Split: a flexible and efficient algorithm to vector-descriptor product[*]

Ricardo M. Czekster
PUCRS
Av. Ipiranga, 6681
Porto Alegre - Brazil
90619-900
rmelo@inf.pucrs.br

Paulo Fernandes
PUCRS–CNPq
Av. Ipiranga, 6681
Porto Alegre - Brazil
90619-900
paulo.fernandes@pucrs.br

Jean-Marc Vincent
Laboratoire LIG
Project MESCAL
51, Av. Jean Kuntzmann
38330 Montbonnot, France
Jean-Marc.Vincent@imag.fr

Thais Webber[†]
PUCRS
Av. Ipiranga, 6681
Porto Alegre - Brazil
90619-900
twebber@inf.pucrs.br

ABSTRACT

Many Markovian stochastic structured modeling formalisms like Petri nets, automata networks and process algebra represent the infinitesimal generator of the underlying Markov chain as a descriptor instead of a traditional sparse matrix. A descriptor is a compact and structured storage based on a sum of tensor (Kronecker) products of small matrices that can be handled by many algorithms allowing affordable stationary and transient solutions even for very large Markovian models. One of the most efficient algorithms used to compute iterative solutions of descriptors is the Shuffle algorithm which is used to perform the multiplication by a probability vector. In this paper we propose an alternative algorithm called Split, since it offers a flexible solution between the pure sparse matrix approach and the Shuffle algorithm using a hybrid solution. The Split algorithm puts the Shuffle approach in perspective by presenting a faster execution time for many cases and at least the same efficiency for the worst cases. The Split algorithm is applied to solve two SAN models based on real problems showing the practical contribution of this paper.

Keywords

Performance Evaluation, Numerical Methods, Tensor Algebra, Kronecker Products

[*]This work is partially supported by the brazilian government (CAPES), FINEP project STGSD grant nb. 4284/05 and SMS Project (France). Authors Ricardo M. Czekster and Thais Webber are invited by the Project MESCAL-INRIA.

[†]Corresponding author. The order of authors is merely alphabetical.

1. INTRODUCTION

Modeling large complex systems always requires a structured description. The state space explosion itself exists only because we can describe a very large system, *i.e.*, a system with a huge state space, by describing a structure simple enough to be understood by us, humans. Therefore, it is admissible to rely on a structured description to all problems sufficiently large to be called that way. This is evident observing a stochastic Petri net (SPN) [1] or a stochastic automata network (SAN) [24], and even when you look deeply how large straightforward Markov chains are defined, *e.g.*, the balls and buckets concept in MARCA [25]. Using a less procedural approach, process algebras [19] or graph grammars [14] also use a structured, and very modular, description.

One interesting option to keep the structured characteristics of a continuous-time Markovian model into its internal representation is to employ tensor (Kronecker) algebra to store the infinitesimal generator of a given model. The basic principle that recommends the use of tensor representation for infinitesimal generators is to take advantage of all the structural information already used in the original description of the model. Such principle appears since the first definitions of stochastic automata networks [24], but recently it has also been used in other stochastic formalisms [12, 19]. In all those references, the term descriptor is used to refer to a tensor represented infinitesimal generator. The reader interested in detailed information about tensor algebra can found basic concepts of Classical and Generalized Tensor Algebra in [2, 11, 16].

The use of tensor representations for the infinitesimal generator, however, is not always helpful. It undeniably reduces the memory requirements [22, 5], but it often increases the CPU time needed to achieve stationary or transient solution. One of the major problems with structured representations is the insertion of unreachable states, but to cope with that, very efficient approaches deal with the determination of reachable set [10, 23]. It remains an open problem the efficient solution of large and complex models where all, or almost all, states are reachable.

The numerical algorithms known to compute the exact solution of large reachable state space, non-product form, Kronecker represented models are usually iterative and they are based on the vector-descriptor product. This operation can be done by using clever data structures, *e.g.*, matrix diagrams [21], or using a sum of tensor product of standard matrix structures [16].

Despite the algorithmic differences, both approaches can be summarized in finding an efficient way to multiply a (usually huge) vector by a non-trivial structure. Old stochastic Petri net solutions [1] translate the model representation into a singular sparse matrix. Obviously, this sparse approach cannot be employed to deal with really large models (*e.g.*more than 500 thousand states), since it usually requires the storage of a too large sparse matrix (*e.g.*more than 4 million nonzero elements). The usual stochastic automata network (SAN) solution, called *Shuffle algorithm*, deals with permutations of matrices and tensors [16] and it can be applied to virtually any structured model. Unfortunately, the Shuffle algorithm suffers with a high CPU cost for many practical cases.

The challenging problem for the numerical solution of structured huge Markov models is to speedup the basic operation of most interative solutions, the *vector-descriptor product*. This operation corresponds to the product of a probability vector υ, as big as the product state space, by a descriptor Q. The structure of Q is the ordinary sum of $N + 2E$ tensor products of N matrices, where N is the number of automata, and E is the number of synchronizing events in the model. Therefore, by a simple distributive property, vector-descriptor product algorithms can be viewed in a simpler format as a sum of products of the a vector υ by a tensor product term composed by N matrices:

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

The reader interested in further details about the descriptor structure can find extensive material in [15].

There are many possible approaches to cope with such problems, varying from hybrid solutions between simulation and numerical analysis [7] to alternative storage structures [23, 8]. The focus on this paper proposition is on pure numerical manipulation bringing together the advantages of a straightforward sparse matrix approach [26] and the Shuffle algorithm [16]. A similar solution, called Slice algorithm, was proposed in [17] where a nearly sparse approach was applied. One of the main contributions of the Split algorithm to this domain is the introduction of a flexible approach, since it allows a tailored solution to each tensor product term that may be more time efficient and less memory demanding than any other approaches. In fact, the extreme options of the Split algorithm are at least as time efficient as the sparse approach, or as memory-efficient as the Shuffle algorithm.

This paper is organized as follows. Section 2 present the pure sparse algorithm which is theoretically the more time-efficient approach. Section 3 presents the Shuffle algorithm which is the more memory-efficient approach. Section 4 presents the intermediate Slice algorithm in order to introduce the central contribution of this paper. Section 5 presents the core of this paper which is the Split algorithm and few variants. Section 6 presents some numerical results comparing the application of the Split algorithm compared to the other options to some families of SAN models. Finally, the conclusion emphasizes this paper contributions and draws possible future works to evolve the Split algorithm.

2. TIME-EFFICIENT ALGORITHM

The more intuitive method to solve the problem, called *Sparse* algorithm, aims to consider the tensor product term as a unique sparse matrix, multiplying it by a product space sized probability vector. Considering a tensor product term of N matrices $Q^{(i)}$, each one of order n_i, and with nz_i nonzero elements, the Sparse algorithm generates element by element one large matrix Q resulting of $\otimes_{i=1}^{N} Q^{(i)}$. Then the corresponding elements of vector υ, also with size $\prod_{i=1}^{N} n_i$, are multiplied using a traditional algorithm to the vector-sparse matrix product.

Defining $\theta(1 \dots N)$ as the set of all possible combinations of nonzero elements of the matrices from $Q^{(1)}$ to $Q^{(N)}$, the cardinality of $\theta(1 \dots N)$, and consequently the number of nonzero elements in Q, is given by: $\prod_{i=1}^{N} nz_i$. Additionally, the Sparse algorithm needs the information of the size of the state space corresponding to all matrices after the k-th matrix of the tensor product, called $nright_k$, and numerically defined by $\prod_{i=k+1}^{N} n_i$. Further in this paper, it will be also used the analogous concept of $nleft_k$ which is numerically equal to $\prod_{i=1}^{k-1} n_i$.

Algorithm 1 Sparse Algorithm - $\Upsilon = \upsilon \times \otimes_{i=1}^{N} Q^{(i)}$

```
 1: $\Upsilon = 0$
 2: for all $i_1, \dots, i_N, j_1, \dots, j_N \in \theta(1 \dots N)$ do
 3:     $e = 1$
 4:     $base_{in} = base_{out} = 0$
 5:     for all $k = 1, 2, \dots, N$ do
 6:         $e = e \times q^{(k)}_{(i_k, j_k)}$
 7:         $base_{in} = base_{in} + (i_k \times nright_k)$
 8:         $base_{out} = base_{out} + (j_k \times nright_k)$
 9:     end for
10:     $\Upsilon[base_{out}] = \Upsilon[base_{out}] + \upsilon[base_{in}] \times e$
11: end for
```

One possible implementation of the sparse algorithm is presented above. The number of floating point multiplications for this implementation is:

$$N \times \prod_{i=1}^{N} nz_i \tag{2}$$

However, in this version, all nonzero elements of Q are generated during the algorithm execution. Such elements generation represents $(N-1) \times \prod_{i=1}^{N} nz_i$ multiplications that could be avoided by generating one (usually huge) sparse matrix to store these $\prod_{i=1}^{N} nz_i$ nonzero elements. It would eliminate rows 3 and 6 from the Sparse algorithm and reduce the number of floating point multiplications to just:

$$\prod_{i=1}^{N} nz_i \tag{3}$$

This option allows the Sparse algorithm to be very time-efficient, but potentially very memory demanding due to the storage of a, potentially huge, sparse matrix Q.

Another interesting approach to the sparse algorithm is to keep the nonzero elements inside the algorithm, but factorizing previous calculations [8]. Note that all combinations of elements of each matrix of the tensor product have multiplications in common, *i.e.*, the combinations can be generated considering their partial results referred to the others. Such solution reduces the complexity in terms of number of multiplications applying an algorithm to exploit levels of factoring reusing previous calculations. This approach is, nevertheless, very sensitive to the number of functional elements in the tensor product term and may not produce good results. Anyway, the worst case scenario of this variant can bring back the number of required multiplications to those stated in Equation 2, and the best case (no functional elements at all) will give:

$$\left(\sum_{i=1}^{N} \prod_{j=1}^{i} n_i \right) + \prod_{i=1}^{N} nz_i \tag{4}$$

For the purpose of comparisons in this paper, the sparse approach to be considered will be the more time-efficient version, *i.e.*, the variant with the previous generation and storage of nonzero elements of Q. Such variant demands the smaller number of floating point multiplications (Equation 3), but demands the storage of a sparse matrix with $\prod_{i=1}^{N} nz_i$ nonzero elements.

3. MEMORY-EFFICIENT ALGORITHM

The basic principle of the Shuffle method is the application of the decomposition of a tensor product in the ordinary product of normal factors property [16]:

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

Hence, the Shuffle algorithm consists in multiplying successively a vector by each normal factor. More precisely, vector v is multiplied by the first normal factor, then the resulting vector is multiplied by the next one, and so on, until the last. These multiplications are done using small vectors called z_{in} and z_{out} in the Algorithm 2. These small vectors store the values of v to multiply by the i^{th} matrix of the normal factor, and store the result, respectively. Their size are given for the value $nright_i$ of the related matrix. Finally, vector Υ stores the results.

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

```
1: for all $i = 1, 2, \ldots, N$ do
2:   $base = 0$
3:   for all $m = 0, 1, 2, \ldots, nleft_i - 1$ do
4:     for all $j = 0, 1, 2, \ldots, nright_i - 1$ do
5:       $index = base + j$
6:       for all $l = 0, 1, 2, \ldots, n_i - 1$ do
7:         $z_{in}[l] = v[index]$
8:         $index = index + nright_i$
9:       end for
10:      $multiply\ \ z_{out} = z_{in} \times Q^{(i)}$
11:      $index = base + j$
12:      for all $l = 0, 1, 2, \ldots, n_i - 1$ do
13:        $v[index] = z_{out}[l]$
14:        $index = index + nright_i$
15:      end for
16:    end for
17:    $base = base + (\, nright_i \times n_i)$
18:  end for
19: end for
20: $\Upsilon = v$
```

The multiplication of the last normal factor ($I_{nleft_N} \otimes Q^{(N)}$) for example, requires the multiplication of contiguous slices of vector v, because the z_{in} vectors are easily located in the product state space, and this normal factor corresponds to $Q^{(N)}$ matrices in the diagonal blocks. A more complicated process to locate the z_{in} and z_{out} vectors is required to multiply the other normal factors ($I_{nleft_i} \otimes Q^{(i)} \otimes I_{nright_i}$), *i.e.*, the normal factors where the most inner blocks are diagonal matrices. Considering the first normal factor ($Q^{(1)} \otimes I_{nright_1}$), *i.e.*, the normal factor composed by $(n_1)^2$ blocks containing diagonal matrices with the same element of matrix $Q^{(1)}$, can be treated by assembling the z_{in} vectors picking vector elements with a n_{right_1} interval.

Generalizing this process for all normal factors, the multiplication of a vector v by the i^{th} normal factor consists in *shuffling* the elements of v in order to assemble $nleft_i \times nright_i$ vectors of size n_i and multiply them by matrix $Q^{(i)}$. Thus, assuming that matrix $Q^{(i)}$ is stored as a sparse matrix, the number of operations needed to multiply a vector by the i^{th} normal factor is: $nleft_i \times nright_i \times nz_i$, where nz_i corresponds to the number of nonzero elements of the i^{th} matrix of the tensor product term ($Q^{(i)}$).

Considering the number of multiplications to all normal factors of a tensor product term, the Shuffle computational cost to perform the basic operation (multiplication of a vector by a tensor product) is given by [16]:

$$\sum_{i=1}^{N} nleft_i \times nright_i \times nz_i = \prod_{i=1}^{N} n_i \times \sum_{i=1}^{N} \frac{nz_i}{n_i} \tag{6}$$

Another feature of Shuffle algorithm is the optimization for functional elements, *i.e.*, the use of generalized tensor algebra properties and matrices reordering that were already studied in [16]. All those optimizations are very important to reduce the overhead of evaluating functional elements, but such considerations are out of the scope of this paper.

4. HYBRID METHODS

Preliminary studies originated the *Slice* algorithm [17] which is a first approach that takes advantage of the *Additive Decomposition* property and the decomposition of a tensor product into an ordinary product of normal factors, the basis of a Shuffle-like algorithm. The Additive decomposition property is quite simple and it states that any tensor product can be decomposed into an ordinary sum of hypersparse matrices [8], and for this property the matrices have all elements but one equal to zero. Assuming $\hat{q}_{(i_1,\ldots,i_{N-1},j_1,\ldots,j_N)}$ the hypersparse matrix of order $\prod_{i=1}^{N} n_i$ composed by only one nonzero element which is in position $i_1, \ldots, i_N, j_1, \ldots, j_N$ and it is equal to $\prod_{k=1}^{N} q^{(k)}_{i_k,j_k}$, the additive decomposition property can be stated as:

$$\begin{array}{l} Q^{(1)} \otimes Q^{(2)} \otimes \ldots \otimes Q^{(N-1)} \otimes Q^{(N)} = \\ \quad \sum_{i_1=1}^{n_1} \cdots \sum_{i_N=1}^{n_N} \sum_{j_1=1}^{n_1} \cdots \sum_{j_N=1}^{n_N} \\ \qquad \left(\hat{q}^{(1)}_{(i_1,j_1)} \otimes \ldots \otimes \hat{q}^{(N)}_{(i_N,j_N)} \right) \end{array} \tag{7}$$

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

The Algorithm 3 proposes the use of this property decomposing the first $N-1$ matrices of each tensor product, then performing the shuffling operation with the last matrix, for each possible $\hat{q}^{(k)}_{(i,j)}$ special matrix of the term. The new factor created is called *Additive Unitary Normal Factor* (*AUNF*). It is clear that the nonzero elements generation follows the principles of the Sparse algorithm, and also can use any optimization technique [9]. For each nonzero element e, it picks a *slice* of the vector v (named as v_{in}), according to the row position of e, and multiply all elements of v_{in} by element e. The resulting vector v_{in} is multiplied by the last matrix $Q^{(N)}$ accumulating the result v_{out} in the positions of vector Υ corresponding to the column position of element e. The computational cost in multiplications is given by:

$$\left(\prod_{i=1}^{N-1} nz_i \right) \times \ [\ (N-2) + (n_N) + (nz_N)\ ] \tag{8}$$

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

Table 1: Split approach as a generalization of traditional algorithms

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

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

where $\prod_{i=1}^{N-1} nz_i$ is the number of *AUNF* to be generated. For each one, we have $(N-2)$ multiplications needed to its generation (considering the total of $(N-1)$ matrices in the tensor product). This scalar is multiplied by the vector n_N times, and then the Shuffle is done with the last matrix costing nz_N multiplications.

More details of the Slice algorithm can be found in [17], and even though as a preliminary version of an hybrid numerical method, the Slice algorithm has shown better overall performance than the traditional Shuffle algorithm for some practical SAN examples [18]. But due to its fixed structure, it is not adequate for all models. However, it already takes advantage of the Additive Decomposition property, as a better way to perform the vector-descriptor product in parallel implementations, due the independence among the normal factors generated. The reader interested in parallel versions of solvers for Markovian models may consult [6] and [4].

5. SPLIT ALGORITHM

SAN models of practical applications are naturally sparse, the local part of a descriptor is intrinsically very sparse due to the tensor sum structure. The synchronizing events are mostly used to describe exceptional behaviors, therefore it lets this part of the descriptor also quite sparse. Hence, following the Slice algorithm guidelines, each tensor product of matrices can be partitioned in two different groups: the first one with the more sparse matrices; and the second one with the matrices with a larger number of nonzero elements. A Sparse-like approach could be applied to the first group of K matrices generating *AUNF*, as Slice does with the first $N-1$ matrices. Each one of those *AUNF* should be *tensorly* multiplied by the second group of matrices using a Shuffle-like approach (Slice deals only with the last matrix in this part). Considering that, the idea is to split the tensor terms in two sets of matrices treating them in two different ways, with a certain degree of dependence. Due to this, the novel algorithm called *Split* is also a generalization for Slice algorithm, because it follows the idea of having distinct parts to treat.

Table 1 presents the general idea of the Split algorithm graphically. It is important to notice that the index of the matrix chosen for delimiting the end of the first set is assigned to the *cut-parameter* σ separating the first K matrices. It is possible to observe that the Sparse ($\sigma = N$), the Slice ($\sigma = N-1$) and the Shuffle ($\sigma = 0$) methods are also particular cases of the Split algorithm, *i.e.*, they are also possibilities of spliting in a tensor term composed by N matrices.

Algorithm 4 defines formally the steps of the Split approach. It consists in the computation of the element e of each *AUNF* by multiplying one nonzero element of each matrix of the first set of matrices (from $Q^{(1)}$ to $Q^{(\sigma)}$). According to the elements row indexes used to generate element e, a contiguous slice of input vector v is taken (v_{in}). Vector v_{in} of size $nright_\sigma$ (corresponding to the product of the order of all matrices after the *cut-parameter* σ of the tensor product term) is multiplied by element e. The resulting vector (v'_{in}) is used as input vector to the Shuffle-like multiplication by the tensor product of the matrices in the second set of matrices (from $Q^{(\sigma+1)}$ to $Q^{(N)}$), due to this there is a kind of dependency between both sets.

Algorithm 4 Split Algorithm - $\Upsilon = \upsilon \times \otimes_{i=1}^{N} Q^{(i)}$

```
1: $\Upsilon = 0$
2: for all $i_1, \ldots, i_\sigma, j_1, \ldots, j_\sigma \in \theta(1 \ldots \sigma)$ do
3:   $e = 1$
4:   $base_{in} = base_{out} = 0$
5:   for all $k = 1, 2, \ldots, \sigma$ do
6:     $e = e \times q^{(k)}_{(i_k, j_k)}$
7:     $base_{in} = base_{in} + ((i_k - 1) \times nright_{(k-1)})$
8:     $base_{out} = base_{out} + ((j_k - 1) \times nright_{(k-1)})$
9:   end for
10:  for all $l = 0, 1, 2, \ldots, nright_\sigma - 1$ do
11:    $\upsilon_{in}[l] = \upsilon[base_{in} + l] \times e$
12:  end for
13:  for all $i = \sigma + 1, \ldots, N$ do
14:    $base = 0$
15:    for all $m = 0, 1, 2, \ldots, nleft_i - 1$ do
16:      for all $j = 0, 1, 2, \ldots, \frac{nleft_i}{nleft_\sigma} - 1$ do
17:        $index = base + j$
18:        for all $l = 0, 1, 2, \ldots, n_i - 1$ do
19:          $z_{in}[l] = \upsilon_{in}[index]$
20:          $index = index + nright_i$
21:        end for
22:        $multiply\ \ z_{out} = z_{in} \times Q^{(i)}$
23:        $index = base + j$
24:        for all $l = 0, 1, 2, \ldots, n_i - 1$ do
25:          $\upsilon_{in}[index] = z_{out}[l]$
26:          $index = index + nright_i$
27:        end for
28:      end for
29:      $base = base + (nright_i \times n_i)$
30:    end for
31:  end for
32:  for all $l = 0, 1, 2, \ldots, nright_\sigma - 1$ do
33:    $\Upsilon[base_{out} + l] = \Upsilon[base_{out} + l] + \upsilon_{in}[l]$
34:  end for
35: end for
```

The computational cost in number of multiplications (Equation 9) for the Split algorithm is computed taking into account the number of multiplications performed to generate the nonzero element of each *AUNF* $(\sigma - 1)$, plus the number of multiplications of the scalar by each position value of the vector υ_{in}. There is also the cost to multiply the values in the input vector υ'_{in} by the tensor product of matrices in the Shuffle-like part.

$$\left(\prod_{i=1}^{\sigma} nz_i\right) \left[(\sigma - 1) + \left(\prod_{i=\sigma+1}^{N} n_i\right) + \left(\prod_{i=\sigma+1}^{N} n_i \times \sum_{i=\sigma+1}^{N} \frac{nz_i}{n_i}\right)\right] \tag{9}$$

In practical implementations of vector-descriptor multiplication algorithms, improvements can be done to speedup the execution. These optimizations can change significantly the theoretical computational cost presented in the Equation 9.

Regarding the Shuffle algorithm, there is an optimization on the way of handling identity matrices. Those matrices do not need to generate normal factors, since being identity matrices, they generate a normal factor that is also an (huge) identity matrix itself. The computational cost is clearly reduced in the Shuffle algorithm execution when using this solution. It corresponds to transform the number of floating point multiplications equation for the Shuffle algorithm (Equation 6) to:

$$\prod_{i=1}^{N} n_i \times \sum_{\substack{i=1 \\ iff Q^{(i)} \neq Id}}^{N} \frac{nz_i}{n_i} \tag{10}$$

This improvement suggests the same skipping-identities optimization to the Shuffle-like part (matrices $Q^{(\sigma+1)}$ to $Q^{(N)}$) of the Split algorithm (4) identifying if the matrix indexed by variable i of the algorithm ($Q^{(i)}$) is not an identity matrix, adding to the cost $\frac{nz_i}{n_i}$ multiplications only for these ones. Analougously to Shuffle algorithm, Equation 9 will be rewritten changing the shuffle-part cost accordingly to σ. The resulting number of floating point multiplications for the Split algorithm will be:

$$\left(\prod_{i=1}^{\sigma} nz_i\right) \left[(\sigma - 1) + \left(\prod_{i=\sigma+1}^{N} n_i\right) + \left(\prod_{i=\sigma+1}^{N} n_i \times \sum_{\substack{i=\sigma+1 \\ iff Q^{(i)} \neq Id}}^{N} \frac{nz_i}{n_i}\right)\right] \tag{11}$$

Usually the tensor product terms of a SAN model are very sparse (a few thousands nonzero elements). The only cases where a more significant number of nonzero elements is found are those when we are dealing with a tensor product term with many identity matrices. It is important to recall that to each *AUNF* scalar e is computed as the product of one single element of each matrix. The second optimization is the precomputation of these nonzero elements and their storage consequently, that for pure sparse solving approaches, this optimization was already largely studied [9]. It results in a reduction of the Split algorithm computational cost similar to that one presented in Section 2 regarding the computation of nonzero elements. Hence, the final definition of the number of floating point multiplications for the Split algorithm is no longer defined by Equation 11, but as:

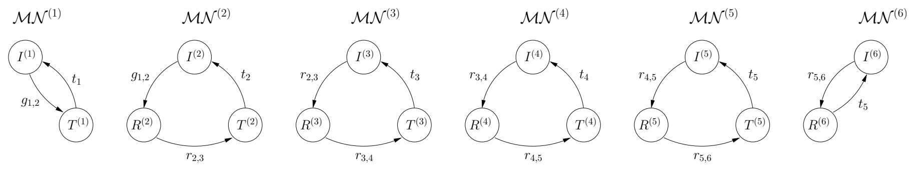
$$\left(\prod_{i=1}^{\sigma} nz_i\right) \left[\left(\prod_{i=\sigma+1}^{N} n_i\right) + \left(\prod_{i=\sigma+1}^{N} n_i \times \sum_{\substack{i=\sigma+1 \\ iff Q^{(i)} \neq Id}}^{N} \frac{nz_i}{n_i}\right)\right] \tag{12}$$

Note that it is not surprising if for very sparse tensor products the best Split option was the pure Sparse approach which, obviously, is much more effective if the nonzero elements do not have to be recomputed at each vector multiplication but it means that we need to store a possibly huge matrix. Therefore, the Split choice must also balance the computational cost in terms of multiplications and memory needs.

6. NUMERICAL RESULTS

The experiments results were collected running a prototype of traditional methods and the new one, the Split algorithm, on a 3.2 GHz Intel Xeon under Linux operating system with 4 GBytes of memory. The prototype was compiled using g++ compiler with optimizations ($-O3$). The methods were executed for practical examples of SAN models [13, 3], running all tensor product terms in all possible *cut-parameters*, collecting outputs for at least 100 runs. The results were obtained in time intervals with 95% of confidence.

The presented results for Split algorithm executions are the collection of the best execution times obtained given the confidence interval. It also considers different cut-parameters σ to each tensor product term in the descriptor. However, we consider a fix order of automata and no automata permutations were considered. In such way, it is possible to say that the Split results presented here tend to force the trade-off between time and memory efficiency towards time savings. It would be absurd to take the oposite decision, since



Type	Event	Rate	Type	Event	Rate
loc	$t_1, \ldots, t_4$	μ	syn	$r_{2,3}$	$f_{2,3}$
syn	t_5	μ	syn	$r_{3,4}$	$f_{3,4}$
syn	$g_{1,2}$	$f_{1,2}$	syn	$r_{4,5}$	$f_{4,5}$
			syn	$r_{5,6}$	$f_{5,6}$

$$f_{1,2} = [((st\ \mathcal{MN}^{(3)} \ != T^{(3)}) \ \&\& \ (st\ \mathcal{MN}^{(4)} \ != T^{(4)}))] * \lambda$$
$$f_{2,3} = [((st\ \mathcal{MN}^{(1)} \ != T^{(1)}) \ \&\& \ (st\ \mathcal{MN}^{(4)} \ != T^{(4)}) \ \&\& \ (st\ \mathcal{MN}^{(5)} \ != T^{(5)}))] * \lambda$$
$$f_{3,4} = [((st\ \mathcal{MN}^{(1)} \ != T^{(1)}) \ \&\& \ (st\ \mathcal{MN}^{(2)} \ != T^{(2)}) \ \&\& \ (st\ \mathcal{MN}^{(5)} \ != T^{(5)}) \ \&\& \ (st\ \mathcal{MN}^{(6)} \ != T^{(6)}))] * \lambda$$
$$f_{4,5} = [((st\ \mathcal{MN}^{(2)} \ != T^{(2)}) \ \&\& \ (st\ \mathcal{MN}^{(3)} \ != T^{(3)}) \ \&\& \ (st\ \mathcal{MN}^{(6)} \ != T^{(6)}))] * \lambda$$
$$f_{5,6} = [((st\ \mathcal{MN}^{(3)} \ != T^{(3)}) \ \&\& \ (st\ \mathcal{MN}^{(4)} \ != T^{(4)}))] * \lambda$$

Figure 1: Ad Hoc WSN SAN **Model (**6 **nodes)**

		Shuffle			Split			Sparse		
N	PSS	time (s)	size (Kb)	fpm (Eq. 10)	time (s)	size (Kb)	fpm (Eq. 12)	time (s)	size (Kb)	fpm (Eq. 3)
6	324	0.00075	**1.52**	7,344	**0.00013**	5.30	1,798	0.00025	18.93	**1,114**
8	2,916	0.00633	**2.25**	93,312	**0.00071**	103.22	14,332	0.00163	219.88	**13,928**
10	26,244	0.07454	**2.99**	1,084,752	**0.01307**	170.11	185,364	0.01520	2,510.61	**160,488**
12	236,196	0.87020	**3.72**	11,967,264	**0.18774**	1,461.79	2,230,740	0.20951	27,513.35	**1,760,616**
14	2,125,760	9.35304	**4.46**	127,545,900	**1.75147**	13,536.58	22,674,856	2.07020	292,060.08	**18,691,560**
16	19,131,900	96.44355	**5.19**	1,326,477,700	**17.99368**	118,103.25	235,251,512	21.02443	3,028,726.82	**193,838,184**

Table 2: Ad Hoc WSN Model Results

forcing Split to save memory would result in practically the same efficiency as the Shuffle algorithm, which already gives the more memory-efficient solution.

6.1 Ad Hoc WSN model

The SAN model in the Figure 1 represents a chain of mobile nodes in a Wireless Sensor Network (Ad Hoc WSN model) running over the 802.11 standard for ad hoc networks. This model [13] resembles the ad hoc forwarding experiment presented in [20] using SAN. The chain of N nodes modeled, have the first node $\mathcal{MN}^{(1)}$ (*Source* automaton) that generates packets as fast as the standard allows. The packets are forwarded through the chain by the *Relay* automata called $\mathcal{MN}^{(i)}$, where the variable i is among the value 2 and $(N-1)$, to the last node $\mathcal{MN}^{(N)}$ (*Sink* automaton).

This SAN model family generically has $(N-2)$ local events and N synchronizing events among N automata. The *Descriptor* Q for these models is formed by a set of $[(N-2)+2N]$ tensor product terms. The numerical results were obtained for this model extending the number of automata and, consequently, the number of tensor product terms.

Table 2 shows the results for the three methods, Shuffle, Split and Sparse, divided each in three columns representing time in seconds (per iteration), size in Kb and the computational cost in floating point multiplications (fpm). The column PSS stands for the product state space $\prod_{i=1}^{N} n_i$, which is the cartesian product of the automata orders (the size of the probability vector). As the number of nodes in the mobile chain increases ($N = 6, 8, 10, 12, 14, 16$), so does the time to solve as well as the memory needed. Since Shuffle is memory-efficient, it is slower than the other two methods. These methods, on the contrary, are much more memory consuming, to pay for the time efficiency.

For small models ($N < 12$ nodes) the time and memory efficiency are reasonable enough to be dealt regardless of algorithm in virtually any machine. It demands few more the 2 Mbytes in the sparse approach and it takes less than 100 miliseconds per iteration for all algorithms. However, even in this small examples we notice the quite impressive memory efficiency of the Shuffle algorithm that keeps the memory needs insignificant even for quite large models.

The remarkable result in Table 2 is the better time efficiency that beats even the sparse approach. Although, for each tensor product, sparse approach could be faster for most cases, terms with many identity matrices could have a better time efficiency in Shuffle or Split algorithms. Since a SAN model is composed of many tensor product terms, Split will be the best option in product terms where the sparse approach could be faster, but too memory demanding.

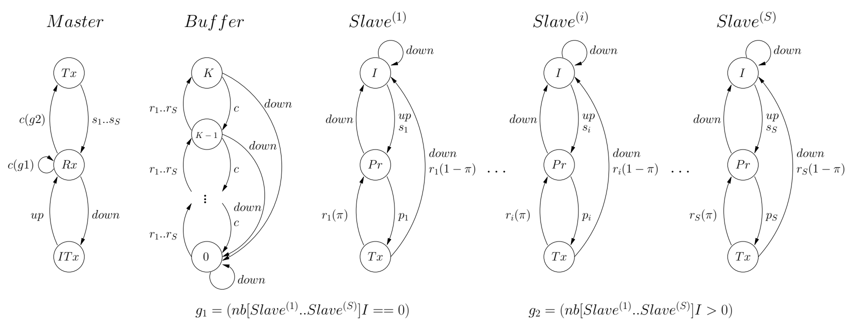
This is clearly the case of the largest model ($N = 16$) where Split is around 3 seconds faster than Sparse with a memory need of little more than 100 Mbytes, rather than 3 Gbytes needed by the sparse solution, *i.e.*, Split takes for this case almost 30 times less memory and still improves the time efficiency compared to Sparse. It is important to observe, as well, that this model has a considerable product state space of more than 19 million states. Such large model could be nearly intractable if a time and memory efficient solution is not found.

It is also noticeable, that the number of floating point multiplications computed to each algorithm is not relevant to indicate a better performance in time. This phenomenon, obviously need more observation, but approximatively, we dare to say that other indications like other floating point operations and maybe allocations/desallocations may have considerable influence on the performance of the algorithms.

6.2 Master-Slave Parallel Model

This SAN model in the Figure 2 refers to an evaluation of the master-slave parallel implementation of the Propagation algorithm considering assynchronous communication [3], indicating to parallel program developers what are the possible execution bottlenecks before the implementation. This model contains one *Master* automaton, one huge *Buffer* automaton, and S automata $Slave^{(i)}$, where $i = 1 \ldots S$.

The total number of automata is generically given by $(S+2)$, having S local events and $(3S-3)$ synchronizing events. The *Descriptor* Q in this case, is formed by a set of $(7S-8)$ tensor terms. This model was extended to run for different numbers of slaves ($S = 3, 4, 5, 6, 7, 8, 10, 12$) and the buffer is of forty po-



Event	Rate
up	λ
$down$	μ
c	σ
s_i	δ
r_i	α
p_i	γ

Figure 2: Master-Slave Parallel SAN **Model**

		Shuffle			Split			Sparse		
S	PSS	time (s)	size (Kb)	fpm (Eq. 10)	time (s)	size (Kb)	fpm (Eq. 12)	time (s)	size (Kb)	fpm (Eq. 3)
3	3,321	0.00932	**11.19**	128,385	**0.00174**	175.11	37,902	0.00257	381.07	**23,672**
4	9,963	0.02884	**14.23**	497,097	**0.00711**	508.61	116,780	0.00869	1,435.29	**90,948**
5	29,889	0.09830	**16.63**	1,797,228	**0.02231**	1,447.69	399,036	0.02537	5,229.25	**333,608**
6	89,667	0.35073	**20.31**	6,488,829	**0.09305**	3,183.29	1,742,271	0.10873	18,531.62	**1,184,724**
7	269,001	1.31777	**23.35**	22,488,916	**0.37132**	9,446.93	6,555,978	0.40385	64,222.72	**4,108,760**
8	807,003	4.58053	**26.43**	76,534,019	**1.23928**	28,236.07	23,037,480	1.34866	231,315.37	**14,802,495**
10	7,263,030	50.07517	**32.43**	847,176,190	**12.92194**	240,596.27	246,651,139	13.90858	2,363,273.71	**151,247,442**
12	65,367,200	535.28774	**38.54**	9,137,063,300	**135.94263**	2,282,495.12	2,787,370,431	147.59428	26,195,236.61	**1,676,492,676**

Table 3: Master-Slave Parallel Model Results

sitions ($K = 40$). Table 3 shows the results for all these model extensions.

The results are consistent with those obtained in previous section. Split once again demonstrate a better time efficiency. In fact, it presents, in general, results a little faster than the sparse approach, *i.e.*, roughly around 10% faster in the large models.

However, the memory savings obtained in these second set of examples seem less impressive than those obtained for the Ad Hoc WSN models. Split still gives a considerable reduction for the huge last example ($S = 12$) bringing the memory needs from nearly intractable 26 Gbytes in sparse approach to large, but tractable 2.2 Gbytes. Once again, it is important to keep in mind that we are dealing with a model with a 65 million states, and then some significant amount of memory and time is expected to achieve a stationary or transient solution.

One interesting point is to observe how inadequate the number of floating point multiplication is to predict how fast an algorithm will execute. We still can observe this phenomenon and, as said before, a deeper analysis has to be done.

7. CONCLUSION

The main contribution of this paper is the proposition of a flexible hybrid vector-descriptor algorithm. Applying the proposed algorithm to SAN models of real problems verifies the good trade-off between memory and time efficiency of the Split algorithm when compared to traditional Sparse and Shuffle approaches. Considering that we need many iterations to calculate the final probability vector, the memory and time spent tradeoffs surely should be evaluated and balanced, maybe according to the available time and computational resources. Nevertheless, it is also shown that the Split algorithm is flexible enough to deliver in extreme cases at least the same time efficiency as the sparse approach, or, alternatively, the same memory efficiency as the shuffle approach.

Since tensor terms can be formed differently due the different structured models we deal with, the performance can also be very dependent on the choice of matrices placed in each group. The tensor product terms that do not have too many identity matrices, or no identities at all, can be multiplied in a sparse fashion. However, Shuffle deals better with terms containing many identities because it simply jumps the execution for the next matrix to multiply. Our experience with structured models suggests that these tensor product terms, with a reasonable number of identity matrices, are the most commonly encountered ones, but if the memory available is not a problem, it is better to treat them in a sparse manner as much as possible.

A clearly open problem is the choice of division point in each tensor term (choice of cut-parameter σ) and, even more important, the choice of order for terms. However, the research for an heuristic to automatically choose the best order of matrices and the *cut-parameter*, for each tensor product is considerable research challenge. This is not a trivial task, due to the tensor product term formation and intrinsic matrices details such as order, total nonzero elements and computational cost in multiplications. These parameters opens the possibility of a thorough analysis of the related theoretical computational cost. Another aspect to be considered is the fact that we can have a tradeoff between memory usage and time spent, *i.e.*, if one have lots of memory and wants performance, the *cut-parameter* could be more easily shifted to use the sparse approach, while if memory is limited the *cut-parameter* should allow more weight in shuffle-like part.

Additionally, the proposed Split algorithm could be enhanced with considerations about the impact of functional elements (with their particular dependencies) in the descriptor. A similar work about these functional dependency changed completely the performance of the Shuffle algorithm [16]. It is only natural to estimate that similar gains with functional dependencies analysis and possible automata permutations could benefit the Split algorithm as well.

Finally, it is also possible to foresee an even more complex analysis that considers not only a sequential version of the Split algorithm, but also parallel implementations. For the sequential version, memory and time efficiency are dealt as a single demand, but parallel implementations should consider the amount of memory needed, volume of data exchanged and processing demands to be as evenly as possible distributed among parallel machines. Obviously, this further analysis is much more deep and complex since neither the number of floating point multiplications, nor any other known index for that matter, seems to be a good estimation of processing time (see the previous section remarks). Nevertheless, it still seems an interesting field of future research.

These future works give hope that Split algorithm can be subject of considerable improvements in a near future. Despite that, an easy and not very thorough analysis of the presented examples, and consequent choice of automata order and cut-parameter σ, shows quite impressive gains of processing time for the Split sequencial implementation compared to Shuffle approach without paying the very high memory costs of the Sparse approach. At least these results lets say that the Split algorithm already is a better choice for practical vector-descriptor products.

8. REFERENCES

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

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

[3] L. Baldo, L. Brenner, L. G. Fernandes, P. Fernandes, and A. Sales. Performance Models for Master/Slave Parallel Programs. *Electronic Notes In Theoretical Computer Science*, 128(4):101–121, April 2005.

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

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

[6] P. Buchholz. A distributed numerical/simulative algorithm for the analysis of large continuous time markov chains. In *PADS '97: Proceedings of the eleventh Workshop on Parallel and Distributed Simulation*, pages 4–11, Washington, DC, USA, 1997.

[7] P. Buchholz. A new approach combining simulation and randomization for the analysis of large continuous time Markov Chains. *ACM Trans. Model. Comput. Simul.*, 8(2):194–222, 1998.

[8] P. Buchholz, G. Ciardo, S. Donatelli, and P. Kemper. Complexity of memory-efficient Kronecker operations with applications to the solution of Markov models. *INFORMS Journal on Computing*, 13(3):203–222, 2000.

[9] P. Buchholz, G. Ciardo, S. Donatelli, and P. Kemper. Complexity of memory-efficient kronecker operations with applications to the solution of markov models. *INFORMS J. on Computing*, 12(3):203–222, 2000.

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

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

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

[13] F. L. Dotti, P. Fernandes, A. Sales, and O. M. Santos. Modular Analytical Performance Models for Ad Hoc Wireless Networks. In 3rd *International Symposium on Modeling and Optimization in Mobile, Ad Hoc, and Wireless Networks*, pages 164–173, Trentino, Italy, April 2005. IEEE Press.

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

[15] P. Fernandes. *Mï£¡hodes numï£¡iques pour la solution de sustï£¡es Markoviens ï£¡grand espace d'ï£¡ats*. PhD thesis, Institut National Polytechnique de Grenoble, France, 1998.

[16] P. Fernandes, B. Plateau, and W. J. Stewart. Efficient descriptor - Vector multiplication in Stochastic Automata Networks. *Journal of the ACM*, 45(3):381–414, 1998.

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

[18] P. Fernandes, R. Presotto, A. Sales, and T. Webber. An Alternative Algorithm to Multiply a Vector by a Kronecker Represented Descriptor. Technical Report TR 047, PUCRS, Porto Alegre, 2005. http://www.inf.pucrs.br/tr/tr047.pdf.

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

[20] J. Li, C. Blake, D. S. J. D. Couto, H. I. Lee, and R. Morris. Capacity of Ad Hoc Wireless Networks. In 7th *Annual International Conference on Mobile Computing and Networking*, pages 61–69, Rome, Italy, July 2001. ACM Press.

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

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

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

[24] B. Plateau and K. Atif. Stochastic Automata Networks for

modelling parallel systems. *IEEE Transactions on Software Engineering*, 17(10):1093–1108, 1991.

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

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