From *Performance Evaluation* 1998.

An efficient disk-based tool for solving large Markov models

Daniel D. Deavours and William H. Sanders [1]

Department of Electrical and Computer Engineering and Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, Urbana, Illinois 61801, USA

Abstract

Very large Markov models often result when modeling realistic computer systems and networks. We describe an efficient tool for solving general, large Markov models on a typical engineering workstation. It uses a disk to hold the state-transition-rate matrix (possibly compressed), a variant of block Gauss-Seidel as the iterative solution method, and an innovative implementation that involves two parallel processes communicating by shared memory. We demonstrate its use on two large, realistic performance models.

1 Introduction

A wide variety of high-level specification techniques now exists for Markov models. These include, among others, stochastic Petri nets, stochastic process algebras, various types of block diagrams, and non-product form queuing networks. In most cases, very large Markov models result when one tries to model realistic systems using these specification techniques. The Markov models are typically quite sparse (adjacent to few nodes), but contain a large number of states. This problem is known as the "largeness problem." Techniques that researchers have developed to deal with the largeness problem fall into two general categories: those that avoid the large state space (for example, by lumping), and those that tolerate the large state space (for example, by recognizing that the model has a special structure and storing it in a compact form). While many largeness avoidance and tolerance techniques exist, few

[1] This work was supported, in part, by NASA Grant NAG 1-1782, and by DARPA/ITO under Contract No. DABT63-96-C-0069. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA/ITO.

are applicable to models without special structure. Methods are sorely needed that permit the solution of very large Markov models without requiring them to have special properties or a particular structure.

In this paper, we describe two similar implementations for solving Markov models with very large state spaces on a typical engineering workstation. The second implementation, which we call a compressed block Gauss-Seidel (CBGS) solver, comes from our experience with the first, which we call a block Gauss-Seidel (BGS) solver, and tries to address some of the limiting aspects of the first, primarily by file compression. Consequently, the second implementation can be thought of as an extension of the first, although it required building a completely new program. We use the word *tool* when referring to both the BGS and CBGS solvers.

The tool makes no assumptions about the underlying structure of the Markov process, and requires little more memory than what is necessary to hold the solution vector itself. It uses a disk to hold the state-transition-rate matrix, a variant of block Gauss-Seidel as the iterative solution method, and an innovative two-process implementation that effectively overlaps retrieval of blocks of the state-transition-rate matrix from disk and computation on the blocks. The tool can solve models with ten million states and about 100 million transitions on a machine with 128 MB of main memory. The first implementation stores the state-transition-rate matrix on a disk in a clever manner, minimizing the overhead in retrieving it from the disk and doing computation. The second implementation takes this a step further and does file compression to increase the effective I/O throughput, but at a cost of significantly higher CPU overhead.

In addition, the first implementation employs a dynamic method for determining the number of iterations to perform on a block before beginning on the next, which we show empirically to provide a near optimum time to convergence. Solution time is typically quick even for very large models, with only about 20% of the CPU time spent retrieving blocks from disk and 80% of the CPU resources available to perform the required computation.

In addition to describing the architecture and implementations, we illustrate the tool's use on two realistic models: one of a Kanban manufacturing system [3], and another of the Courier protocol stack executing on a VME bus-based multiprocessor [15]. Both models have appeared before in the literature, and are excellent examples of models that have very large state spaces for realistic system parameter values. In particular, both models have been used to illustrate the use of recently developed Kronecker-based methods [8,3], and the Courier protocol has been used to illustrate an approximate method based on lumping [15]. Both numerical results and solution times are presented for each model and, when possible, compared to previously obtained values and solution times. In each case, we can obtain an exact solution (to the desired

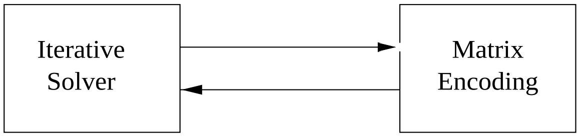

Fig. 1. Solution paradigm.

precision) in significantly less time than previously reported using Kronecker-based methods. It is thus our belief that if sufficient disk space is available to hold the state-transition-rate matrix, our approach is the method of choice for exact solutions.

The remainder of the paper is organized as follows. First, in Section 2, we address issues in the choice of a solution method for very large Markov models, comparing three alternatives: Kronecker-based methods (e.g., [8,3]), "on-the-fly" methods [4], and the disk-based method that we ultimately choose. This section presents clear arguments for the desirability of disk-based methods if sufficient disk space is available. In Section 3, we review the block Gauss-Seidel algorithm that will form the core of our solver. Section 4 then describes the architecture of the BGS solver, followed by the results in Section 5 based on the two models described earlier. Finally, Section 6 describes the second implementation.

2 The Case for Disk-Based Methods

We first evaluate disk-based methods in the context of other state-of-the-art numerical solution techniques, namely Kronecker-based techniques and "on-the-fly" techniques. To evaluate each method, we introduce a paradigm based on Figure 1. Here, we divide the numerical solution process into the iterative solver and the matrix encoding. The key to solving large matrices is to encode the matrix so that it takes little main memory (RAM), but still allows quick access to matrix elements. The iterative solver is thus a data consumer, and the matrix encoder is a data producer. We would like for both to be as fast as possible to obtain a solution quickly. An additional important factor is how effectively a particular iterative method uses the data it consumes. For example, certain iterative methods, such as Gauss-Seidel [13] and adaptive Gauss-Seidel [6], typically do more effective work with the same number of accesses to the matrix as Jacobi or the Power method, and hence do not require as high a data production rate to efficiently obtain a solution. We want to find a fast and compact but general matrix encoding scheme and an effective iterative method with a low data consumption rate.

The first class of encoding schemes we consider are those of Kronecker-based methods. These methods require and make use of the fact that in certain models, particular parts of the model (called submodels) interact with one another in a limited way. One way to insure a model has this structure is to construct it according to a prescribed set of rules from smaller models, as is done, for example, in the case of stochastic automata networks [13]. By following these rules, one may express the transition rate matrix for the model as a function of Kronecker operators on the transition rate matrices of the submodels.

More recently there has been work on a type of model decomposition called superposed generalized stochastic Petri nets (SGSPNs) [2,3,5,7,8]. SGSPNs are essentially independent models that may be joined by synchronization of a transition. We believe [3] to be the state of the art in Kronecker operator methods, and although the more recent techniques can solve a much larger class of models than originally proposed in [5], they are still restrictive in the models that they can effectively solve.

To evaluate the speed of the Kronecker operator methods, ye observe the rates at which the iterative solver and matrix encoding operate. We have observed that on a 120 MHz Hewlett-Packard Model C-110, the SOR iterative solver can consume data at a rate of about 50 MB/second. From numbers published by Kemper [8], we estimate that his implementation of the Kronecker-based method can produce data at a rate of 700 KB/second on an 85 MHz Sparc 4, and we extrapolate that the rate would be about 2 MB/second on our HP C-110. Since both the data production and consumption require the CPU, the whole process will proceed at a rate of about 1.9 MB/second. Kemper's method is also restricted to Jacobi or the Power method, which usually exhibit poor convergence characteristics, so the effectiveness of its use of generated data is low. Ciardo and Tilgner [3] present their own tool, but they do not present data in such a way that we can analyze the data generation rate. We can compare actual times to solution for their benchmark model, however, and do so in Section 5. Ciardo gives algorithms to perform Gauss-Seidel on a Kronecker representation in [2], but we are not aware of a tool that implements these algorithms.

The second class of encoding schemes we considered for implementation in this tool are "on-the-fly" methods introduced in [4]. On-the-fly methods have none of the structural restrictions of Kronecker-based methods, and they can operate on nets with general enabling predicate and state change functions, such as those present in stochastic activity networks [10,11]. In addition, they can obtain a solution with little additional memory, or perhaps even less memory than needed by SGSPN solvers, while at the same time using Gauss-Seidel or variants. However, the prototype implementation described in [4] generates data at about 440 KB/second on an HP C-110. Although [4] introduces iterative methods that are usually more effective than Jacobi or the Power method in their use of data, the overall solution speed for these methods will be somewhat slower than for Kronecker-based methods, but still reasonable, given that they can be used without restrictions on the structure of a model.

The final class we considered was that of disk-based methods, where the workstation disk holds an encoding of the state-transition matrix. If we can find an iterative method that accesses data from the state-transition-rate matrix in a regular way and use a clever encoding, disks can deliver data to an iterative algorithm at a high rate, 5 MB/second or higher, with low CPU overhead. Furthermore, high performance disks are inexpensive relative to the cost of RAM, so we would like to find a way to utilize disks effectively. Experimental results show that if we can do both disk I/O and computation in parallel, we can perform Gauss-Seidel at a rate of 5 MB/second while using the CPU only 30% of the time. Thus disk-based methods have the potential to greatly outperform Kronecker and on-the-fly methods, while using even less memory, at the cost of providing a disk that is large enough to hold the state-transition-rate matrix of the Markov model being solved. The challenge is to find a more effective solution method that has a data consumption rate of about 5 MB/second at 80% CPU utilization.

Clearly, the method of choice depends on the nature of the model being solved, and the hardware available for the solution. If the state-transition-rate matrix is too large to fit on available disk space and the model meets the requirements of Kronecker-based methods, then they should be used. If the model does not fit on disk and does not meet the requirements of Kronecker-based methods, on-the-fly methods should be used. However, SCSI disks are inexpensive relative to RAM (in December 1997, approximately $850 (U.S.) for 4 GB fast wide SCSI), so space can be made available inexpensively to store the state-transition-rate matrix. Since a single disk can provide the high data production rate only for sequential disk access, the efficiency of disk-based methods will depend on whether we can find a solution algorithm that can make effective use of the data in the sequential manner. We discuss how to do this in the following sections.

3 Block Gauss-Seidel Review

To make effective use of a disk-based solver, the iterative solver must have two important features. First, the solver must access the transition-rate matrix in a predictable and consistent way throughout different iterations. Most iterative solvers, such as Jacobi, Gauss-Seidel, and projection methods, exhibit this feature. Second, once a portion of the iteration matrix has been generated, we would like the solver to reuse the data as much as possible. Most iterative solvers access each element only once through an iteration, so an element is generated, used once, and then discarded. Since the cost of generating the element is high, we would like a method that exhibits some locality so that an element may be generated and used several times before being discarded. This property of data reuse is important and allows us to effectively slow down the data consumption rate to match the data production rate. Block Gauss-Seidel (BGS) has all the desired properties we describe.

BGS is a relatively simple technique that has a long history in computational sciences. We believe that it is appropriate to revisit BGS in the context of solutions to large Markov chains, and we show an implementation that performs favorably compared to SGSPN solution methods.

First, we review Block Gauss-Seidel. We wish to solve the linear equation $\pi Q = 0$, where π is the steady state probability vector, and Q is the transition rate matrix. Let $Q \in \mathcal{R}^{n \times n}$. The k-th Gauss-Seidel iteration does

$$\pi_i^{(k+1)} = \frac{-1}{q_{ii}} \left(\sum_{j=1}^{i-1} \pi_j^{(k+1)} q_{ji} + \sum_{j=i+1}^{n} \pi_j^{(k)} q_{ji} \right) \tag{1}$$

for i from 1 to n. Put simply, BGS is Gauss-Seidel where π_i is a subvector of π and q_{ji} is a submatrix of Q.

More precisely, we partition π into N subvectors: $\Pi_i = (\pi_{m_i}, \pi_{m_i+1}, \ldots, \pi_{m_{i+1}-1})$. Now we partition Q into $N \times N$ submatrices accordingly, so that

$$Q_{ji} = \begin{pmatrix} q_{m_j,m_i} & \cdots & q_{m_j,m_{i+1}-1} \\ q_{m_j+1,m_i} & \cdots & q_{m_j+1,m_{i+1}-1} \\ \vdots & \ddots & \vdots \\ q_{m_{j+1}-1,m_i} & \cdots & q_{m_{j+1}-1,m_{i+1}-1} \end{pmatrix},$$

where $m_1 = 1$ (we add commas to the subscripts for clarity). Now we may substitute Π_i for π_i and Q_{ji} for q_{ji}, and we rewrite Equation 1 as

$$\Pi_i^{(k+1)} = - \left(\sum_{j=1}^{i-1} \Pi_j^{(k+1)} Q_{ji} + \sum_{j=i+1}^{N} \Pi_j^{(k)} Q_{ji} \right) Q_{ii}^{-1}$$

Performing this computation for i from 1 to N is a block Gauss-Seidel iteration. Notice that we reduced the solution of an $n \times n$ matrix to a series of solutions on smaller matrices. At each step in the iteration, we need to solve

$$\Pi_i^{(k+1)} Q_{ii} = - \left(\sum_{j=1}^{i-1} \Pi_j^{(k+1)} Q_{ji} + \sum_{j=i+1}^{N} \Pi_j^{(k)} Q_{ji} \right) \tag{2}$$

for $\Pi_i^{(k+1)}$. In general, the solution for π takes fewer iterations if we use BGS rather than Gauss-Seidel, but each BGS iteration requires more work.

It is often more efficient to solve (2) only approximately at each step, especially for the first several iterations. Stewart [13] suggests doing a small, fixed number of Gauss-Seidel (GS) iterations in order to solve (2) approximately. We perform a small number of Gauss-Seidel iterations in our tool, but let that number vary dynamically based on the progress of block retrieval from the disk. We call the number of Gauss-Seidel iterations used to solve (2) approximately the number of *inner* iterations.

Block Gauss-Seidel has many of the properties that we desire when using a disk to store the matrix. Access to Q is predictable, and therefore we can schedule the disk to read data in advance. Also, we can overlap reading in portions (block columns) of Q while performing approximate solutions to (2), so the two can proceed in parallel. Most importantly, elements can be read into memory once and used multiple times during the inner iteration. Finally, we can tune the number of inner iterations we use to solve (2) so that the iterative solution method accesses the iteration matrix at the same rate that the disk delivers data, maximizing the utilization of the CPU and disk.

4 BGS Solver

In this section, we discuss the architecture of our first implementation. In particular, we discuss the basic block Gauss-Seidel (BGS) algorithm and how it maps onto a program or set of programs that run on a workstation. An important issue we solve is how to effectively do computation and disk I/O in parallel. We develop a flexible implementation with several tunable parameters that can vary widely for optimum performance for a particular computer.

The sequential algorithm for a single BGS iteration follows directly from Section 3. In particular, let $r \in \mathcal{R}^n$ be an auxiliary variable.

```
for $i = 1$ to $N$
    $r = 0$
    for $j = 1$ to $N | j \neq i$
        $r = r - \Pi_j Q_{ji}$
    Solve $\Pi_i Q_{ii} = r$ for $\Pi_i$
```

One can easily see that the access to Q is very predictable, so we may have blocks of Q ordered on disk in the same way that the program accesses them. This way the program accesses the file containing Q sequentially entirely throughout an iteration. One could therefore easily write a utility to write Q appropriately to a file and an implementation of BGS. What is not trivial is to build a tool that overlaps computation (the solution of $\Pi_i Q_{ii} = r$) and reading from disk in a flexible, efficient way.

4.1 Tool Architecture

Our solution to this is to have two cooperating processes, one of which schedules disk I/O, and the other of which does computation. Obviously, they must communicate and synchronize activity. We use System V interprocess communication mechanisms since they are widely available and simple to use. We use semaphores for synchronization and shared memory for passing data. We call the process that schedules I/O the *I/O process*, and we call the process that solves $\Pi_i Q_{ii} = r$ the *compute process*. To minimize memory usage, we want to have as few blocks of Q in memory at one time as possible, so we must be careful how we compute the step $r = r - \Pi_j Q_{ji}$, $\forall j \neq i$. For simplicity, we assign the task of computing r to the I/O process.

We first looked at several large matrices that were generated by GSPN models. (We looked at GSPNs because they can easily create large transition rate matrices, not because our solution technique is limited to them.) We noticed that the matrix is usually very banded; that is, for reasonable choices for N, the number of non-zero elements in the blocks $Q_{i,j} : |i - j| > 1$ is small, if not zero. By lumping all the blocks in a column into a smaller number (three) of larger blocks, we can eliminate the overhead of reading small or empty blocks. For the i-th column, we call $Q_{i,i}$ the *diagonal* block; $Q_{i-1,i}$ is the *conflict* block; and all other blocks are lumped into a single block that we call the *off* block. We use the term *off block* because it includes all the off diagonal blocks except the conflict block. Let D represent the diagonal block, C the conflict block, and O the off block. The following represents a matrix where $N = 4$.

$$\left(\begin{array}{c|c|c|c} \mathrm{D} & \multicolumn{2}{c|}{\mathrm{O}} & \mathrm{C} \\ \hline \mathrm{C} & \mathrm{D} & \multicolumn{2}{c}{\mathrm{O}} \\ \hline \mathrm{O} & \mathrm{C} & \mathrm{D} & \mathrm{O} \\ \hline \multicolumn{2}{c|}{\mathrm{O}} & \mathrm{C} & \mathrm{D} \end{array}\right)^T$$

The reason we have a conflict block will be apparent soon.

Lumping several blocks into the off block complicates (2), but does not require any extra computation. The actual mechanics of the computation of $\Pi Q_{\mathrm{off},i}$ are the same as the computation of $\Pi_j Q_{ji}$. For the formula $r = \Pi Q_{\mathrm{off},i}$, we compute $r = \sum_{k \neq i, i-1} \Pi_k Q_{ki}$. We may now compute r the following way:

```
Shared variables: $\pi$, $Q_{\text{diag0}}$, $Q_{\text{diag1}}$, $r_0$, $r_1$
Semaphores: $S_1$ locked, $S_2$ unlocked
```

```
Compute Process
Local variable (unshared): $t$
$t = 0$
while not converged
  for $i = 1$ to $N$
    Lock($S_1$)
    for $j = 1$ to $MinIter$
      Do GS iteration: $\Pi_i Q_{\text{diag}t} = r_t$
    $j = MinIter + 1$
    while $j \leq MaxIter$ and
        I/O process not blocked on $S_2$
      Do GS iteration: $\Pi_i Q_{\text{diag}t} = r_t$
      j = j + 1
    Unlock($S_2$)
    $t = \bar{t}$
```

```
I/O Process
Local variable (unshared): $t$, $Q_{\text{tmp}}$
$t = 0$
do forever
  for $i = 1$ to $N$
    $Q_{\text{diag}t}$ = disk read($Q_{ii}$)
    $Q_{\text{tmp}}$ = disk read($Q_{\text{off},i}$)
    $r_t = -\Pi Q_{\text{tmp}}$
    $Q_{\text{tmp}}$ = disk read($Q_{\text{conflict},i}$)
    Lock($S_2$)
    $r_t = r_t - \Pi_{i-1} Q_{\text{tmp}}$
    Unlock($S_1$)
    $t = \bar{t}$
```

Fig. 2. Compute and I/O processes for BGS algorithm.

$$\begin{aligned} r &= -\Pi Q_{\text{off},i} \\ r &= r - \Pi_{i-1} Q_{\text{conflict},i} \end{aligned}$$

Let us denote $r_i = \Pi_i Q_{ii}$ to distinguish between different r vectors. In order to make the computation and disk I/O in parallel, the program must solve $\Pi_i Q_{ii} = r_i$ while, at the same time, computing r_{i+1}. Therefore, while the compute process is solving $\Pi_i Q_{ii} = r_i$, the I/O process is prefetching $Q_{i+1,i+1}$, and reading $Q_{\text{off},i+1}$ and $Q_{\text{conflict},i+1}$ to compute r_{i+1}. Notice that when computing r_{i+1}, we need the most recent value of Π_i to multiply by $Q_{\text{conflict},i}$, which introduces a data dependency. Thus, we can not completely compute r_{i+1} while in parallel computing Π_i. (We could also use a less recent value of Π_i, but that would reduce the effectiveness of BGS.)

Finally, we add synchronization to ensure that the I/O process has the most recent version of Π_i to compute r_{i+1}. The full algorithm we use is presented in Figure 2. We used a large, shared memory array to represent the steady state probability vector Π, two shared diagonal block buffers Q_{diag0} and Q_{diag1}, and two r vectors r_0 and r_1. The processes share two diagonal block and r variables so that one can be used to compute (2) while the other one is being prepared for the next computation. The processes also share two locking variables, S_1 and S_2, which they use to communicate and control the relative progress of the other process.

4.2 Compute Process

We first explain the compute process. A local variable t alternates between 0 and 1, which indicates which of the two shared block and r variables the process should use. After each step, t is alternated between 0 and 1, which we denote $t = \bar{t}$. The function `Lock(`S_1`)` will lock S_1 if S_1 is unlocked. If S_1 is already locked, it will block until S_1 is unlocked (by the I/O process); then it will lock S_1 and proceed. While the compute process is blocked on S_1, it uses no CPU resources.

The compute process has two parameters, *MinIter* and *MaxIter*. The compute process is guaranteed to do at least *MinIter* Gauss-Seidel inner iterations to approximately solve (2). Then the compute process will proceed to do up to *MaxIter* iterations or until the I/O process is complete with the current file I/O and is waiting for the compute process to unlock S_2, whichever comes first. This allows the compute process to do a dynamic number of Gauss-Seidel iterations, depending on how long the I/O process takes to do file I/O. We ignore the boundary conditions in the figures for simplicity. If $i - 1 = 0$, for example, then we use N for $i - 1$ instead.

The convergence criterion we use in this tool is a modification to the $||\pi^{(k+1)} - \pi^{(k)}||_\infty$ criterion. In particular, we compute $||\Pi_i^{(k+1)} - \Pi_i^{(k)}||_\infty$ for the *first* inner iteration and take the $\max_i ||\Pi_i^{(k+1)} - \Pi_i^{(k)}||_\infty$ to be the number we use to test for convergence. We use this for two reasons: the first inner iteration usually results in the greatest change of Π_i, so computing the norm for all inner iterations is usually wasteful; and the computation of the norm takes a significant amount of time. We have observed experimentally that this measured norm is at least as good as the commonly used $||\pi^{(k+1)} - \pi^{(k)}||_\infty$ criterion.

The dynamic nature of the iteration count is an interesting feature of this solver. If the system on which the program is running is doing other file I/O and slowing the I/O process down, the compute process may continue to proceed to do useful work. At some point, however, additional Gauss-Seidel iterations may not be useful at all, presumably after *MaxIter* inner iterations, so the process will stop doing work and block waiting for S_1 to become unlocked. Choosing a good *MinIter* and *MaxIter* is difficult and requires some knowledge about the characteristics of the transition rate matrix. However, if we allow the compute process to be completely dynamic, some blocks may consistently get fewer inner iterations and converge more slowly than other blocks, causing the whole system to converge slowly. In Section 5, we show some experimental results of varying these parameters.

4.3 Input/Output Process

The I/O process is straightforward. The greatest complexity comes in managing the semaphores properly. This is a case of the classical producer-consumer or bounded buffer problem, and we refer the reader to [14] or a similar text on operating systems to show the motivation and correctness of this technique. The primary purpose of the I/O process is to schedule disk reads and compute r_t, where t alternates between 0 and 1. It does this by issuing a C function to read portions of the file directly into the shared block variable or the temporary block variable. Because the I/O process may execute in parallel with the compute process, the I/O process may issue read requests concurrently with the computation, and since file I/O uses little CPU (under 20%), we can effectively parallelize computation and file I/O on a modern, single-processor workstation.

This implementation of BGS uses relatively little memory. The size of the steady state probability vector π is proportional to the number of states, which is unavoidable using BGS or any other exact method. Other iterative methods, such as Jacobi, require additional vectors of the same size as π, which our program avoids. Two diagonal blocks, Q_{diag0} and Q_{diag1}, are necessary; the compute process requires one block to do the inner iteration, and the I/O process reads the next diagonal block at the same time. Two r variables are also required for the same reason. Finally, the I/O process requires a temporary variable Q_{tmp} to hold Q_{off} and Q_{conflict}. We could eliminate Q_{tmp} by instead using $Q_{\text{diag}t}$, but doing so would require us to reverse the order in which we read the blocks, causing us to read $Q_{\text{diag}t}$ last. This would reduce the amount of time we could overlap computation and file I/O. We chose to maximize parallelization of computation at the expense of a modest amount of memory.

5 Results of First Implementation

To better understand the algorithms presented in the previous section, we implemented them and tested the resulting tool on two large models presented in the literature. We present the models for the purpose of taking performance measurements on the solver, so we are not so concerned about the details or results of the model.

N	States	NZ Entries	Size (MB)	e_1	e_2	e_3	e_4	τ
1	160	616	0.008	0.90742	0.67136	0.67136	0.35538	0.09258
2	4,600	28,128	0.34	1.81006	1.32851	1.32851	0.76426	0.17387
3	58,400	446,400	5.3	2.72211	1.94348	1.94348	1.52460	0.23307
4	454,475	3,979,850	47	3.64641	2.51298	2.51298	1.50325	0.27589
5	2,546,432	24,460,416	290	4.58301	3.03523	3.03523	1.81096	0.30712
6	11,261,376	115,708,992	1,367	5.53098	3.50975	3.50975	2.07460	0.33010

Table 1
Characteristics and reward variables for the Kanban model.

5.1 Kanban Model

The Kanban model we present was previously used by Ciardo and Tilgner [2,3] to illustrate a Kronecker-based approach. They chose this model because it has many of the characteristics that are ideal for superposed GSPN solution techniques, and it also does not require any mapping from the product space to the tangible reachable states. We refer to [3] for a description of the model and specification of the rates in the model. Briefly, the model is composed of four subnets. At each subnet, a token enters, spends some time, and exits or restarts with certain probabilities. The manufacture of a product involves use of the first subnet, then the second and third subnet in parallel, and then the fourth subnet. We chose to solve the model in which the synchronizing transitions are timed.

Table 1 shows some information about the model and the corresponding transition rate matrix. Here, N represents the maximum number of tokens that may be in a subnet at one time. There are two important variables that we may vary, namely the number of blocks and the number of inner iterations, that greatly affect performance. We present two experiments. First, we vary the number of blocks while keeping the number of inner iterations fixed, and second, we vary the number of inner iterations while keeping the number of blocks fixed. All solutions of the Kanban model were solved on an HP C-110 workstation with 128 MB of RAM (without using virtual memory) and 4 GB of fast disk memory.

For the first experiment, we use the Kanban model where $N = 5$. We divide the transition rate matrix into 32×3 blocks, and perform a constant number of inner iterations. We vary the number of inner iterations from 1 to 20. The results of the solution execution time and the number of BGS iterations are shown in the top two graphs in Figure 3. All the timing measurements that we present in this paper are "wall clock" times. The plots show the time to achieve three levels of accuracy based on the modified $||\Pi^{(k+1)} - \Pi^{(k)}||_\infty < \{10^{-6}, 10^{-9}, 10^{-12}\}$ convergence criterion explained in Section 4.

Figure 3 shows how doing an increased number of inner iterations yields diminishing returns, so that doing more than about 7 inner iterations does not significantly help reduce the number of BGS iterations. For this model, setting *MaxIter* to 6 or 7 makes sense. The figure also shows that the optimal number of inner iterations with respect to execution time is 4. For fewer than four inner iterations, the compute process spends time idle and waiting for the I/O process. This leads us to choose *MinIter* to be 3 or 4.

It is interesting to note that solving this model with a dynamic number of inner iterations takes 10,436 seconds, which is more time than is required if we fix the number of inner iterations to be 3, 4, or 5 (10269, 10044, and 10252 seconds respectively). We observed that some blocks always receive 4 or fewer inner iterations, while others always receive 7 or more. This shows us several important things. First, some blocks always receive more iterations than others, and we know that the solution vector will converge only as fast as its slowest converging component. Second, we argued above that doing more than 7 inner iterations is wasteful, so allowing the number of inner iterations to be fully dynamic is wasteful, since the I/O process does not read data quickly enough to keep the compute node doing useful work. Finally, if the compute process is always doing inner iterations, it checks to see if the I/O process is blocked on S_2 only after completing an inner iteration. This requires the I/O process to always block on S_2 and wait for the compute process to complete its inner iteration, which is wasteful since the I/O process is the slower of the two processes.

For the next experiment, we set the number of inner iterations to be 5, vary the number of blocks, and observe convergence rate, execution time, and memory usage. The bottom two plots of Figure 3 range the number of blocks from 8 to 64 and plot execution time and number of iterations respectively for the convergence criteria $||\Pi^{(k+1)} - \Pi^{(k)}||_\infty < \{10^{-6}, 10^{-9}, 10^{-12}\}$. Figure 4 shows how memory usage varies with the number of blocks. Notice that in a comparison between 8 and 64 blocks, the execution time is nearly double while the memory usage is about one third. We see that there is clearly a memory/speed tradeoff. Note that the solution vector for a 2.5 million state model alone takes about 20 megabytes of memory.

Finally, as a basis for comparison, we present results given in [3] in Figure 5 and compare the solution times to those of our tool. The column entitled ‘Case 1’ represents the tool in [3] with mapping from the product space to tangible reachable states enabled, while ‘Case 2’ is with no mapping (an idealized case). Cases 1 and 2 are performed on a Sony NWS-5000 workstation with 90 MB of memory. We present no results for $N = 1, 2, 3$ because the matrix was so small that the operating system buffered the entire file in memory, yielding uncharacteristically fast performance. Times for both solvers are for the same stopping criteria of 10^{-6}.

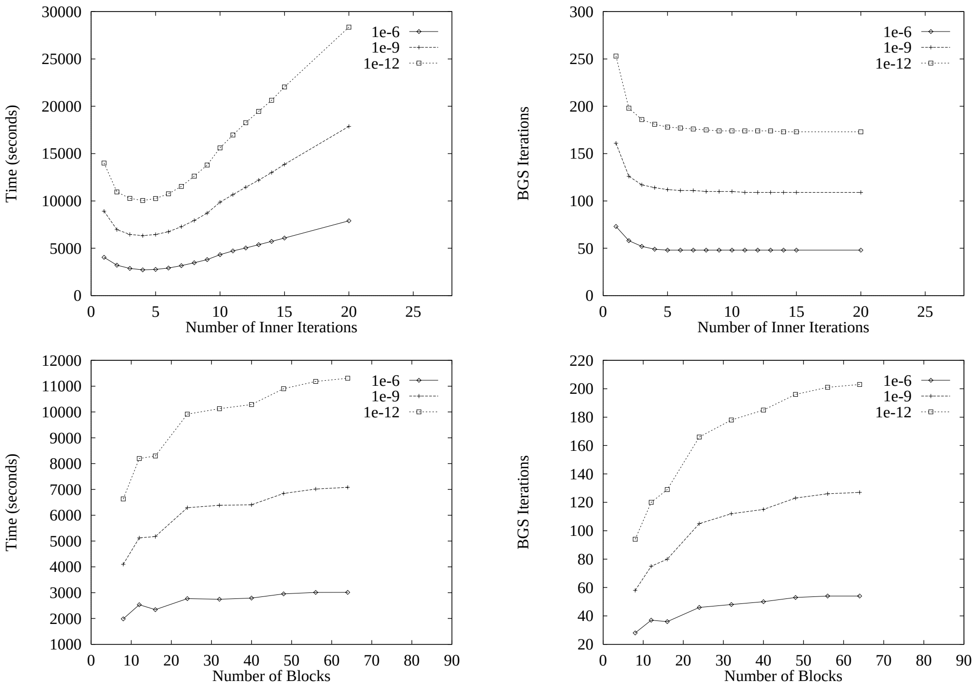

Fig. 3. Performance graphs of Kanban model ($N = 5$).

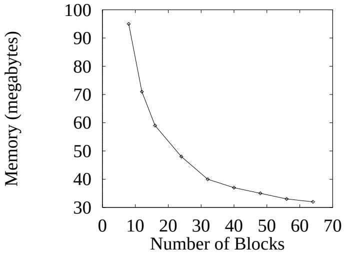

Fig. 4. Number of blocks versus memory.

	Kronecker		BGS		
N	Case 1	Case 2	BGS time	Blocks	Memory
1	1 s	1s	-	-	-
2	13 s	2 s	-	-	-
3	310 s	2 s	-	-	-
4	4,721 s	856 s	225 s	4	28 MB
5	22,215 s	6,055 s	2,342 s	16	59 MB
6	-	-	18,563 s	128	111 MB

Fig. 5. Comparison of performance.

5.2 Courier Protocol Model

The second model we examine is a model of the Courier protocol given in [9,15]. This is a model of an existing software network protocol stack that has been implemented on a Sun workstation and on a VME bus-based multiprocessor. The model is specified in [9,15]. The GSPN models only a one-way data flow through the network. For our experiment, we are only interested in varying the window size N. The transport space and fragmentation ratio is kept at one. All solutions to the Courier Protocol model were performed on an identically configured HP C-160 (only the CPU was upgraded).

Table 2 shows the characteristics of the model. The column 'Matrix Size' contains the size of the matrix in megabytes if the matrix were to be kept

	$N = 1$	$N = 2$	$N = 3$	$N = 4$	$N = 5$	$N = 6$
States	11,700	84,600	419,400	1,632,600	5,358,600	15,410,250
Nonzero Entries	48,330	410,160	2,281,620	9,732,330	34,424,280	105,345,900
Matrix	0.6	5	28	118	414	1,264
Blocks	4	4	4	32	64	128
Generation Time (s)	3	27	166	762	3,166	10,700
Conversion Time (s)	2	16	86	373	1,384	4,436
Solution Time (s)	3	14	91	1,274	6,560	26,143
Iterations	18	19	24	46	69	85
Memory (MB)	0.4	3.4	18	21	57	144

Table 2
Characteristics of BGS for Courier protocol model.

entirely in memory. One can see that this transition-rate matrix is less dense than the one for the Kanban model. For this model, we wish to show how the solution process varies as the size of the problem gets larger. We allow the number of inner iterations to be fully dynamic (the number ranged between 3 and 8) and let N range. Table 2 summarizes these results.

There are several interesting results from this study. First, we note that for $N < 3$, the file system buffers significantly decrease the conversion and solution times, so they should not be considered as part of a trend. More traditional techniques would probably do as well or better for such small models. For $N \geq 3$, the model becomes interesting. We wrote our own GSPN state generator for these models, and it was optimized for memory (so we could generate large models), not for speed. It was also designed to be compatible with the *UltraSAN* [12] solvers and reward variable specification. The conversion time is the time it took to convert the Q-matrix in *UltraSAN*'s format to one used by the tool, which includes taking the transpose of Q and converting from an ASCII to binary floating point representation.

The data gives us a rough idea about the relative performance of each step in the solution process. The conversion process takes between half and one third the generation time. We believe that much of the conversion time is spent translating an ASCII representation of a real number into the computer's internal representation. The solution times are the times to reach the convergence criterion $||\Pi_i^{(k+1)} - \Pi_i^{(k)}||_\infty < 10^{-12}$ described above, and are roughly twice the generation times. This shows that the solution process does not take a disproportionate amount of time more than the state generation or conversion process.

Another interesting observation of this model is that in the case where $N = 6$, the transition-rate matrix generated by the GSPN state generator (a sparse textual representation of the matrix) would be larger than 2 GB, which is larger than the maximum allowable file size on our workstation. To solve this system, we rounded the rates to 6 decimal places. There are other simple

	$N = 1$	$N = 2$	$N = 3$	$N = 4$	$N = 5$	$N = 6$
λ	74.3467	120.372	150.794	172.011	187.413	198.919
P_{send}	0.01011	0.01637	0.02051	0.02334	0.02549	0.02705
P_{recv}	0.98141	0.96991	0.96230	0.95700	0.95315	0.95027
P_{sess1}	0.00848	0.01372	0.01719	0.01961	0.02137	0.02268
P_{sess2}	0.92610	0.88029	0.84998	0.82883	0.81345	0.80197
$P_{transp1}$	0.78558	0.65285	0.56511	0.50392	0.45950	0.42632
$P_{transp2}$	0.78871	0.65790	0.57138	0.51084	0.46673	0.43365

Table 3
Reward variables for Courier protocol model.

solutions to this problem; we simply state this observation to give the reader a feel for the size of the data that the tool is manipulating. Also, of the 144 MB necessary to compute the solution, 118 MB are needed just to hold the solution vector.

In Table 3 we show several of the reward variables in the model as N varies from 1 to 6. The λ we compute here corresponds to measuring λ_{lsp} in the model, which corresponds to the user's message throughput rate. The measures λ_{frg} can easily be computed as $\lambda_{frg} = \lambda q_1/q_2$. Similarly, $\lambda_{ack} = \lambda_{lsp} + \lambda_{frg}$. From this, we can see how the packet throughput rate (λ) increases as the window size increases. Other reward variables are explained in [9], and they correspond to the fraction of time different parts of the system are busy. We note that the values we computed here differ from those Li found by approximate techniques [9,15]. We suspect that Li used a fragmentation ratio in his approximation techniques that is different (and unpublished) from the ratio for which he gives "exact" solutions, because we were able to reproduce the exact solutions.

Although we do not show any direct comparison between the execution of the solver on the two different CPUs, we experienced little significant speedup in the solution time with the upgrade. This is because the solver is limited by the disk I/O rate. We address this bottleneck in the next section.

6 CBGS Solver

We view the first implementation as a great success as a tool for solving large Markov processes generated from stochastic Petri nets without imposing any restrictions on the model. In analyzing the bottleneck in the performance of the tool, we see that the disk I/O speed is the constraining factor. This is somewhat distressing, because we anticipate that processor speeds will increase at a much faster rate than disk speeds.

A natural approach to this discrepancy is to trade disk speed for processor time, i.e., file compression. The remainder of this section describes the compression technique we implemented and shows results of the new implementation on a faster computer.

6.1 Compression Technique

The file format for the second implementation was simplified for building a prototype quickly and for making compression simpler. The matrix is divided into block columns in the CBGS solver, with no distinction between diagonal, off, or conflict blocks. (All three are now included in one block.)

The data format we used is similar to (but different from) Compressed Column Storage [1, p. 65] for each block column. The matrix is essentially represented as three arrays. The first, `col_ptr`, is a pointer to the place in the other two arrays for the first nonzero element in a particular column. The `row_ind` array stores the distance from the diagonal of a particular nonzero entry. The `val` array stores an index into a table for the value of the nonzero entry. The table is stored in the header. The compression technique we used is simply to use only the necessary number of bits to store values in each of the three arrays. If there is a maximum of 4000 states in a block column, for example, then only 12 bits are used for each `col_ptr` entry. Admittedly, this compression technique works well only when there are a small number of distinct non-zero entries. This is the case in most models we have seen.

6.2 Solver Algorithm

The file compression requires a change in the algorithm used by the solver. In some ways, the algorithm is actually simpler. This results in some compromises, for which improvements are suggested in Section 6.4.

We keep the paradigm of using a compute and an I/O process, but now the I/O process simply reads blocks of data from the file into shared buffers. The compute process now has the additional tasks of decompressing the block and computing r. The decompression algorithm is a straightforward use of bit masks and shifting. The way the two processes communicate and share data is the same as before. The detailed algorithms are shown in Figure 6. Notice that the form looks very similar to Figure 2. What is different is that Q_0 and Q_1 now hold compressed block rows instead of decompressed blocks, and the I/O process does only disk I/O.

We believe the new algorithm is superior to the previous one because it addresses all of the limiting constraints and increases flexibility. The I/O process

Shared variables: Q_0, Q_1
Semaphores: S_1 locked, S_2 unlocked

```
Compute Process
Local variable (unshared): $\pi$, $t$, $r$
$t = 0$
while not converged
  for $i = 1$ to $N$
    Lock($S_1$)
    decompress $Q_t$ into $Q_{\text{local}}$ and $r$
    Unlock($S_2$)
    for $j = 1$ to $MinIter$
      Do GS iteration: $\Pi_i Q_{\text{local}} = r$
    $j = MinIter + 1$
    while $j \leq MaxIter$ and
        I/O process not blocked on $S_2$
      Do GS iteration: $\Pi_i Q_{\text{local}} = r$
      j = j + 1
    $t = \bar{t}$
```

```
I/O Process
$t = 0$
do forever
  for $i = 1$ to $N$
    $Q_{\text{t}}$ = disk read block $i$
    Lock($S_2$)
    Unlock($S_1$)
    $t = \bar{t}$
```

Fig. 6. Compute and I/O processes for CBGS algorithm.

is now free to read up to two blocks ahead of the compute process. As we discovered experimentally, that is unnecessary, but it does ensure that the I/O process is never unnecessarily blocked. Also, since the compute process computes r while decompressing the block column, there is no need to make a conflict block. This also frees the compute process to normalize π should $||\pi||_1$ drift far from 1. (The current implementation normalizes if $||\pi||_1$ is less than 0.1 or greater than 10.)

6.3 Results

All results for this section were performed on the same HP C-160, but with a larger and slightly faster hard disk. The old system had a maximum file transfer rate of about 6 MB/s, while the new system has a transfer rate of about 8 MB/s. We compare the two implementations on nearly identical machines, and we acknowledge that the new implementation is using a faster hard disk. However, we see that for most solutions the average disk transfer rate is around 6 MB/s.

For the first comparison, we compare solutions of the Kanban model (N=5), varying the number of inner iterations. We compare CBGS in Figure 7 with BGS in Figure 3. Notice that the time is faster than the first implementation by a factor of approximately three to four. Although a faster computer may

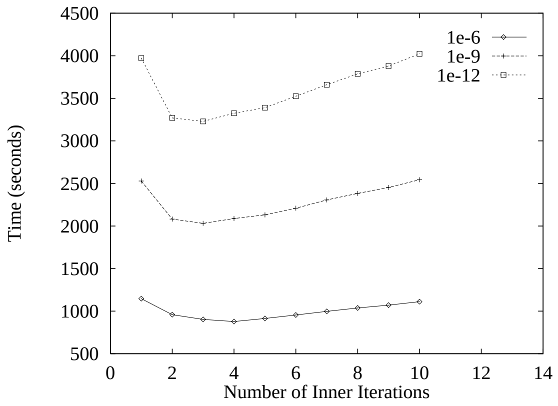

Fig. 7. CBGS Solution to Kanban model (N=5).

	$N = 1$	$N = 2$	$N = 3$	$N = 4$	$N = 5$	$N = 6$
States	11,700	84,600	419,400	1,632,600	5,358,600	15,410,250
Nonzero Entries	48,330	410,160	2,281,620	9,732,330	34,424,280	105,345,900
Matrix (MB)	0.6	5	28	118	414	1,264
File size (MB)	0.2	1.7	10.7	48	174	569
Blocks	4	4	4	32	64	128
Generation Time (s)	3	27	168	762	3,166	10,700
Conversion Time (s)	3	22	164	769	3,066	9667
Solution Time (s)	1	8	51	476	2,068	7,863
Iterations	13	17	20	49	69	85
Memory (MB)	0.5	4.4	25	24	62	151

Table 4
Characteristics of CBGS on Courier protocol model.

account for some of this improved performance, it certainly does not account for all of it.

We observed experimentally that the I/O process can proceed much faster than the compute process; the I/O process can read in one block faster than the compute process can decompress it and do one inner iteration. Consequently, the number of inner iterations is always determined by *MinIter*, and the CBGS solver has no observed dynamic behavior in the number of inner iterations.

Next, we compare performance using the model of the courier protocol with the number of inner iterations fixed at four. Table 4 contrasts with Table 2. Naturally, the information about the Markov chain is the same. The entry for “File size” shows the size of the compressed matrix file. Notice that the file sizes are between two and three times smaller than the matrix size (given in the “Matrix” row). The compression ratio was not as good as we observed on the Kanban model, and we discuss ways of improving this in Section 6.4.

The state space generation times (“Generation Time” row) are, of course, identical, since they were generated on nearly identical machines. The conversion times, however, are done differently, and this converter must also do file compression. The solution times are approximately three times faster! The only difference between the two machines is a slightly faster hard disk, but since the maximum (average) transfer rate of the CBGS solver is 6.4 MB/s, this can account for only a small fraction of the improved performance. Finally, CBGS takes more memory than BGS, unnecessarily, and methods for improving this are discussed in Section 6.4.

6.4 Profile and Suggested Improvements

To get a better understanding of how well CBGS works, we did some profiling. Approximately 15-20% of the CPU time was spent doing file I/O, with the remaining time spent in the compute process. Within the compute process doing four inner iterations, slightly more than half the time is spent decompressing the block, with the remaining half spent doing inner iterations. Within the decompression routine, slightly more than half of the time is spent decompressing the data, while the rest is spent arranging the matrix properly and computing r.

Simple calculations show that the I/O process was performing file I/O at an average rate of 6.4 MB/s, which is roughly equivalent to the rate for the BGS solver, so the faster disk had virtually no impact on the solution time. An unmistakable conclusion, however, is that the BGS solver is limited by the speed of the hard disk, and compression, when available, offers a substantial improvement in performance. This is what leads us to believe that future disk-based solvers must implement file compression to be efficient.

We were somewhat disappointed with the results of the file compression for the Courier protocol model. One reason for this is that the CBGS solver must store the diagonal entry as well as the off-diagonal non-zero entries. The number of distinct off-diagonal non-zero entries is small (on the order of tens), but the number of distinct diagonal entries is large (several thousand). To achieve better compression, it is worth creating two tables: one for the diagonal elements and one for the off-diagonal elements. This could result in a substantial reduction in file size.

We learned several things from the implementation of the CBGS solver. First, file compression is a necessity for faster computers. For this particular implementation, since the I/O process is always able to deliver data faster than the compute process can process it, having two compressed block column buffers is unnecessary; one would be sufficient. Also, the temporary block buffer used by the compute process allocates enough space to hold the whole block column, which is unnecessarily large. Only the diagonal block needs to be stored because the rest of the information is stored in r. These are the primary reasons why the CBGS implementation takes more memory than the BGS.

7 Conclusion

We have described a new tool for solving Markov models with very large state spaces. By devising a method to efficiently store the state-transition-rate matrix on disk, overlap computation and data transfer on a standard workstation, and utilize an iterative solver that exhibits locality in its use of data, we are able to build a tool that requires little more memory than the solution vector itself to obtain a solution. This method is completely general to any model for which one can derive a state-transition-rate matrix. As illustrated in the paper, the tool can solve models with 10 million states and 100 million non-zero entries on a machine with only 128 MB of main memory. Because we make use of an innovative implementation using two processes that communicate via shared memory, we are able to efficiently utilize the system disk and CPU.

We described two implementations: a disk-based block Gauss-Seidel solver, and a similar solver that uses file compression. While the CBGS solver is less general, it takes advantage of the characteristics frequently found in models generated by stochastic Petri nets to substantially increase performance. In addition, we have illustrated the use of both implementations on two large-scale models: a Kanban manufacturing system and the Courier protocol stack executing on a VME bus-based multiprocessor. For each model, we present detailed results concerning the time and space requirements for solutions so that our tool may be compared with existing and future tools. The results show that the speed of solution is much faster than those reported for implementations based on Kronecker operators or on-the-fly solvers. These results show that our approach is the current method of choice for solving large Markov models if sufficient disk space is available to hold the state-transition-rate matrix.

Acknowledgement

We would like to thank a person (whose name we do not remember) who attended the Tools '97 conference and heard our talk about the BGS solver, and first suggested to us the idea of using file compression.

References

[1] R. Barrett, M. Berry, T. F. Chan. et al, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods* (SIAM, Philadelphia, 1994).

[2] G. Ciardo, Advances in compositional approaches based on Kronecker algebra: Application to the study of manufacturing systems, in: *Third International Workshop on Performability Modeling of Computer and Communication Systems* (Bloomingdale, IL, Sept. 7–8, 1996).

[3] G. Ciardo and M. Tilgner, On the use of Kronecker operators for the solution of generalized stochastic Petri nets, ICASE Report #96-35 CR-198336, (NASA Langley Research Center, May 1996).

[4] D. D. Deavours and W. H. Sanders, 'On-the-fly' solution techniques for stochastic Petri nets and extensions, in: *Proc. Int. Workshop on Petri Nets and Performance Models (PNPM'97)* (IEEE Comp. Soc. Press, Saint Malo, France, June 3–6, 1997) 132–141.

[5] S. Donatelli, Superposed generalized stochastic Petri nets: Definition and efficient solution, in: R. Valette, ed, Application and Theory of Petri Nets 1994, Lecture Notes in Computer Science 815, *Proc. 15th Int. Conf. on Application and Theory of Petri Nets, Zaragoza, Spain* (Springer-Verlag, June 1994) 258–277.

[6] G. Horton, Adaptive Relaxation for the Steady-State Analysis of Markov Chains, ICASE Report #94-55 NASA CR-194944, (NASA Langley Research Center, June 1994).

[7] P. Kemper, Numerical analysis of superposed GSPNs, in: *Proc. Int. Workshop on Petri Nets and Performance Models (PNPM'95)* (IEEE Comp. Soc. Press, Durham, NC, Oct. 1995) 52–61.

[8] P. Kemper, Numerical Analysis of Superposed GSPNs, in: *IEEE Transactions on Software Engineering* **22** (September 1996) 615–628.

[9] Y. Li, Solution Techniques for Stochastic Petri Nets, Ph.D. Dissertation, (Department of Systems and Computer Engineering, Carleton University, Ottawa, Ontario, May 1992).

[10] J. F. Meyer, A. Movaghar, and W. H. Sanders, Stochastic activity networks: Structure, behavior, and application, in: *Proc. International Workshop on Timed Petri Nets* (Torino, Italy, July 1985) 106–115.

[11] A. Movaghar and J. F. Meyer, Performability modeling with stochastic activity networks, in: *Proc. 1984 Real-Time Systems Symp.* (Austin, TX, December 1984) 215–224.

[12] W. H. Sanders, W. D. Obal II, M. A. Qureshi, F. K. Widjanarko, The *UltraSAN* modeling environment, in: *Performance Evaluation* **24** (1995) 89–115.

[13] W. J. Stewart, *Introduction to the Numerical Solution of Markov Chains* (Princeton University Press, 1994).

[14] A. S. Tanenbaum, *Modern Operating Systems* (Prentice Hall, 1992).

[15] C. M. Woodside and Y. Li, Performance Petri Net Analysis of Communications Protocol Software by Delay-Equivalent Aggregation, in: *Proc. Fourth Int. Workshop on Petri Nets and Performance Models* (Melbourne, Australia, Dec. 2–5, 1991) 64–73.