social science; the numerical computation of stationary and transient probabilities associated with large scale applications is a fundamental concern. A major reason for this is the well-known state-space explosion problem, whereby even simply stated models generate huge numbers of states. This leads to computational difficulties, both

* Corresponding author. Tel.: +33-476-612089; fax: +33-476-612099.
E-mail addresses: anne.benoit@imag.fr (A. Benoit), billy@csc.ncsu.edu (W.J. Stewart).

0166-5316/$ – see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.peva.2004.04.002

from the point of view of memory requirements (for storing the states, transition matrix and solution/work vectors) and from the point of view of computation time, since large state-space models frequently require large amounts of time to compute transient and/or stationary distributions.

These obstacles were apparent from the very first modeling packages. Indeed, when Wallace and Rosenberg [34] presented their RQA-1 (Recursive Queue Analyzer) software package in 1966, they incorporated a compact storage scheme that took advantage of the special structure of the transition matrices their package could handle. Wallace and Rosenberg identified within the nonzero structure of the matrix, linear patterns (generally parallel to the diagonal) in which the elements were all identical. They then represented all these elements by specifying a nonzero value and the beginning and ending position of the linear structure in the matrix. The numerical solution method used was the power method of Von Mises. There has been much progress since 1966 both in terms of storage requirements and numerical solution techniques. Novel ways of model representation were developed and these in turn instigated further savings in storage and computation time. Chief among these are stochastic Petri nets (SPNs) and stochastic automata networks (SANs). Each of these approaches has benefited from advances in the other.

Stochastic Petri nets, and later generalized SPNs (GSPNs) and superposed GSPNs have a Markov chain as their underlying stochastic structure [1,10,11,16,17,24]. This Markov chain may be stored in any number of ways. Memory efficient implementations favor an implementation that is based either on a Kronecker product formalism or on a structured tree format such as binary decision diagrams (BDDs) [4,5,31], Multi-Terminal BDDs (MTBDDs) [22] and matrix diagrams [14,25]. Using these techniques, the transition matrix can be stored extremely compactly, to such an extent that memory requirements for this matrix is negligible. These efficient means for storing the transition matrix may be accompanied with efficient techniques for storing the state space of the Markov chain [12,13,26]. The state space is required during the matrix generation phase and again after stationary or transient solutions have been found in order to relate computed stationary or transient measures to the initial model itself. The only remaining storage sink is the solution vector itself, plus work vectors that are used in computing solution vectors. Some numerical solution methods (e.g., Gauss–Seidel [35]) need only a single vector (although for slowly converging processes it may be necessary to keep the computed approximation from several tens of iterations prior to the current iterate) and even here some recent work aims to mimimize this in certain types of model, specifically those that can be represented as Kronecker products.

Stochastic Automata Networks (SANs) have been discussed in the literature for over a decade [6,16,19, 21,23,27–29]. It has been observed that they provide a natural means of describing parallel and distributed systems. Each component of the system is represented by a separate, low-order matrix which describes, probabilistically, how this component makes a transition from one possible state to another. Thus, if a component were completely independent, its representing matrix would define a Markov chain whose evolution in time mimics that of the component. Since components generally do not function independently of each other, information must also be provided which details their interaction. For example, the actions of one component might be synchronized with those of another, or again the behavior of one component might depend on (i.e., be a function of) the internal state of a second component.

With this information, it is of course possible to construct a global matrix representing the entire system, to manage this global matrix in a compacted format and to apply sparse matrix technology to obtain both transient and stationary distributions from which all manners of system performance measures may be derived. However, the key element of the SAN approach is that a global matrix is never generated. The

individual component matrices and information concerning component interactions are combined into what is called the SAN *Descriptor*, and is written as a sum of tensor products.

The SAN descriptor approach keeps memory requirements within manageable limits and avoids the state space explosion associated with some state-based approaches. We wish to stress that in order to benefit from this compact form, the descriptor is never expanded into a single large matrix. Consequently, all subsequent operations must necessarily work with the model in its descriptor form and hence numerical operations on the underlying Markov chain transition matrix become more costly. Previously, this cost was sufficiently high to discourage the application of SAN technologies. Recent results obtained by the authors and others [8,9,20] will most likely change this situation. In this paper we show how the application of successive modelling strategems and numerical savoir-faire reduce the time needed to compute stationary distributions by several orders of magnitude, thereby reducing considerably this perceived disadvantage.

In the next section, we shall describe SANs more formally and introduce a number of results and theorems of which we shall have need in this paper. Section 3 presents a suite of approaches for reducing computational costs and memory requirements. In particular, we shall emphasize the essential role that functional transitions play in this scenario, and introduce new forms of the basic shuffle algorithm that works exclusively with vectors that are the size of the reachable state space. We shall also consider some reordering strategies and the effect of grouping large numbers of very small automata into a much fewer number of larger automata. We also briefly describe our efforts in attempting to derive preconditioners for a number of iterative solution methods. Section 4 describes two rather different types of model, a resource sharing model and a queueing network model, on which the various strategies described in Section 3 are tested. In the final section, we discuss the results obtained and exhibit the benefits that can be obtained from the complete suite of time-reducing strategies.

2. Stochastic automata networks

A SAN is a set of automata whose dynamic behavior is governed by a set of *events*. Events are said to be *local* if they provoke a transition in a single automaton, and *synchronizing* if they provoke a transition in more than one automaton. It goes without saying that a single event can generate more than one transition. A transition that results from a synchronizing event is said to be a *synchronized transition*; otherwise it is called a *local transition*. We shall denote the number of states in automaton i by n_i and we shall denote by N the number of automata in the SAN.

The behavior of each automaton, $\mathcal{A}^{(i)}$, for $i = 1, \ldots, N$, is described by a set of square matrices, all of order n_i. In our context, a SAN is studied as a continuous-time Markov chain. The rate at which event transitions occur may be constant or may depend upon the state in which they take place. In this last case they are said to be functional (or state-dependent). Synchronized transitions may be functional or non-functional. Functional transitions allow a system to be modeled as a SAN using fewer automata and fewer synchronizing transitions. In other words, if functional transitions cannot be handled by the modelling techniques used, then a given system may be modeled as a SAN if additional automata are included and these automata are linked to others by means of synchronizing transitions.

In the absence of synchronizing events and functional transitions, the matrices which describe $\mathcal{A}^{(i)}$ reduce to a single infinitesimal generator matrix, $Q^{(i)}$, and the global Markov chain generator may be written as

$$Q = \bigoplus_{i=1}^{N} Q^{(i)} = \sum_{i=1}^{N} I_{n_1} \otimes \cdots \otimes I_{n_{i-1}} \otimes Q^{(i)} \otimes I_{n_{i+1}} \otimes \cdots \otimes I_{n_N}. \tag{1}$$

The tensor sum formulation is a direct result of the independence of the automata, and the formulation as a sum of tensor products, a result of the defining property of tensor sums [15]. The probability distribution at any time t of this independent N-dimensional system is known to be

$$\pi(t) = \bigotimes_{i=1}^{N} \pi^{(i)}(t). \tag{2}$$

Now consider the case of SANs which contain synchronizing events but no functional transitions and let us denote by $Q_l^{(i)}, i = 1, 2, \ldots, N$, the matrix consisting only of the transitions that are local to $\mathcal{A}^{(i)}$. Then, the part of the global infinitesimal generator that consists uniquely of local transitions may be obtained by forming the tensor sum of the matrices $Q_l^{(1)}, Q_l^{(2)}, \ldots, Q_l^{(N)}$. As is shown in [27], stochastic automata networks may always be treated by separating out the local transitions, handling these in the usual fashion by means of a tensor sum and then incorporating the sum of two additional tensor products per synchronizing event. The first of these two additional tensor products may be thought of as representing the actual synchronizing event and its rates, and the second corresponds to an updating of the diagonal elements in the infinitesimal generator to reflect these transitions. Eq. (1) becomes

$$Q = \bigoplus_{i=1}^{N} Q_l^{(i)} + \sum_{e \in \mathcal{E}} \left(\bigotimes_{i=1}^{N} Q_{e^+}^{(i)} + \bigotimes_{i=1}^{N} Q_{e^-}^{(i)} \right). \tag{3}$$

Here $\mathcal{E}$ is the set of synchronizing events. Furthermore, since tensor sums are defined in terms of a matrix sum of tensor products, the infinitesimal generator of a system containing N stochastic automata with E synchronizing events (and no functional transition rates) may be written as

$$Q = \sum_{j=1}^{2E+N} \bigotimes_{i=1}^{N} Q_j^{(i)}. \tag{4}$$

This formula is referred to as the *descriptor* of the stochastic automata network.

Let us momentarily return to Eq. (3) to consider how best to handle the diagonal elements of Q. Since the numerical methods used to compute solutions of SANs are usually iterative, the most important operation is that of multiplying the descriptor with a vector and hence it is essential to keep the cost of this multiplication to a minimum. One way to reduce costs is to precompute the *diagonal of the descriptor*. In this case, the descriptor may be considered as being composed of two parts:

- D, a vector containing the diagonal of the descriptor;
- $\bar{Q}$, the descriptor itself with the exception that all the diagonal elements of the matrices of each tensor product term are set to zero.

From a practical point of view, this is most easily accomplished by setting all the diagonal elements of local matrices $Q_l^{(i)}$ to zero. In each tensor product term corresponding to a synchronizing event e, the diagonal elements of the matrices corresponding to the automaton which "owns" this event must also be set to zero. A second advantage of this astuce now becomes apparent. Normally each synchronizing event generates two tensor product terms. The first term contains the rates of occurrence of the synchronizing

event and the second contains exclusively the elements with which to adjust the diagonal. This second term, once computed, only generates elements on the diagonal of the descriptor. Precomputing the diagonal therefore allows us to eliminate the second tensor product term of each synchronizing event thereby reducing the number of terms requiring manipulation during the multiplication phase. The diagonal elements arising from synchronization terms are added to the diagonal elements corresponding to the tensor sum part of the descriptor and the number of tensor product terms in the descriptor is reduced from $2E + N$ to $E + N$. Precomputing the diagonal brings about even greater savings when the descriptor has functional elements, since the evaluation of functions on the diagonal must all be precomputed, but only once.

Precomputation of the diagonal does have a cost associated with it, although this cost manifests itself only once, namely, during the preparation of the descriptor. On the other hand, the benefits derived from this approach occur each time a vector–descriptor product is computed. A second disadvantage of this approach is the necessity of storing the diagonal elements themselves. Since the representation of the matrix is extremely small, this has the effect of almost doubling the amount of memory needed. However, the augmentation in memory use nevertheless remains low compared to the needs of storing the entire matrix using sparse matrix technology. To counterbalance these inconveniences, this approach provides rapid access to the diagonal of the descriptor with the resulting advantages of

- Easy computation of the largest element of the descriptor, since, given that the descriptor is a representation of the infinitesimal generator, the largest element will always be found along the diagonal.
- Ease of use for implementing certain preconditioning techniques, which, as we shall see later, require access to the diagonal.

Now consider the effect of introducing functional transitions into SANs. It should be apparent that the introduction of functional transition rates has no effect on the *structure* of the global transition rate matrix other than when functions evaluate to zero in which case a degenerate form of the original structure is obtained. In other words, the placement of zero versus nonzero elements essentially remains unchanged. What may change is the value of nonzero elements. The nonzero structure of the SAN descriptor is just as before (except in the case when a function evaluates to zero but even here, the sparse data structures used need not be altered). However, because of possible value changes, the usual tensor operations are no longer valid. Since regular tensor products are unable to handle functional transitions it is necessary to use a *Generalized Tensor Algebra* (GTA) [19], to overcome this difficulty. In particular, this GTA provides some associativity, commutativity, distributivity and compatibility over multiplication properties that enable the descriptor of a SAN with synchronizing events and functional transitions to be handled with algorithms almost identical to those of SANs with no functional transitions. We shall use $B[\mathcal{A}]$ to denote a matrix B, associated with the automaton $\mathcal{B}$, which may contain transitions that are a function of the state of the automaton $\mathcal{A}$; $A^{(m)}[\mathcal{A}^{(1)}, \mathcal{A}^{(2)}, \ldots, \mathcal{A}^{(m-1)}]$ indicates that the matrix $A^{(m)}$ may contain elements that are a function of one or more of the states of the automata $\mathcal{A}^{(1)}, \mathcal{A}^{(2)}, \ldots, \mathcal{A}^{(m-1)}$. The notation $\otimes_{\mathrm{g}}$ denotes a generalized tensor product. Thus $A \otimes_{\mathrm{g}} B[\mathcal{A}]$ denotes the generalized tensor product of the matrix A (having only constant entries) with the functional matrix $B[\mathcal{A}]$.

With this background information, we are now ready to explore a suite of approaches destined to minimize memory requirements and computational burden of applying the SAN modelling concepts.

3. A suite of approaches for reducing computation costs

3.1. Functional transitions via generalized tensor algebra

In real systems, most events do not simply occur of and by themselves. They occur as a result of activities and constraints that, if not global to the system are, at a minimum, dependent on various aspects of it. In other words, they are *functions* of the state of different components in the system. Thus it is natural when building mathematical models of complex systems to include transitions that are functions of the different components. Although this is the most natural manner with which to model functional transitions, it is not the only way.

We can always create a model without functional transitions, but this implies a possibly substantial increase in the number of automata and synchronizing transitions needed. Each different function often requires one additional automaton and a synchronizing transition for each automaton associated with the function.

There is yet another reason why we should work with functional transitions. The incorporation of functional transitions into SANs generally leads to a small number of matrices that are relatively full. On the other hand, avoiding functional transitions using additional automata and synchronizing events leads to many very sparse matrices. Given that the SAN approach is especially effective when the matrices are full (indeed, for a large number of sparse matrices, the SAN approach is less effective than a general sparse matrix [19]), it behooves us to work with functional transitions whenever possible.

However, we saw in Section 2 that to permit the use of functional transitions in SANs, it is necessary to work with a generalized tensor algebra. It now remains to examine the cost of using this generalized tensor approach. When the matrices which represent the automata contain only constant values and are full, the cost of performing the operation basic to all iterative solution methods, that of matrix-vector multiply, or in this case, the product of a vector with the SAN descriptor, is given by

$$\rho_N = \prod_{i=1}^{N} n_i \times \sum_{i=1}^{N} n_i, \tag{5}$$

where n_i is the number of states in the ith automaton and N is the number of automata in the network. When the matrices are sparse, the cost is even less. In [19] it is shown that for sparse matrices, the complexity is of the order of

$$\sum_{i=1}^{N} z_i \prod_{i=1, i\neq j}^{N} n_i = \prod_{i=1}^{N} n_i \sum_{j=1}^{N} \frac{z_j}{n_j}, \tag{6}$$

where z_i denotes the number of nonzero entries in $Q^{(i)}$. To compare this complexity with a global sparse format, the number of nonzero entries of $\bigotimes_{i=1}^{N} Q^{(i)}$ is $\prod_{i=1}^{N} z_i$. It is hard in general to compare the two numbers $\prod_{i=1}^{N} z_i$ and $\prod_{i=1}^{N} n_i \sum_{j=1}^{N} (z_j/n_j)$. Note however that if all $z_i = N^{1/(N-1)} n_i$, both orders of complexity are equal[1]. If all matrices are sparse (e.g., below this bound), the sparse approach is probably better in terms of computation time. This remark is valid for a single tensor product. For a descriptor

[1] The value $N^{1/(N-1)}$ lies between 1 and 2.

which is a sum of tensor products and where functions in the matrices $Q^{(i)}$ may evaluate to zero, it is hard to compute, a priori, the order of complexity of each operation.

The savings are due to the fact that once a partial product is formed, it may be used in several places without having to re-do the multiplication [33]. With functional rates, the elements in the matrices may change according to their context so it is conceivable that this same savings is not always be possible. It was observed in [33] that in the case of two automata $\mathcal{A}$ and $\mathcal{B}$ with matrix representations A and $B[\mathcal{A}]$ respectively, the number of multiplications needed to premultiply $A \otimes B[\mathcal{A}]$ with a vector x remains identical to the nonfunctional case and moreover, exactly the same algorithm could be used. Furthermore, it was also observed that when A contains functional transition rates, but not B, the computation could be rearranged (via permutations) to compute $x(A[\mathcal{B}] \otimes B)$ in the same small number of multiplications as the nonfunctional case. When both contained functional transition rates (i.e., functions of each other) no computational savings appeared to be possible.

3.2. A reduced memory shuffle algorithm

SANs allow Markov chains models to be described in a memory efficient manner because their storage is based on a tensor structure (descriptor). However, the use of independent components connected via synchronizations and functions may produce a representation with many unreachable states. In the following, we denote by PSS the product state space, and by RSS the reachable state space.

Within this tensor (Kronecker) framework, a number of algorithms have been proposed to compute the product of a probability vector and the descriptor. The first and perhaps best-known is the shuffle algorithm [2,18,19], which computes the product but never needs the matrix explicitly. However, this algorithm needs to use "extended" vectors $\hat{\pi}$ with the size of PSS. This algorithm is denoted **E-Sh**, for **extended shuffle**. When there are many unreachable states ($|\text{RSS}| \ll |\text{PSS}|$), **E-Sh** is not efficient, because of its use of extended vectors. The probability vector can therefore have many zero elements, since only states corresponding to reachable states have nonzero probability. Moreover, computations are carried out for all the elements of the vector, even those elements corresponding to unreachable states. Therefore, the computation gain obtained by exploiting the tensor formalism can be ruined if many useless computations are performed, and memory is used for states whose probability is always, by definition of unreachable states, zero.

The use of *reduced* vectors (vectors π which contains entries only for reachable states, i.e., vectors of size |RSS|) allows a reduction in memory needs, and some useless computations are avoided. This leads to significant memory gains when using iterative methods such as Arnoldi or GMRES which can possibly require many probability vectors. A modification to the **E-Sh** shuffle algorithm permits the use of such vectors. However, to obtain good performance at the computation-time level, some intermediate vectors of size PSS are also used. An algorithm described in [2] allows us to save computations by taking into account the fact that the probabilities corresponding to non-reachable states are always zero in the resulting vector. This **partially reduced** computation corresponds to the algorithm called **PR-Sh**. However, the savings in memory turns out to be somewhat insignificant for the shuffle algorithm itself. In most cases, this algorithm performs |RSS| multiplications of a vector slice by a column of a matrix, instead of the |PSS| multiplications required by **E-Sh**. However, sometimes we cannot reduce the number of computations when there are synchronizations in the model. From a memory point of view, there is no real gain even if we use vectors the size of RSS, because the algorithm uses intermediate data structures whose size is PSS.

A final version of the shuffle algorithm concentrates on the amount of memory used, and allows us to handle even more complex models. This **fullyreduced** computation corresponds to the algorithm called **FR − Sh**. This algorithm is described in [3]. In this new algorithm, all intermediate data structures are stored in reduced format. Two arrays of size RSS must be used to keep all information, instead of one array the size of PSS. This algorithm will be more efficient in terms of memory only when less than half of the states are reachable. As far as computation time is concerned, the number of multiplications is reduced to the order of $|\mathrm{RSS}| \times \sum_{j=1}^{N} z_j/n_j$. However, we introduce some supplementary costs, most notably, the cost of a sort which could reach $O(|\mathrm{RSS}|\log(|\mathrm{RSS}|))$.

3.3. Further strategies for reducing vector–descriptor computation costs

We saw in Section 3.1 that generalized tensor product algorithms are available that, in most cases, keep the cost of multiplying a vector with a tensor product to that of the usual nongeneralized case. This is most important in the presence of functional transitions, which would otherwise require considerably more terms. However, it should be borne in mind that a SAN descriptor is a sum of tensor products. In this section we recommend two different procedures which when implemented reduce the computational costs of forming the product of a vector with a complete SAN descriptor. The first concerns the manner in which normal factors should be processed; the second is that of automata grouping.

3.3.1. Re-ordering normal factors

Generally, in a descriptor, there are terms with functions and terms without functions; furthermore the function arguments have various forms. It is obvious that when a term includes functions, it incurs an overhead. This cost is mainly due to

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 actual cost of all three of these is model dependent. However, the number of these evaluations and the cost may be reduced as shown below.

Remark. The operators $\otimes$ and $\oplus$ are not commutative. However, we shall have need of a pseudo-commutativity property that may be formalized as follows. Let σ be a permutation of the set of integers $[1, 2, \ldots, N]$. Then there exists a permutation matrix, P_σ, of order $\prod_{i=1}^{N} n_i$, such that

$$\bigotimes_{i=1}^{N} A^{(i)} = P_\sigma \bigotimes_{i=1}^{N} A^{(\sigma(i))} P_\sigma^T.$$

A proof of this property may be found in [29] wherein P_σ is explicitly given. Intuitively, the P_σ transformation boils down to considering the probability vector entries according to different lexicographic orders. P_σ^T is the transpose (and inverse) of this permutation matrix. In the sequel, we assume that, using a permutation σ, the tensor product is well ordered as given in [19].

Algorithms. We have tested different algorithms based on the algorithm **E-Sh** without functional elements to show how we can use reordering in order to develop improved algorithms. The basic version is an algorithm without normal factor permutations. The function evaluations are performed in an inner

loop (the so-called r loop, see [2,3]). The cost of function evaluations can be reduced by choosing an appropriate order for each normal factor. If we use a permutation, we minimize the number of function evaluations by moving all the function evaluations out of the inner r loop. Evaluation takes place only when a parameter value has changed (in the l loop, in the terminology of [2,3]). This improved algorithm reorders each normal factor, for all terms of the descriptor.

A final algorithm applies the reordering of each normal factor, if and only if necessary. Indeed, the choice of the multiplication procedure may be different for each tensor term. We wish to point out two simple cases in which the number of function evaluations cannot be reduced. The first occurs if the ranking $1, \ldots, N$ is best; the second when there is a single functional normal factor whose parameters are all the other automata. These features are discussed in detail in [2].

3.3.2. Automata grouping

We now examine a computation-saving strategy that is based on the reduction of a SAN to an equivalent SAN having fewer automata. The concept of equivalence is made by means of an algebraic transformation of the infinitesimal generator to which the SAN corresponds. As we have seen, the descriptor of a well defined SAN may be written as

$$Q = \bigoplus_{i=1}^{N}{}_{\mathrm{g}} Q_l^{(i)} + \sum_{e\in\mathcal{E}} \left[\bigotimes_{i=1}^{N}{}_{\mathrm{g}} Q_{e^+}^{(i)} + \bigotimes_{i=1}^{N}{}_{\mathrm{g}} Q_{e^-}^{(i)} \right]. \tag{7}$$

Let us now define B groups called $b_1, \ldots, b_B$ and, without loss of generality, let us suppose that the group b_i is composed of indices $[c_i + 1, \ldots, c_{i+1}]$ with $c_1 = 0$ and $c_{B+1} = N$. Thus $b_1 = [1, \ldots, c_2]$, $b_2 = [c_2 + 1, \ldots, c_3]$, …, and $b_B = [c_B + 1, \ldots, N]$. Letting $d_i = c_i + 1$ and using the associativity of generalized tensor sums and products, the descriptor may be rewritten as

$$Q = \bigoplus_{i=1}^{B}{}_{\mathrm{g}} \left(\bigoplus_{j=d_i}^{c_{i+1}}{}_{\mathrm{g}} Q_l^{(j)} \right) + \sum_{e\in\mathcal{E}} \left[\bigotimes_{i=1}^{B}{}_{\mathrm{g}} \left(\bigotimes_{j=d_i}^{c_{i+1}}{}_{g} Q_{e^+}^{(j)} \right) + \bigotimes_{i=1}^{B}{}_{\mathrm{g}} \left(\bigotimes_{j=d_i}^{c_{i+1}}{}_{g} Q_{e^-}^{(j)} \right) \right].$$

Let us define the following matrices:

$$R_l^{(i)} = \bigoplus_{j=d_i}^{c_{i+1}}{}_{\mathrm{g}} Q_l^{(j)}, \qquad R_{e^+}^{(i)} = \bigotimes_{j=d_i}^{c_{i+1}}{}_{\mathrm{g}} Q_{e^+}^{(j)}, \qquad R_{e^-}^{(i)} = \bigotimes_{j=d_i}^{c_{i+1}}{}_{\mathrm{g}} Q_{e^-}^{(j)}.$$

By definition, these matrices correspond to a *grouped automata*, which we shall call $\mathcal{G}^{(i)}$, of a SAN that is equivalent to the one defined in (7), i.e.

$$Q = \bigoplus_{i=1}^{B}{}_{\mathrm{g}} R_l^{(i)} + \sum_{e\in\mathcal{E}} \left[\bigotimes_{i=1}^{B}{}_{\mathrm{g}} R_{e^+}^{(i)} + \bigotimes_{i=1}^{B}{}_{\mathrm{g}} R_{e^-}^{(i)} \right]. \tag{8}$$

The product state space of each automata $\mathcal{G}^{(i)}$ is $\prod_{j=d_i}^{c_{i+1}} n_j$. This purely algebraic formulation is the basis for a computation reducing strategy obtained by grouping automata. Grouping may be used to achieve a number of objectives including those of

- eliminating synchronizing events;
- eliminating functional transitions;
- reducing the size of the state space.

Grouping can bring about many benefits. The limit of this process is a single automaton containing all the states of the Markov chain. However we do not wish to go to this extreme. Observe that just four automata each of order 100 brings us to the limit of what is currently possible to solve using regular sparse matrix techniques yet memory requirements for four automata of size 100 remain modest. More details may be found in [30].

3.4. Projection methods and preconditioning for sums of tensor product matrices

In searching to minimize computation time, we first examined how the use of functional transitions can keep SAN models relatively simple without increasing computational complexity. Then we turned our attention to two ways of reducing the computational burden of forming vector-descriptor products. The final step in the use of the SAN methodology is the computation of stationary distributions by means of iterative or projection methods. It is to this final aspect that we now direct our attention.

As we have seen, an operation that can be carried out with little difficulty is that of forming the product of the compact descriptor of a SAN and a vector. This means that numerical iterative methods, whose basic operation is a vector-matrix multiply, may be applied with little difficulty. In the modelling community, the best known of these methods is the power method [32]. Unfortunately this is often a very slow method. More sophisticated projection methods also have a matrix vector multiplication as their basic operation and these generally converge much more quickly. However, in order for projection methods to perform to their potential, they need to be complemented with preconditioners, a means whereby the distribution of the eigenvalue spectrum is modified to achieve a better rate of convergence. Our primary purpose in the remainder of this section is to address this issue, to examine what types of preconditioners can be used within the SAN context and to observe which ones work and which ones do not. Some previous results concerning the application of projection methods to compute stationary solutions of SANs have been presented in [33]. Strategies used so far to find such approximations in stochastic automata networks have been based on two principles. The first is that multiplication by A does not require that A be stored explicitly: one can find approximations to A^{-1} in terms of powers of A. The second is that although it is expensive to invert the sum of the A_i's, *it is relatively easy to invert each of the component matrices* A_i.

The first approach was previously tested and proved to be unsatisfactory [33]. For example, it is possible to use polynomial preconditioning, or to obtain the Neuman series expansion of the inverse of A. In [33], it was shown that the cost of such preconditionings was prohibitive. This was again found to be the case in [18]. Other results can be found in [7] but they do not address functional transition rates. Consequently, we will not address these techniques further. More information, including much experimentation may be found in the references. Instead, we will move on to alternative approaches.

3.4.1. Incomplete factorizations

The most common, and general-purpose, approach to preconditioning a sparse linear system is to obtain the incomplete factorization of A:

$$A = LU + R$$

in which L is a sparse unit lower triangular matrix, U is a sparse upper triangular matrix and R, the residual matrix, is supposedly small, i.e., close to zero. The reader is referred to [32] for details. For example, the ILU(0) factorization obtains factors L and U which have the same structure as the lower and upper parts of A respectively. For this to be possible, the matrix A must be available explicitly. A SAN descriptor can

not directly use these options because of the manner in which the matrix (descriptor) is stored, not in a standard sparse format, but as a sum of tensor products. Nevertheless we point out that a tensor product $A = \bigotimes_{i=1}^{N} A^{(i)}$ may still be able to benefit from such incomplete LU factorizations. Indeed, the property of compatibility of tensor product with traditional matrix multiplication and the property of associativity implies that

$$A = \bigotimes_{i=1}^{N} A^{(i)} = \bigotimes_{i=1}^{N} [L^{(i)} U^{(i)}] = \bigotimes_{i=1}^{N} [L^{(i)}] \bigotimes_{i=1}^{N} [U^{(i)}], \tag{9}$$

where $L^{(i)}$ and $U^{(i)}$ are the LU factors of the sequence $\{A^{(i)}\}$. It should be noticed that the tensor product of two upper (lower) triangular matrices is also an upper (lower) triangular matrix. This allows us to factor a tensor product $A = \bigotimes_{i=1}^{N} A^{(i)}$ by factoring the $A^{(i)}$. Furthermore, an algorithm similar to the one used to multiply a vector by a tensor product may be used to obtain the product of an arbitrary vector v by the inverse of a triangular tensor term ($\bigotimes_{i=1}^{N} [L^{(i)}]$ or $\bigotimes_{i=1}^{N} [U^{(i)}]$).

3.4.2. ILU preconditioning for functional tensor terms

As shown by Eq. (9), each tensor product term of a descriptor can be decomposed into matrices L and U in a tensor format. We shall work with the descriptor, Eq. (4), before the diagonal elements are removed, since removing diagonal terms may aversely affect the LU decomposition of local matrices. In addition to this, two problems may hinder and indeed prevent us from obtaining the LU decomposition of tensor terms and their subsequent use in preconditioning. This results from the nature of the individual matrices to be decomposed. They may contain functional elements and they may be singular.

3.4.2.1. Matrices containing functional elements. The first restriction to the individual decomposition of each tensor term is the existence of functional elements in local matrices. Given that we only require an approximation, it is possible to use heuristics to eliminate such elements. In our experiments five possibilities were tested:

- replace functional elements with zero;
- replace functional elements by one of their values, chosen at random;
- replace functional elements by their largest value;
- replace functional elements by their smallest value;
- replace functional elements by their average value.

Our experiments showed that all these options are pretty much the same, with a slight advantage for the choice of the average value. However, these experiments showed that to obtain satisfactory preconditioning, it is necessary to reduce, if not completely eliminate functional elements by means of automata grouping.

3.4.2.2. Matrices that are singular. It is possible that during the application of the decomposition algorithm, pivot elements become zero, even when a full pivoting strategy is employed. This is guaranteed to happen when the matrix is singular. A number of approaches were used to try to mitigate this situation. One possibility is to incorporate a *regularization* of the matrices before their decomposition. These regularization techniques transform the local matrices, while retaining some of their properties. We experimented with two such forms of regularization,

- a translation which consists of adding a small value to all of the diagonal elements of the matrix and
- a transformation which consists of replacing each of the diagonal elements by the largest (in modulus) of all diagonal elements,

as well as with no regularization at all. In this latter case, only the unit upper triangular matrix, L, available at the moment at which the decomposition breaks down, was used (the corresponding upper triangular matrix U is ignored).

Having described how the experiments handle incomplete factorizations we now move on to describe how they are incorporated into preconditioning strategies. We shall use the following notation:

- $\mathcal{Q}$ the descriptor obtained with the elimination of all of the functional elements of the descriptor Q and replaced by their average values.
- $\mathcal{M}^{(i)}$ the matrix that regroups the ith term of the sum of the descriptor $\mathcal{Q}$ ($\mathcal{Q} \equiv \sum_{i=1}^{(N+2E)} \mathcal{M}^{(i)}$).
- $\mathcal{L}^{(i)}$ ($\mathcal{U}^{(i)}$) the lower (upper) triangular matrix that results from the LU decomposition of the matrix $\mathcal{M}^{(i)} (\equiv [\mathcal{L}^{(i)}\mathcal{U}^{(i)}])$

Note that the inverse of each $\mathcal{M}^{(i)}$ may be easily obtained by means of Eq. (9). In attempting to form incomplete LU factorizations we experimented with combinations of the inverses of the individual tensor terms. We tried three different possibilities:

- the sum of the inverses of all the tensor terms $\mathcal{M}^{(i)}$;
- the product of the inverses of all the tensor terms $\mathcal{M}^{(i)}$;
- the inverse of a single tensor term $\mathcal{M}^{(i)}$.

These approaches, which may seem naive, are actually derived from the *Additive Schwarz* and *Multiplicative Schwartz* methods. Just as in these methods, the principle is to use the inverse of each term as a component of the problem. In this way, the use of combinations of components permits us to develop modular preconditioning. Intuitively, this type of preconditioning will be most beneficial when applied to a model in which there is little interaction among the components. In the option which uses the sum of the inverses of each matrix $\mathcal{M}^{(i)}$ as a preconditioner, we need to multiply the descriptor by

$$\sum_{i=1}^{N+2E} [\mathcal{M}^{(i)}]^{-1}.$$

We call this *additive preconditioning*. For example, in applying this preconditioning to the power method, the iteration becomes

$$x^{(k+1)} = x^{(k)} Q \sum_{i=1}^{N+2E} [\mathcal{M}^{(i)}]^{-1}.$$

Notice that the execution of this algorithm requires an additional vector in which to store intermediate values of the multiplication by each of the $(N + 2E)$ terms.

In the second option, we use the product of the inverses of each matrix $\mathcal{M}^{(i)}$ as the preconditioning matrix. In the power method, the corresponding iteration becomes

$$x^{(k+1)} = x^{(k)} Q \prod_{i=1}^{(N+2E)} [\mathcal{L}^{(i)}\mathcal{U}^{(i)}]^{-1}.$$

We call this *multiplicative preconditioning*. Unlike the previous preconditioning, multiplicative preconditioning does not require an additional vector.

The third and final preconditioning option is the least expensive, because a single term $\mathcal{M}^{(i)}$ is chosen and its inverse used as the preconditioning matrix:

$$[\mathcal{M}^{(i)}]^{-1} \quad \text{with } i \in [1 \ldots (N + 2E)].$$

Furthermore, it is possible to use each of the terms $\mathcal{M}^{(i)}$ in an alternating manner so that all are used at one point or another during the execution of the algorithm. We refer to this as *alternating preconditioning*.

Unfortunately, the results we obtained with *ILU* decompositions of the tensor terms did not turn out to be very successful. Although we did get convergence in several cases, there was only one case in which this was achieved with a better computation time than the non-preconditioned case. In fact, in the power method, almost none of these preconditioning approaches worked. It turned out that among all three types of preconditioning, the alternating type worked best. It has the advantage of being simple to handle and only minimally increases the computation time of an iteration.

3.4.3. Diagonal preconditioning

Having failed to develop good preconditioners based on either the inverse written in terms of its powers or on incomplete factorizations, we turned our attention to the simplest of all preconditioning strategies: the use a diagonal matrix D, whose diagonal elements are simply those of the diagonal of Q. The system to be solved then becomes

$$xQD^{-1} = 0.$$

In the case of a matrix stored as a descriptor, this type of preconditioning is extremely easy to use. Indeed, the diagonal of the descriptor is already calculated and even stored in a vector format. Furthermore, the computation of the inverse of a diagonal matrix is elementary. Although we found that diagonal preconditioning performed better than the other preconditioning techniques, in the final analysis, all that can be said is that diagonal preconditioning is sometimes marginally better than no preconditioning. Much research remains to be carried out in this area.

4. Examples

4.1. A model of resource sharing

We shall use a well-known "Resource Sharing" model as a test example. In the model, N distinguishable processes share a certain resource. Each of these processes alternates between a *sleeping* state and a resource *using* state. However, the number of processes that may concurrently use the resource is limited to P where $1 \leq P \leq N$ so that when a process wishing to move from the sleeping state to the resource using state finds P processes already using the resource, that process fails to access the resource and remains in the sleeping state. Notice that when $P = 1$ this model reduces to the usual mutual exclusion problem. When $P = N$ all of the processes are independent. 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.

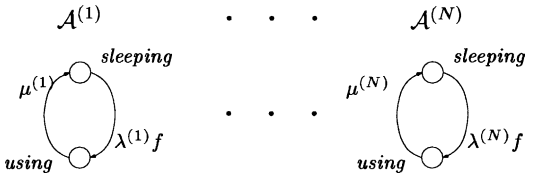

Fig. 1. Resource sharing model – Mutex1.

4.1.1. Model with functions

In our SAN representation with functions, each process is modeled by a two state automaton $\mathcal{A}^{(i)}$, the two states being *sleeping* and *using*. We shall let $s\mathcal{A}^{(i)}$ denote the current state of automaton $\mathcal{A}^{(i)}$. Also, we introduce the function

$$f = \delta\left(\sum_{i=1}^{N} \delta(s\mathcal{A}^{(i)} = \text{using}) < P\right),$$

where $\delta(b)$ is an integer function that has the value 1 if the boolean b is true, and the value 0 otherwise. Thus the function f has the value 1 when access is permitted to the resource and has the value 0 otherwise. This model, which we shall call *Mutex1*, is graphically illustrated in Fig. 1.

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.

4.1.2. Model without functions

Let us now look at how this same system may be modeled without using functional transitions. One possibility is to introduce an additional automaton, a resource pool automaton, which counts the number of units of resource available at any moment. The action of a process in acquiring a resource could then be represented as a synchronizing event requiring the cooperation of the demanding process and the resource pool. A further synchronizing event would be needed for a process to return resource to the resource pool. Fig. 2 illustrates the *Resource Sharing* system modeled using an additional "resource pool" automaton and synchronizing events. To differentiate it from the previous implementation, we shall call this model *Mutex2*.

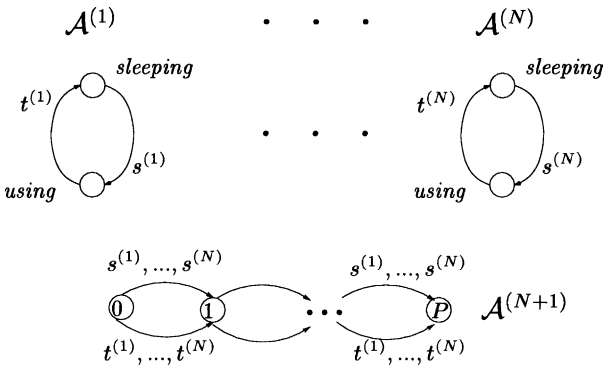

Fig. 2. Resource sharing model without functions – Mutex2.

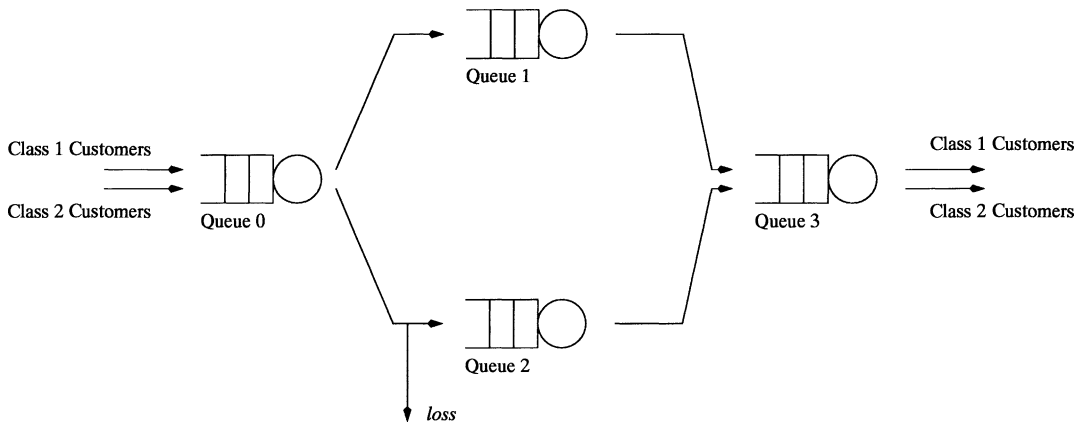

Fig. 3. Queueing network example with four queues and two classes of customer – QN model.

In this model, each process i wishing to acquire a resource must synchronize its action (by means of the synchronizing event s_i) with that of the resource pool automaton. Likewise, when finished with a unit of the resource, its return to the resource pool is governed by the synchronizing event t_i. The SAN product state space for this model is of size $2^N * (P + 1)$, and the reachable state space size is identical to that of *Mutex1*.

4.2. A queueing network model

The second example refers to an open queueing network model with finite capacity queues (with blocking and loss patterns), different classes of customers, priority and complex load-dependent service rates[2]. We shall use a four-queue system with two classes of customer as indicated in Fig. 3.

We assume that both classes of customer enter the first service center (queue 0) with rates λ_1 for class 1 and λ_2 for class 2. Let the capacity of queue 0 be K_0 for customers of both classes. Once served at this center, customers of classes 1 and 2 go, respectively, to service centers 1 and 2. If queue 1 is full (its capacity is denoted by K_1), customers of class 1 will be blocked in service center 0. On the other hand, class 2 customers are lost if queue 2 is full. The capacity of queue 2 is denoted by K_2.

In service center 0, customers of class 1 are served with a variable rate that is inversely proportional to the number of class 1 customers present in service center 3. Similarly, in this same service center, class 2 customers are served with a variable rate that is inversely proportional to the number of class 2 customers in service center 2. There is no priority between customers of classes 1 and 2 in this first service center.

In service center 1, which serves class 1 customers only, the service rate is given by μ_{11}. After service here, customers enter the final service center (queue 3) if there is an available slot in the queue. Otherwise the customer is blocked. In service center 2, which serves class 2 customers only, the service rate is given by μ_{22} and again, exiting customers attempt to enter the final service center, but may be blocked.

[2] Usually the term “load-dependent service rate” refers to a service rate that depends only on the number of customers in the current queue. We use the term “complex load-dependent service rate” to indicate a different kind of dependency where the service rate depends on the number of customers in other queues.

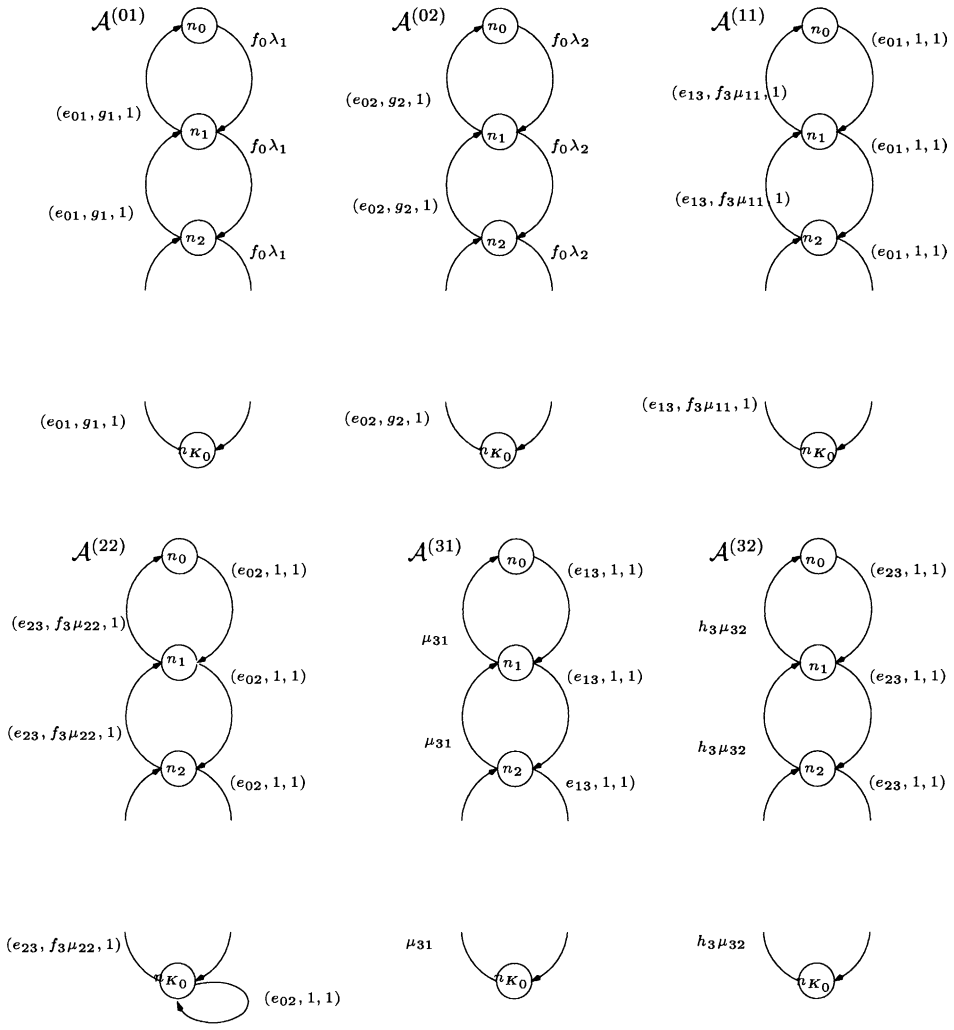

Fig. 4. Queueuing network model – QN1.

Service center 3 provides service to both classes of customer giving priority to class 1 over class 2. Class 2 customers are served only if there are no customers of class 1 in the queue. The service rates in the service center are given by μ_{31} and μ_{32} respectively and its capacity is denoted by K_3.

4.2.1. Model with functions

A SAN model equivalent to the queueing network model just presented may be defined with six automata and functional transition rates. Two automata are needed to describe each of the service centers visited by both classes of customer and one automaton to describe each of the service centers visited by only one class of customer. This SAN model is represented graphically in Fig. 4 and we shall refer to it as QN1.

Arrivals to and departures from the system are represented by local events since they affect only one automaton. The routing of customers between service centers occasions synchronized events since the

state of two automata are altered simultaneously. We denote such events by e_{ij} representing the departure of a customer from service center i to service center j. The event e_{01} represents the departure of a class 1 customer from service center 0 to service center 1[3]. The departure of class 2 customers from service center 0 (loss behavior) is also represented by a synchronized event even though it only changes the state of the automaton representing class 2 customers in service center 0 ($\mathcal{A}^{(02)}$). However since this only happens when queue 2 is full (automaton $\mathcal{A}^{(22)}$ is in its last state) a synchronized event synchronizes the transition representing the departure of a customer from queue 0 (an arc from state i to state $i-1$ in automaton $\mathcal{A}^{(02)}$) with the "circular" transition of the last state of automaton $\mathcal{A}^{(22)}$.

Functional rates are used to represent:

- the capacity restriction of queues represented by automata 0 and 3;
- the dependent service rates in service center 0.

The function f_0 represents the capacity restriction in queue 0. It is evaluated as true (1) if there is room for another customer in queue 0, i.e., if the number of class 1 plus class 2 customers is less than the capacity of the queue. Hence both rates λ_1 and λ_2 must be multiplied by f_0 where

$$f_0 = (\mathrm{st}\mathcal{A}^{(01)} + \mathrm{st}\mathcal{A}^{(02)}) < K_0.$$

Analogously, function f_3 represents the capacity restriction in queue 3, and the transition rates (μ_{11} and μ_{22}) of the synchronized events e_{13} and e_{23} in automata $\mathcal{A}^{(11)}$ and $\mathcal{A}^{(22)}$ respectively must be multiplied by the function

$$f_3 = (\mathrm{st}\mathcal{A}^{(31)} + \mathrm{st}\mathcal{A}^{(32)}) < K_3.$$

The dependent service rates in service center 0 are represented by two functions called respectively g_1 and g_2. The function g_1 is inversely proportional to the number of class 1 customers in service center 1 (the state of automaton $\mathcal{A}^{(31)}$), i.e.,

$$g_1 = \frac{\mu_{01}}{(1 + \mathrm{st}\mathcal{A}^{(31)})}.$$

The service rate of class 2 customers is analogously represented by

$$g_2 = \frac{\mu_{02}}{(1 + \mathrm{st}\mathcal{A}^{(32)})}.$$

The last function, h_3, represents the priority of class 1 over class 2 customers in service center 3. This function must be multiplied by the rate μ_{32}:

$$h_3 = \mathrm{st}\mathcal{A}^{(31)} = 0.$$

The product state space of this model is given by

$$\mathrm{PSS} = (K_0 + 1)^2 \times (K_1 + 1) \times (K_2 + 1) \times (K_3 + 1)^2.$$

However, only some of these states are reachable since obviously, the sum of the states of automata representing the same service center cannot be greater than the capacity of the service center.

[3] Notice that it is unnecessary to specify the class of the customer in the name of the event.

4.2.2. Model without functions

Unlike the previous Mutex example, the equivalent functionless SAN model has the same number and size of automata and will therefore have the same product state space. In fact, to remove the functions of the previous model (QN1) we must introduce a large number of synchronized events that mostly represent possible (non-zero) evaluations of each function.

The function f_0 is replaced with several synchronized events to handle the two automata that represent the first service station. These events allow the arrival of a customer only if queue 0 is not full. It is necessary to include two synchronized events (one for each customer class) for each state in which an arrival is possible (states 0 through $K_0 - 1$).

The function f_3 already appears in synchronized events (events e_{13} and e_{23}) in QN1. These events synchronize the automaton from which a customer departs and the automaton to which the customer arrives. To remove the function, it is necessary to include into these synchronizations, a third automaton which is used to verify if queue 3 has an available slot into which the incoming customer can be placed. This can be achieved using a similar technique to that employed to remove function f_0. It is necessary to replace each of the synchronized events e_{13} and e_{23} by as many synchronized events as the number of states of the automata for which an arrival is possible (i.e., $K_3 - 1$).

The functions g_1 and g_2 also appear in synchronized events in QN1. Each event synchronizes only two automata ($\mathcal{A}^{(01)}$ and $\mathcal{A}^{(11)}$ for event e_{01} and $\mathcal{A}^{(02)}$ and $\mathcal{A}^{(22)}$ for event e_{02}). However these functions refer to the automata describing the state of service center 3. In order to remove these functions it is necessary to extend the synchronized events e_{01} and e_{02} to include the corresponding automaton ($\mathcal{A}^{(31)}$ for e_{01} and $\mathcal{A}^{(32)}$ for e_{02}). Not only is the complexity of these synchronized events increased, but it is also necessary to split these events into different events with different rates according to the number of clients in service center 3.

Function h_3 is the most easily eliminated. It transforms local events consisting of the departure of class 2 customers in service center 3 (transitions from state i to state $i - 1$ state of $\mathcal{A}^{(32)}$) into events that are always synchronized with the first state of automaton $\mathcal{A}^{(31)}$.

5. Numerical results

The numerical experiments where conducted in a IBM PC-like environment with an Intel Xeon processor, 1700 MHz clock speed, and 1.5 Gb memory running the software PEPS [29] on Linux OS (Mandrake 8.0)[4].

The memory use is taken from the system during execution. It represents the totality of memory used by PEPS during its execution (i.e., during the solution of a model). This includes the data, memory structures reserved by the procedure, and also the process stack. The only parameters that change from one algorithm to the next are the memory structures reserved by the algorithms (probability vectors, intermediate array structures, and so on).

[4] The examples used in this paper are available in the PEPS format with the software package PEPS at http://www-id.imag.fr/Logiciels/peps.

5.1. Mutex models

In the Mutex models, we used the following parameter values: $N = 20$ and $P = 16$ with λ_1 through $\lambda_8 = 6$, λ_9 through $\lambda_{15} = 5$, λ_{16} through $\lambda_{20} = 3$, and, for all customers, $\mu = 5$. This model has 1,047,225 reachable states.

The table below displays the results obtained. The results obtained with the base model (Mutex 2; the model with no functional transitions) are shown in the first row. The solution method used is the standard power method. Each successive row adds an additional feature to this base model. The second row incorporates functional transitions (Mutex1). The third row includes the concept of re-ordering automata (which is meaningless for this example, since all automata are identical). The fourth row groups the automata in two groups (each group having 10 automata). The fifth row uses the GMRES method with 10 vectors in the Krylov subspace. The last row indicates the use of diagonal preconditioning and fails to converge in this example.

Where appropriate, results are given in both PSS and RSS implementations. Notice however, that for this example, it is inappropriate to continue the series of optimizations in the RSS context. Observe that in the row "ordering" the solution in the PSS context is better both from the execution time and memory use points of view. This is as should be expected. For this example, once the concept of functional transitions is introduced, the number of states in the product space becomes close to the number of reachable states. This fact, together with the greater complexity of the algorithms in the RSS context, dictate that the solution in the PSS is preferable.

Technique	Common information		Solution in PSS		Solution in RSS	
	PSS (st.)	Convergence (it.)	Time per it. (s)	Mem. use (Mb)	Time per it. (s)	Mem. use (Mb)
No Func.	17825792	111	53.63	564.1	44.27	88.9
Functions	1048576	111	20.32	35.4	33.85	72.2
Ordering	1048576	111	20.32	35.4	33.85	72.2
Grouping	1048576	111	4.55	35.9	–	–
Project.	1048576	39	7.06	109.7	–	–
Precond.	–	–	–	–	–	–

5.2. Queueing network models

In the queueing network models, the parameter values used are $K_0 = K_1 = K_2 = K_3 = 9$ (10 states to each automaton), $\lambda_1 = 3$, $\lambda_2 = 2$, $\mu_{01} = 9$, $\mu_{02} = 8$, $\mu_{11} = 7$, $\mu_{22} = 5$, $\mu_{31} = 6$, and $\mu_{32} = 4$. This model has 302,500 reachable states.

The layout of the table of results is similar to that of the first model. The first row in the table concerns the base model, without functions (QN2); no reordering of automata, the use of the power method, and so on. The second row shows the results obtained when functional transitions are incorporated (QN1). The third row shows the effects of re-ordering automata. The fourth row presents the results obtained when the automata are combined into three groups, the first containing the automata that represent service center 1, the second contains the automata that represent service centers 1 and 2, and the third

contains the automata representing service center 3. The fifth row shows the results obtained with the Arnoldi method using 10 vectors in the Krylov subspace and the final row indicates the use of diagonal preconditioning.

Observe that in these results, the effect of the grouping strategy is to reduce the number of states in the PSS to exactly the reachable state space. Thus, it is inappropriate to continue the suite of experiments in the RSS context.

Technique	Common information		Solution in PSS		Solution in RSS	
	PSS (st.)	Convergence (it.)	Time per it. (s)	Mem. use (Mb)	Time per it. (s)	Mem. use (Mb)
No Func.	1000000	3286	4.61	33.9	11.06	26.3
Functions	1000000	3286	4.61	33.8	4.68	26.2
Ordering	1000000	3286	1.40	33.8	3.58	26.2
Grouping	302500	3286	0.05	12.1	–	–
Project.	302500	488	0.06	33.1	–	–
Precond.	302500	315	0.07	33.1	–	–

5.3. Analysis of the results

A number of important observations may be made concerning the results just presented. Notice first that the use of functions may result in a decrease in the size of the product state space and the number of synchronizing events in a model. This decrease results in memory savings in the PSS context (obviously, the probability vectors are smaller), and also in the RSS context because functions reduce the size of the descriptor. This may be observed in the Mutex example.

To analyze the impact of functions on execution time, the analysis is a little more complex: As a rough guideline, working in the RSS context is beneficial when the ratio between the RSS and PSS is greater than 50%. The use of functions can, in some cases. Reduce the size of the PSS (the RSS remains the same), as can be seen in the Mutex example and it should be noted that a decrease in the size of the PSS can diminish the benefits of working in the RSS context. The table below provides additional data when the ratio RSS over PSS is below 50% for models with and without functions (line 1, $P = 4$); when the ratio is below 50% for the model without function, and above 50% for the model with functions (line 2, $P = 16$); and when the ratio is above 50% for the model with and without function (line 3, *queue*). (Note that the fourth configuration is impossible.) In the case $P = 4$, the solution in RSS with function is significantly better, and the solutions in RSS are globally better. In the case $P = 16$, the solution in PSS with function is the best, it is better than the solution in RSS without function. In the case *queue*, the solution in PSS with function is significantly better, and the solutions in PSS are globally better. The benefits obtained from the use of functions is clearly shown in this table: functions simplify the descriptor, leading to less terms in the summation and to local matrices that are less sparse. Indeed, Kronecker algorithms are most efficient on models that are “not too sparse” and the use of functions helps

significantly in this direction.

Model	Sol. in PSS functions		Sol. in PSS no func.		Sol. in RSS functions		Sol. in RSS no func.	
	Time	Mem	Time	Mem	Time	Mem	Time	Mem
$P = 4$	19.68	35.4	15.51	167.9	2.68	3	4.88	4.5
$P = 16$	20.32	35.4	53.63	564.1	33.85	72.2	44.27	88.9
Queue	1.4	33.8	4.6	33.9	3.58	26.2	11.06	26.3

In the queuing network example, the use of functions does not lead to a diminished PSS, and the simplification of the descriptor seems to be offset by the cost of function evaluations. Although the use of functions seems to be beneficial in terms of memory use and in many cases, in terms of execution time too, we noticed that the cost of function evaluations can be rather high. The module which implements function evaluation in our software package PEPS can probably be optimized, thus reducing this model-dependent additional cost. One possibility is to reduce the cost of a function evaluation using a more clever implementation; another is to reduce the number of function evaluations. We investigated two directions to reduce this number, one related to the shuffle algorithm itself (re-ordering normal factors) and the second related to the model representation (automata grouping). It can be observed that re-ordering does not bring any benefit to the Mutex example, as every functional rate in this model depends on the global state. Nevertheless, re-ordering brings significant gain for the queuing network example, for obtaining the solution in the PSS. Grouping of automata is very beneficial for both models. Unfortunately, this is not always the case, for we have encountered models where grouping adds complexity to the function set and leads to worse performance.

The last two optimization attempts do not concern functions, but they show how far one can go. In both examples, projection methods bring an important reduction in the number of iterations with only a small increase in iteration time. Preconditioning does not lead to convergence for the Mutex example, and brings little benefit to the queuing network example.

It can be observed that, for the Mutex example, in going from 111 iterations of 53 s each to 39 iterations of 7 s each we gain a factor of 21, and for the queuing network example in going from 3286 iterations of 4.61 s each to 315 iterations of 0.07 s each, we gain a factor 687. This illustrates the fact that the generalized shuffle algebra is an important concept, but that its efficient use requires technical optimizations and hence should be used in conjunction with enhancement techniques such as those found in the Markov chain literature: techniques such as the use of projection methods. It also argues for the need of further research in the area of preconditioning techniques that are effective and well adopted to SANs.

Acknowledgements

Anne Benoit's research was supported by CNRS-INRIA-INPG-UJF joint project Apache, and CNPq/INRIA Agreement (project PAGE). Paulo Fernandes' research was supported by CNPq/INRIA Agreement (project PAGE), and FAPERGS (project PEPS). Brigitte Plateau's research was supported in part by NSF (ACI-0203971). William Stewart's research was supported in part by NSF grant ACI-0203971.

References

[1] M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, G. Franceschinis, Modelling with Generalized Stochastic Petri Nets, Wiley, New York, 1995.
[2] A. Benoit, B. Plateau, W.J. Stewart, Memory efficient iterative methods for stochastic automata networks, IRISA Research Report Number 4259, INRIA Rhone-Alpes, 38330 Montbonnot-St-Martin, France, September 2001.
[3] A. Benoit, B. Plateau, W.J. Stewart, Memory-efficient Kronecker algorithms with applications to the modelling of parallel systems, in: Proceedings of the International Workshop on Performance Modelling, Evaluation, and Optimization of Parallel and Distributed Systems (PMEO-PDS'03) at the International Parallel and Distributed Processing Symposium (IPDPS'03), IEEE Computer Society Press, Nice, France, April 2003.
[4] R.E. Bryant, Graph-based algorithms for boolean function manipulation, IEEE Trans. Comp. 35 (8) (1986) 677–691.
[5] R.E. Bryant, Symbolic boolean manipulation with ordered binary-decision diagrams, ACM Comp. Surv. 24 (3) (1992) 318–393.
[6] P. Buchholz, Equivalence relations for stochastic automata networks, computations with Markov chains; in: W.J. Stewart (Ed.), Proceedings of the Second International Meeting on the Numerical Solution of Markov Chains, Kluwer, Boston, 1995.
[7] P. Buchholz, Projection methods for the analysis of stochastic automata networks, in: B. Plateau, W.J. Stewart, M. Silva (Eds.), Numerical Solution of Markov Chains, Prensas Universitarias de Zaragoza, Zaragoza, Spain, September 1999, pp. 149–168.
[8] P. Buchholz, G. Ciardo, S. Donatelli, P. Kemper, Complexity of memory-efficient Kronecker operations with applications to the solution of Markov models, INFORMS J. Comp. 12 (3) (2000) 203–222.
[9] P. Buchholz, M. Fischer, P. Kemper, Distributed steady state analysis using Kronecker algebra, in: B. Plateau, W.J. Stewart, M. Silva (Eds.), Numerical Solution of Markov Chains, Prensas Universitarias de Zaragoza, Zaragoza, Spain, September 1999, pp. 76–95.
[10] G. Chiola, GreatSPN 1.5 Software Architecture, in: Computer Performance Evaluation, Elsevier, Amsterdam, 1992, pp. 121–136.
[11] G. Ciardo, Discrete-time Markovian stochastic Petri nets, in: W.J. Stewart (Ed.), Computations with Markov Chains, Kluwer, Boston, MA, 1995, pp. 339–358.
[12] G. Ciardo, J. Gluckman, D. Nicol, Distributed state-space generation of discrete-state stochastic models, INFORMS J. Comp. 10 (1) (1998) 82–93.
[13] G. Ciardo, A.S. Miner, Storage alternatives for large structured state spaces, in: R. Marie, B. Plateau, M. Calzarossa, G. Rubino (Eds.), Proceedings of the Ninth International Conference on Modelling Techniques and Tools for Computer Performance Evaluation, Lecture Notes in Computer Science, Vol. 1245, St. Malo, France, June 1997, Springer, Berlin, pp. 44–57.
[14] G. Ciardo, A.S. Miner, A data structure for the efficient Kronecker solution of GSPNs, in: P. Buchholz (Ed.), Proceedings of the Eighth International Workshop on Petri Nets and Performance Models (PNPM'99), Zaragoza, Spain, September 1999, IEEE Comp. Soc. Press, 1999, pp. 22–31.
[15] M. Davio, Kronecker products and shuffle algebra, IEEE Trans. Comp. C-30 (1981) 116–125.
[16] S. Donatelli, Superposed stochastic automata: a class of stochastic Petri nets with parallel solution and distributed state space, Perf. Eval. 18 (1993) 21–26.
[17] S. Donatelli, Superposed generalized stochastic Petri nets: definition and efficient solution, in: R. Valette (Ed.), Proceedings of the 15th International Conference on Applications and Theory of Petri Nets, Lecture Notes in Computer Science, Vol. 815, Zaragoza, Spain, June 1994, Springer, Berlin, 1994, pp. 258–277.
[18] P. Fernandes, Méthodes numériques pour la solution de systèmes Markoviens à grand espace d'états, Ph.D. Thesis, University of Grenoble, 1998.
[19] P. Fernandes, B. Plateau, W.J. Stewart, Efficient descriptor-vector multiplication in stochastic automata networks, J. ACM 45 (3) (1998) 381–414.
[20] P. Fernandes, B. Plateau, W.J. Stewart, Optimizing tensor product forms in stochastic automata networks, Oper. Res. 32 (1998) 325–351.
[21] J.-M. Fourneau, F. Quessette, Graphs and stochastic automata networks, computations with Markov chains, in: W.J. Stewart (Ed.), Proceedings of the Second International Meeting on the Numerical Solution of Markov Chains, Kluwer, Boston, 1995.

[22] H. Hermanns, J. Meyer-Kayser, M. Siegle, Multi terminal binary decision diagrams to represent and analyse continuous time Markov chains, in: B. Plateau, W.J. Stewart, M. Silva (Eds.), Numerical Solution of Markov Chains, Prensas Universitarias de Zaragoza, Zaragoza, Spain, June 1997, pp. 188–207.

[23] P. Kemper, Closing the gap between classical and tensor based iteration techniques, computations with Markov chains, in: W.J. Stewart (Ed.), Proceedings of the Second International Meeting on the Numerical Solution of Markov Chains, Kluwer, Boston, 1995.

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

[25] A.S. Miner, Efficient solution of GSPNs using canonical matrix diagrams, in: Proceedings of the Ninth International Workshop on Petri Nets and Performance Models, Aachen, Germany, September 2001, in press.

[26] A.S. Miner, G. Ciardo, Efficient reachability set generation and storage using decision diagrams, in: H. Kleijn, S. Donatelli (Eds.), Proceedings of the 20th International Conference on Applications and Theory of Petri Nets, Williamsburg, VA, USA, Lecture Notes in Computer Science, Vol. 1639, Springer, Berlin, June 1999, pp. 6–25.

[27] B. Plateau, On the stochastic structure of parallelism and synchronisation models for distributed algorithms, in: Proceedings of the 1985 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, Austin, TX, USA, May 1985, pp. 147–153.

[28] B. Plateau, K. Atif, Stochastic automata network for modeling parallel systems, IEEE Trans. Softw. Eng. 17 (10) (1991) 1093–1108.

[29] B. Plateau, J.-M. Fourneau, K.H. Lee, PEPS: a package for solving complex Markov models of parallel systems, in: R. Puigjaner (Ed.), Proceedings of the Fourth International Conference Modelling Techniques and Tools, 1988, pp. 341–360.

[30] B. Plateau, W.J. Stewart, Stochastic automata networks, in: W. Grassmann (Ed.), Advances in Computational Probability; Kluwer Academic Publishers, Boston, 1999 (Chapter 4).

[31] M. Siegle, BDD extensions for stochastic transition systems, in: Proceedings of the UKPEW, Ilkley, UK, July 1997.

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

[33] W.J. Stewart, K. Atif, B. Plateau, The numerical solution of stochastic automata networks, Eur. J. Oper. Res. 86 (1995) 503–525.

[34] V.L. Wallace, R.S. Rosenberg, RQA-1, the recursive queue analyzer, Technical report, University of Michigan, Ann Arbor, 1966.

[35] E. Uysal, T. Dayar, Iterative methods based on splittings for stochastic automata networks, Eur. J. Oper. Res. 110 (1) (1998) 166–186.

Anne Benoit is an ENSIMAG engineer (applied mathematics and computer science), she got her Ph.D. in 2003 at the Polytechnical Institute of Grenoble (INPG). She is now a Research Associate in the Institute for Computing Systems Architecture at the University of Edinburgh. Her research interests include the performance evaluation of systems with a large state space, using stochastic automata networks or process algebra, but also high-level parallel programming and Grid applications.

Paulo Fernandes is Professor of Computer Science at PUCRS, Catholic University of Porto Alegre, Brazil. He got his Dr. degree in 1998 at INPG, Institut National Polytechnique de Grenoble, France, in the numerical solution of large Markovian models. His research interests include performance evaluation of computer and communication systems, as well as theory and numerical solution of stochastic modeling formalisms. His current research topics are structured formalisms, in particular stochastic automata networks and stochastic Petri nets.

Brigitte Plateau received a Master's degree in applied mathematics from the University of Paris 6 in 1976, a *Thèse de Troisième Cycle* in Computer Science from the University of Paris 11 in 1980, and a *Thèse d'Etat* in Computer Science from the University of Paris 11 in 1984. She was *Chargé de Recherche* at CNRS (France) from 1981 to 1985, and Assistant Professor of Computer Science at the University of Maryland from 1985 to 1987. She currently holds a position of Professor at the engineering school ENSIMAG in Grenoble (France) and heads a research group whose main interest is cluster and grid computing. Her research interests include modelling and performance evaluation of parallel and distributed computer systems, and the numerical solution and simulation of large Markov models.

William J. Stewart is Professor of Computer Science at North Carolina State University. His research interests include the theory and numerical solution of Markov chains, the performance evaluation of computer and communication systems and numerical linear algebra. Dr. Stewart initiated a series of international meetings on the numerical solution of Markov chains and has written extensively on the subject.