From *Petri Nets and Performance Models*, St. Malo, France, June 3–6, 1997.

"ON-THE-FLY" SOLUTION TECHNIQUES FOR STOCHASTIC PETRI NETS AND EXTENSIONS

Daniel D. Deavours and William H. Sanders
Center for Reliable and High-Performance Computing
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
{deavours,whs}@crhc.uiuc.edu

Abstract

Use of a high-level modeling representation, such as stochastic Petri nets, frequently results in a very large state space. In this paper, we propose new methods that can tolerate such large state spaces and that do not require any special structure in the model. First, we develop methods that generate rows and columns of the state transition-rate-matrix on-the-fly, eliminating the need to explicitly store the matrix at all. Next, we introduce a new iterative solution method, called modified adaptive Gauss-Seidel, that exhibits locality in its use of data from the state transition-rate-matrix. This permits the caching of portions of the matrix, hence reducing the solution time. Finally, we develop a new memory- and computationally-efficient technique for Gauss-Seidel-based solvers that avoids the need for generating rows of A in order to solve $Ax = b$. Taken together, these new results show that one can solve very large SPN, GSPN, SRN, and SAN models without any special structure.

1 Introduction

Problems of scalability in models and the resulting state-space explosion are daunting. The traditional approach of generating a state-level model from a high-level specification, such as stochastic Petri nets, typically results in very large state spaces for practical models. Such problems are further compounded with even higher-level formalisms, such as stochastic Petri nets with tokens that have attributes. This problem is often called the "largeness problem," and is a major impediment to accurately modeling large and complex systems.

There have been numerous attempts to address the largeness problem, resulting in techniques that produce either exact or approximate results. The exact approaches tend to fall into two general categories: those that attempt to reduce the state-space size (e.g., methods based on stochastic well-formed nets [3] or reduced base model construction [17]), and those that attempt to tolerate the large state space. Several techniques that tolerate large state spaces take advantage of the fact that some components of a model (called submodels) interact in a limited way with other submodels, so that the state-transition-rate matrix of the model is a function of Kronecker operators on the state-transition-rate matrix of the submodels. Solution methods for stochastic automata networks [18] are an example of this type of method.

More recently, there has been work on superposed generalized stochastic Petri nets (SGSPNs), which are essentially independent submodels that may be joined by synchronizing on a timed transition. This class seems to be more promising as a less restrictive modeling technique. First introduced in [7], solutions for SGSPNs were restricted by the so-called product space (the product of the submodels' state spaces), which could be much larger than the set of tangible reachable states. Kemper, in [10, 11], devised a method to operate on the tangible space, rather than the product space, by providing a mapping from product space to the tangible reachable space. Ciardo and Tilgner [6] built on Kemper's work by removing some of the imposed restrictions, e.g., by allowing synchronizing transitions to be immediate.

We believe that there are three substantial restrictions with current SGSPN techniques. First, all known methods based on Kronecker operators require models to have a structure such that there are partially independent components with limited interaction between them. While Ciardo and Tilgner relax these requirements significantly, many models still do not exhibit the structure required to use these methods.

Second, the sum of the state spaces' sizes of the component models must be smaller than the size of state space of the combined model for Kronecker-based methods to be advantageous. This requires the submodels to be approximately the same size.

Third, Kronecker-based methods have generally been limited to the Power or Jacobi methods, both of which usually exhibit poor convergence behavior. This is particularly undesirable because large systems of equations tend to exhibit worse convergence characteristics than small systems. A notable exception is the work of Ciardo [4], who presents algorithms for doing a Gauss-Seidel iteration, although we are unaware of any tool that uses them.

In contrast, we develop methods in this paper that can be used with all variants of stochastic Petri nets, regardless of the structure of the model. We develop new techniques that permit the use of more general iterative methods, which often converge more quickly. We do this in three ways. First, we develop algorithms that can generate, on-the-fly, the required incoming and outgoing transition rates from a state. In particular, we give two algorithms: one for standard stochastic Petri nets (SPNs) and one for generalized stochastic Petri nets (GSPNs) [12]. We assume the reader has a basic understanding of Markov models, state space generation, and basic iterative solution techniques.

Second, since the generation of the state-transition-rate matrix on-the-fly takes significantly more time than doing an iteration with the matrix in memory, we develop a new iterative solution method that exhibits locality in its use of data from the state-transition-rate matrix. This algorithm, which we call *modified adaptive Gauss-Seidel* (MAGS), reuses generated rows and columns in the state-transition-rate matrix within an iteration in order to reduce the performance penalty incurred by their generation.

Third, any solution algorithm based on Gauss-Seidel, such as SOR, requires access to rows of A in order to solve $Ax = b$, which corresponds to accessing the incoming rates of a state in the corresponding Markov model. We describe a new approach that only needs to compute outgoing (columns), not incoming (rows), rates, at the cost of having to keep two vectors of size equal to the number of tangible reachable states: one vector for the solution and another additional vector. This approach can be used with any solution method that is based on Gauss-Seidel, such as SOR or adaptive Gauss-Seidel.

These three contributions, namely on-the-fly rate generation, MAGS, and column Gauss-Seidel, are somewhat orthogonal in that they are independent contributions that can be applied in other contexts. For example, solutions based on Kronecker operators could benefit from both MAGS and column Gauss-Seidel. However, each contribution enhances the other, and taken as a whole, they present a new solution technique that addresses all restraining aspects of computing a solution to models that are otherwise intractable.

The remainder of the paper is organized as follows. Section 2 presents algorithms for computing incoming and outgoing transition rates for SPN and GSPN models. These algorithms are the core of our on-the-fly solution methods, since they compute the needed rows and columns of the state-transition-rate matrix directly from the net representation, without requiring explicit storage of the matrix in memory or on disk. Section 3 then presents a new iterative solution algorithm that exhibits locality in its access to rows and columns of the state-transition-rate matrix. Then, Section 4 introduces a new approach that avoids the need to have incoming transition rates for Gauss-Seidel-based iterative methods, at the expense of keeping one additional vector of size equal to the number of states in the model. This technique can easily double the speed of a solution if sufficient memory is available. Finally, Section 5 presents some empirical results from a prototype implementation of the method.

2 Forward/Backward Access Algorithms

The first class of algorithms we develop makes use of both incoming and outgoing state transition rates. In this section, we show two algorithms: one for SPN models, and one for GSPN models.

Before proceeding, we will introduce some helpful notation. In particular, we will address the solution of a system of simultaneous linear equations written as $\pi Q = 0$, where π is a row vector and Q is the state-transition-rate matrix. Since we focus on numerical solution techniques, we adopt the notation $Ax = b$, or more precisely $Ax = 0$, where $A = Q^T$ and $x = \pi^T$. Here, the off-diagonal i-th row elements of Q represent outgoing rates of state i in the corresponding Markov chain, and the off-diagonal column elements of A represent the same. Similarly, the off-diagonal i-th column elements of Q and the off-diagonal i-th row elements of A represent the incoming rates to state i in the corresponding Markov chain.

To facilitate understanding our new approach, we present in Figure 1 a simple paradigm for viewing the solution process. Instead of viewing the matrix as data, we view it as a function returning the requested portion of the matrix. Hence, when the matrix is stored explicitly in memory, the function may be quite trivial and efficient in terms of computation, but costly in terms of memory consumption. In this paradigm, the superposed GSPN methods use Kronecker operators on smaller matrices and a mapping function to generate an element of A. Thus, accessing an element of A requires more computation, but (usually) less memory. Kronecker-based methods have the disadvantage of requiring a special structure in the model in order to work efficiently. In contrast, our methods act directly on the net representation to generate a row or column of A. This requires significant computation, but it will work with any model and will always take memory proportional to the size of the model.

More specifically, let s_i represent an encoding of the i-th state of the model. The encoding may be a simple concatenation of the bit encoding of the number of tokens in places, or a more sophisticated encoding suggested by

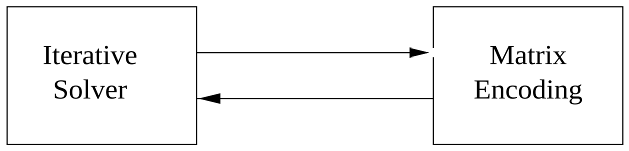

Figure 1: Solution Paradigm.

Kemper [10, 11], called a *mix*. The encodings of all the states in the model form the set $S = \{s_1, s_2, \ldots, s_n\}$ (computed initially by a state space search). To compute the i-th column of A, we take the state encoding s_i with the model and compute the successor states and the rates to those states. The significant computational requirements to compute a column in A are to

1. Decode s_i
2. Determine all enabled timed transitions in s_i
3. Fire all enabled transitions and possibly search a network of immediate transitions to determine the rate to each successor state j
4. For each successor state j:
   (a) Encode s_j
   (b) Search for s_j in S to determine index j

If we must do a binary search to look for an element in S (for the most efficient use of space), the most expensive operation is probably 4 (b), which takes time $O(\log n)$, where n is the number of states in the model. If we are willing to use more memory, we may use a hash table to do the lookup in $O(1)$ time. Since the problems we are addressing are limited by the memory of the machine, we must be careful how we use the memory.

Generating a column of A is therefore straightforward, but accessing A only by columns limits our choice of iterative methods to Jacobi or the Power method (unless the new approach from Section 4 is used). In order to use the more powerful Gauss-Seidel method, or variants of it, we need to have access to rows of A. To illustrate the need for access to rows of A, consider the basic action in the Gauss-Seidel method that we call a Gauss-Seidel *step*:

$$x_i^{(k+1)} = \frac{-1}{a_{ii}} \left(\sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} + \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} - b_i \right) \tag{1}$$

where $x^{(k)}$ is the solution vector after k iterations. Doing a Gauss-Seidel step for i from 1 to n is called an *iteration*. In order to explicitly do the summation as shown above, one must have access to a row of A. Entries in the i-th row correspond to the incoming rates from predecessor states,

```
/* Return the vector of off-diagonal row $i$ */
$a = 0$
for each $t \in T_i^{-1}$ do
    $s_j \overset{r(t)}{\longleftarrow} s_i$
    if $s_j \in S$
        $a_j = a_j + r(t)$
return $a$
```

Figure 2: Algorithm to get i-th column for SPN model.

so the task is to find the predecessor states and the corresponding incoming rates. Finding the diagonal element, a_{ii} in Equation 1, is also non-trivial, since it is defined as the negative sum of the outgoing rates. In general, to compute a_{ii} we must compute the outgoing rates and sum them. In the following, we describe how to compute the rows of A for SPN and GSPN models.

To find the off-diagonal elements of the i-th row of A, we must know the predecessor states and incoming rates. In a model, this corresponds to finding the set of states that lead to the state s_i. The approach we take to finding these predecessor states is basically to execute the model one step "backwards" in time.

2.1 SPNs

For SPNs, by which we mean Petri nets (with no inhibitor arcs) and exponentially timed transitions, the algorithm is simple. To understand it, we first introduce the notion of a reverse model. A *reverse model* is the corresponding model where the directions of all the arcs have been reversed. The firing rules are the same except that any marking-dependent rates are determined *after* a transition fires. We let T_i be the set of (timed) transitions enabled in a marking or state i, and T_i^{-1} be the set of transitions enabled in the reverse model. The notation $s_i \xrightarrow{r(t)} s_j$ means state i goes to state j with rate $r(t)$ by firing transition t. Similarly, $s_j \xleftarrow{r(t)} s_i$ means state i goes to state j with rate $r(t)$ in the reverse model, or, equivalently, state j goes to state i with rate $r(t)$ in the forward model. The symbol S denotes the reachable set of states. The algorithm for computing the non-diagonal row entries is shown in Figure 2.

To perform a Gauss-Seidel step on x_i, we need to access the i-th row of A, including the diagonal element a_{ii}. To compute the diagonal, we must also compute the i-th column vector. The necessity of computing both the i-th row and the i-th column presents an additional significant cost to computing the row vector. In Section 3 we show how we can make use of the already computed column vector in a more powerful iterative technique. In Section 4 we show how to perform Gauss-Seidel by *only* accessing A by

```
/* Return the vector of off-diagonal column $i$ */
$a = 0$
for each $t \in T_i^{-1}$ do
    $s_j \xleftarrow{r(t)} s_i$
    if $s_j \in S$
        $a_j = a_j + r(t)$
set $T^{-1} = \{\}$
call search_back_im($s_i$, 1)

procedure search_back_im($s_i$, $r$)
for each $m \in I_i^{-1}$
    $s_j \xleftarrow{w(m)} s_i$        (update $T^{-1}$)
    $\hat{r} = r \times w(m) / \sum_{\forall k | s_j \xrightarrow{\hat{w}(k)} s_k} \hat{w}(k)$
    for each $t \in T^{-1}$ do
        $s_k \xleftarrow{\bar{r}(t)} s_j$
        if $I_k = \{\}$ and $s_k \in S$
            $a_k = a_k + \hat{r}\bar{r}(t)$
    call search_back_im($s_j$, $\hat{r}$)
```

Figure 3: Algorithm to get i-th column for GSPN model.

columns.

The SPN modeling paradigm is simple, but modeling complex systems with simple SPNs is difficult. We present this algorithm because SPNs are simple and fast. This also gives a framework on which we can build more complex algorithms.

2.2 GSPNs

The procedure for computing the outgoing states and rates for a GSPN model is a straightforward extension of SPNs and is generally well known. However, it is less trivial to compute the incoming states and rates, or correspondingly, the i-th off-diagonal row elements of A. Figure 3 shows the algorithm we propose to do this. This algorithm allows for general marking-dependent rates and weights, so we can replace inhibitor arcs with transitions with marking-dependent rates or weights. The new notation is as follows: T_i is the set of transitions enabled in state s_i, T_i^{-1} is the set of transitions enabled in the reverse model in state s_i, T^{-1} is a set containing transitions enabled in the reverse model that have become enabled exclusively by the firing of some immediate transition, and $s_j \xleftarrow{w(m)} s_i$ means s_i goes to s_j by firing a single immediate transition m with weight $w(m)$ in the reverse model. I_i is the set of immediate transitions enabled in state s_i in the forward model, and I_i^{-1} is the set of immediate transitions enabled in state s_i in the reverse model.

The algorithm consists of two basic procedures that correspond to searching timed and immediate transitions. At a high level, we simply reverse the directions of the arcs and search all paths involving the firing of any number of immediate transitions followed by the firing of a timed transition. The algorithm we present does this in an organized way.

In particular, the algorithm starts by searching predecessor states reached by firing timed transitions in the reverse model. Those are the states that lead to the current state by firing only a single timed transition. After those are searched, T^{-1} is set to $\{\}$. Transitions are added to T^{-1} only as they become enabled by firing an immediate transition in the reverse model. An intuitive explanation for this is that in the forward model, a stable marking goes to a stable marking by firing a timed transition followed by a number of immediate transitions. Therefore, if we trace the same path backwards in the reverse model, the path can not end with the firing of a timed transition that does not become enabled by the firing of immediate transitions along the path. We can avoid examining many vanishing states this way and therefore prevent unnecessary computation.

The search_back_im procedure recursively searches through the network of immediate transitions. After an immediate transition is fired in the reverse model, we determine the probability $\hat{r}$ and try firing each $t \in T^{-1}$ to see if it results in a stable marking. We have found that maintaining I_i can be done efficiently and can prevent unnecessary searching in S for a vanishing marking (which is usually computationally more expensive).

Figure 3 shows the basic algorithm, but there are some possible improvements. We noted above that inhibitor arcs are a special case of marking-dependent values, which is the simplest way to deal with them. We could also build static data structures that can tell us if a transition in the reverse model is "inhibited," that is, that there is no need to fire a transition in the reverse model because it would result in a state where that transition is inhibited in the forward model.

3 Numerical Solution Methods That Exhibit Locality

Given algorithms to generate desired row/columns of A on-the-fly, we need iterative methods to solve $Ax = 0$ for the non-trivial solution of x. Typically, A is very sparse, so a vector multiplied by a row (or column) of A requires few operations. For superposed GSPN methods and the methods we present here, the time to compute a row or column of A is much greater than the time to do a vector-vector multiply, so we would like to have iterative techniques that can re-use the row (or column) of A as much as possible within a single iteration. In this section, we will present a method that has this property. Informally, the strategy is to generate a sequence of rows and columns, store them in a software cache, re-use that part of the matrix as long as it is useful, and then discard the sequence, generate a new sequence, and continue. We are willing to do more work in the solution process in return for fewer accesses to the matrix in order to speed up the overall time to compute the solution.

Adaptive Gauss-Seidel Modified adaptive Gauss-Seidel (MAGS) is an extension to adaptive Gauss-Seidel [8, 9] that exhibits locality. To motivate its formulation, we first review adaptive Gauss-Seidel. Adaptive Gauss-Seidel (AGS) is based intuitively on the observation that some elements sometimes converge or change more quickly than others, that is, $|x_i^{(k+1)} - x_i^{(k)}| > |x_j^{(k+1)} - x_j^{(k)}|$. If this is true, then a Gauss-Seidel step on x_i is considered more *effective* than a Gauss-Seidel step on x_k, and therefore more work should be done on x_i. The intuition is that x_i is getting to the solution faster, so we should do steps on it more frequently. AGS is thus a variant of Gauss-Seidel where Gauss-Seidel steps are not necessarily performed in sequential order. Adaptive Gauss-Seidel is based on the methods of Rüde [16], for which he shows rigorously the effectiveness of the algorithm for the case where A is symmetric, positive definite. Since A is not symmetric or positive definite for Markov models, we use AGS as a heuristic. Our belief in its effectiveness is based on the fact that Horton [9] shows empirically that AGS needs significantly fewer floating point operations than standard point Gauss-Seidel to solve certain Markov models to the same accuracy.

Heuristically, if we do a Gauss-Seidel step on element i and we find that $|x_i^{(k+1)} - x_i^{(k)}|$ is large, then we have done effective work on element i. Because the change in x_i is large, we should also do work on states whose occupancy probability directly depends on x_i, since they too could change significantly. These states are the successor states of state i in the corresponding Markov chain and are also the non-zero off-diagonal elements of the i-th column of A. For simplicity, we quantify effectiveness by a single number ϵ, and if $|x_i^{(k+1)} - x_i^{(k)}| > \epsilon$, then we should also do work on the successors of state i. In Section 2, we noticed that in order to compute a_{ii}, we need to compute the outgoing rates of state i. This heuristic can take advantage of this by noticing the successor states of s_i (i.e., the non-zero entries of the i-th column). Now we may begin to formulate the basis of an algorithm based on these observations.

In particular, let M be the set of states on which we need to perform work, which is initially set to S. Figure 4 shows the algorithm in detail for a given ϵ. The algorithm continues until M is empty. We call this one AGS iteration. The strategy to get a solution efficiently is to pick an initial large ϵ_0, call AGS, and then repeat the process with a successively smaller ϵ.

```
procedure AGS($\epsilon$)
$M = s_1, \ldots, s_n$
while $M \neq \{\}$
    choose state $s_i \in M$
    $M = M \setminus \{s_i\}$
    $t = x_i$
    Gauss-Seidel_Step($i$)
    if $|t - x_i| > \epsilon$
        for all $j \neq i, a_{ji} \neq 0$
            $M = M \cup \{s_j\}$
```

Figure 4: Adaptive Gauss-Seidel iteration.

The way to decrease ϵ at each iteration is a difficult problem. Horton [8, 9] proposes decreasing it by a multiplicative constant $\Delta\epsilon$, shown here.

```
$\epsilon = \epsilon_0$
while not converged
    AGS($\epsilon$)
    $\epsilon = \epsilon \times \Delta\epsilon$
```

Choosing a good $\Delta\epsilon$ is also difficult. If we choose a value near one, it makes MAGS work like normal Gauss-Seidel. If $\Delta\epsilon$ is too small, fast-changing elements may start to converge to the wrong values, resulting in unnecessary work. Horton suggests values between 0.5 and 0.1, and our experimentation shows that these values are good for our modification to AGS as well. The convergence criteria could be any of the known criteria, or it could be a sufficiently small ϵ. This seems to be at least as good as the commonly used $||x^{(k+1)} - x^{(k)}||$ method.

Modified Adaptive Gauss-Seidel Although AGS may speed convergence, since it works on states according to an "effectiveness" criterion, it does not ensure any kind of locality for data re-use. In particular, we note that AGS does not specify which state should be removed from M. We have modified the algorithm to narrow the choices in order to create locality. Specifically, we modify AGS by adding another set C, which is used to represent a software cache of M. The set C has two types associated with each element: *activated* and *deactivated*. We modify AGS by first limiting our working set to C, and when we would add s_i to M, we instead first check whether $s_i \in C$, and if it is, activate s_i; otherwise we add s_i to M. The algorithm for modified AGS (MAGS) is given in Figure 5.

In practice, the order in which we choose elements from

```
procedure MAGS($\epsilon$)
$M = s_1, \ldots, s_n$
while $M \neq \{\}$
    $C \subset M$
    $M = M \setminus C$
    while there exists an active element in $C$
        choose an active $s_i \in C$
        deactivate $s_i$ in $C$
        $t = x_i$
        call Gauss-Seidel_Step($i$)
        if $|t - x_i| > \epsilon$
            for all $j \neq i, a_{ji} \neq 0$
                if $s_j \in C$ then activate $s_j$ in $C$
                else $M = M \cup \{s_j\}$
```

Figure 5: Modified adaptive Gauss-Seidel iteration.

M or C plays a very significant role in the convergence characteristics. Experience has shown that the best convergence occurs when elements are chosen from C or M in a breadth first order. Experience has also shown that MAGS, while it is a valid implementation of AGS, does not usually perform as well as Horton's implementation of AGS. This tells us that the convergence characteristics of AGS are very dependent on the order in which elements are removed from M or C. In the worst performance, MAGS performs roughly as well as Gauss-Seidel.

4 Forward Solution Methods

The complexity in applying the above iterative solution techniques comes because they are based on Gauss-Seidel iteration steps, and hence require row access to A. One can avoid accessing rows, but this restricts solution techniques to the Jacobi or Power methods. In the two previous sections, we showed how to solve models using Gauss-Seidel-like methods by computing the predecessor states. Finding predecessor states requires more computation per iteration than finding successor states, but allows the use of iterative methods that typically converge with fewer iterations.

In this section, we show how, with a little additional work and memory requirements identical to those for the Jacobi method (one additional vector the size of a solution vector), we can also perform Gauss-Seidel-based methods, and yet require only the computation of successor states. This result is very important, since it shows that if one can use the Jacobi method, then with little additional work and no additional memory, one can use Gauss-Seidel-based iterative methods. If we have the memory to hold a second vector of size equal to the number of states, we can perform all iterative solution techniques that are based on Gauss-Seidel iteration steps without the cost of computing predecessor states. When this is done, we can compute solutions on-the-fly to more expressive modeling paradigms, such as stochastic activity networks (SANs) [13, 14] and stochastic reward networks (SRNs) [5]. Although the method works for all iterative solution techniques based on Gauss-Seidel steps, we develop it in terms of standard Gauss-Seidel first, and then show how it can be used in more sophisticated variants, such as SOR and modified adaptive Gauss-Seidel.

4.1 Column Gauss-Seidel

To understand how we can eliminate the need for row access, we recall that the basic operation in many Gauss-Seidel-based iteration schemes is the Gauss-Seidel step, given in (1). By using this step as the basic unit of computation, we can seamlessly replace Gauss-Seidel steps with the new variant, which requires only column (successor) access in other iterative methods.

We introduce our strategy with a vector δ, which we define as

$$\delta_i = x_i^{(k+1)} - x_i^{(k)},$$

so that a Gauss-Seidel step on element i is equivalent to setting $x_i^{(k+1)} = x_k^{(k)} + \delta_i$. We show how to initialize δ, and then given δ, we show how doing a Gauss-Seidel step on element i affects δ_j for all $j \neq i$.

In particular, let $x^{(1)}$ be some initial guess. We initialize δ by the following:

```
$\delta = 0$
for $i = 1$ to $n$
    for $j = 1$ to $n | j \neq i$
        $\delta_j = \delta_j + a_{ji} x_i^{(1)}$
for $i = 1$ to $n$
    $\delta_i = (b_i - \delta_i)/a_{ii} - x_i^{(1)}$
```

This essentially does a Jacobi iteration and places $x^{(2)} - x^{(1)}$ in δ. This is what we want because if we choose to start Gauss-Seidel at x_i, then $x_i^{(2)} = x_i^{(1)} - \delta_i$. (The first Gauss-Seidel step is identical to the first Jacobi step.)

Now we may do a Gauss-Seidel step on any element by simply doing the computation $x_i^{(2)} = x_i^{(1)} + \delta_i$. Once we do the computation, however, δ_j is in general obsolete. We now show how to update δ_j after each Gauss-Seidel step. Say we do a Gauss-Seidel step on x_i, in the most general form

$$x_i^{(k)} = \frac{1}{a_{ii}} \left(- \sum_{\substack{j=1 \\ j \neq i}}^{n} a_{ij} x_j^* + b_i \right) ,$$

where x_j^* is the most recently computed value of x_j. After this step, $\delta_i = 0$. Now say we do step p on x_c, and then observe the effects that this computation has on δ_i.

$$x_c^{(p+1)} = x_c^{(p)} + \delta_c$$

$$x_i^{(k+1)} - x_i^{(k)} = \frac{-1}{a_{ii}} \left(a_{ic} x_c^{(k+1)} - a_{ic} x_c^{(k)} \right)$$

Finally,

$$\delta_i = \frac{-a_{ic}\delta_c}{a_{ii}} \ .$$

Now let us not assume $\delta_i = 0$. We denote δ_i^0 as the value of δ_i before performing a Gauss-Seidel step on x_c. Inductively, we can show that after we do a Gauss-Seidel step on x_c, we can compute the new δ_i from the value of δ_i^0 and δ_c.

$$x_i^{(k+1)} - x_i^{(k)} = \delta_i^0 + \frac{-1}{a_{ii}} \left(a_{ic} x_c^{(k+1)} - a_{ic} x_c^{(k)} \right)$$

$$\delta_i = \delta_i^0 - \frac{a_{ic}\delta_c}{a_{ii}} \qquad (2)$$

Now we can see that updating δ_i after performing a Gauss-Seidel step on x_c requires access to the c-th column of A. In addition, computing δ_i also needs a_{ii}, but this dependency is easy to eliminate. If we let $d_i = \delta_i a_{ii}$, and d_i^0 is the value of d_i before performing a Gauss-Seidel step on x_c, then

$$d_i = d_i^0 - a_{ic}\delta_c \ .$$

Then, when doing the Gauss-Seidel step on x_i, simply divide d_i by a_{ii} and update all $d_j | \, a_{ji} \neq 0$. We now have everything necessary to perform a Gauss-Seidel step on x_i by accessing only the i-th column of A.

Successive Over-Relaxation We now show how to easily extend this method to Successive Over-Relaxation (SOR). Recall the basic step for SOR:

$$x_c^{(k+1)} = \omega \bar{x}_c^{(k+1)} + (1-\omega) x_c^{(k)} \ ,$$

where $\bar{x}_i$ is the Gauss-Seidel iterate. We computed above

$$\bar{x}_c^{(k+1)} = x_c^{(k)} + \delta_c \ ,$$

and by substitution,

$$x_c^{(k+1)} = x_c^{(k)} + \omega\delta_c \ .$$

The updating of δ_i, $\forall i \neq c$ is done by (2). From this, we can see that the column-only SOR step involves only a minor extension to the column-only Gauss-Seidel.

Algorithm Figure 6 shows the algorithm for doing a Gauss-Seidel step using only column access. We show the algorithm with the feature of over-relaxation parameter ω, which is set to 1 for standard Gauss-Seidel, but in general can take on values $\omega \in (0,2)$. Notice that the column Gauss-Seidel step procedure accesses only the i-th column

```
/* Matrix $A \in \mathcal{R}^{n \times n}$ */
/* arrays $x$, $b$, and $d \in \mathcal{R}^n$ */
/* Solve $Ax = b$ using $d$. */
procedure cGauss-Seidel_Step_Init()
$d = 0$
for $i = 1$ to $n$
    for $j = 1$ to $n | \, j \neq i$
        $d_j = d_j + a_{ji} x_i$
for $i = 1$ to $n$
    $d_i = (b_i - d_i / a_{ii} - x_i) a_{ii}$

procedure cGauss-Seidel_Step(int $i$)
$\delta = \omega d_i / a_{ii}$
$x_i = x_i + \delta$
for $j = 1$ to $n | \, j \neq i$
    $d_j = d_j - a_{ji} \times \delta$
```

Figure 6: Gauss-Seidel step requiring only column access to A.

of A. Consequently, we can perform any Gauss-Seidel-based step on element x_i using on-the-fly matrix generation solely by computing the successor states and corresponding rates of state s_i.

As an example of the use of this implementation of a Gauss-Seidel step, we show an implementation of standard Gauss-Seidel.

```
$x$ = initial guess
call Gauss-Seidel_Step_Init()
while $x$ not converged
    for $i = 1$ to $n$
        call cGauss-Seidel_Step($i$)
```

We call this algorithm *column Gauss-Seidel*. Notice that we can do column Gauss-Seidel with the same memory requirements as Jacobi, and after an initialization cost, the same number of operations per iteration as Jacobi and Gauss-Seidel.

Recall that the diagonal element a_{ii} is the negative sum of the off-diagonal elements in the i-th column. Performing a Jacobi iteration while accessing A by columns requires two basic steps. In the first step, we do the matrix-vector multiply $x^{(k+1)} = (A - D)x^{(k)}$, where D is the diagonal matrix of A. This step accesses the off-diagonal elements of A. The next step does $x^{(k+1)} = D^{-1}x^{(k+1)} + D^{-1}b$, which accesses the diagonal elements of A. If A is encoded, the two steps require two sweeps of A, one for the off-diagonal elements and one for the diagonal elements. The alternative is one sweep of A and explicit storage of the diagonal elements. Two sweeps of A would substantially decrease performance, and explicit storage of D requires additional memory of the same size as x. Therefore, the Jacobi method requires two sweeps of the matrix per iteration with $|A| + 2n$ memory, or one sweep per iteration with $|A| + 3n$ memory. Column Gauss-Seidel, on the other hand, only needs $|A| + 2n$ memory and a single sweep of the matrix per iteration. Thus, for on-the-fly techniques, column Gauss-Seidel takes either less work or less memory than Jacobi.

Furthermore, column Gauss-Seidel has some improved numerical properties relative to Gauss-Seidel or Jacobi. As the iteration process approaches the solution, the algorithm keeps d to full precision, even while variations in x are small. Column Gauss-Seidel may thus proceed as if x were kept to greater precision, because all the important information about x (namely $x^{(k+1)} - x^{(k)}$) is stored in d; this is useful when elements in x vary in size by many orders of magnitude and the user requires a high degree of accuracy. The algorithm is not self-correcting, however. If somehow (due to rounding errors, for example) x is perturbed, the algorithm will converge to the wrong answer. An easy solution to this is to reinitialize d when the iteration process is near the solution, or after every several digits of accuracy acquired.

4.2 Column Modified Adaptive Gauss-Seidel

As mentioned earlier, the approach used to obtain column-only Gauss-Seidel can be extended to all Gauss-Seidel-based algorithms. In this case of modified adaptive Gauss-Seidel, this is very straightforward; the routine `cGauss-Seidel_Step` is a direct replacement for the routine `Gauss_Seidel_Step`. This substitution results in an algorithm that can solve any model class, especially SAN or SRN models, with much greater speed than the variant that requires row access. The cost of using column-only Gauss-Seidel is some extra time spent in initialization (negligible) and the extra memory to hold d. If this memory is available, the column-only variant should be used.

5 Prototype Implementation Performance Comparison

We have made a prototype implementation to compare the speed of our technique to that of existing solvers based on Kronecker methods. As is typically the case with such comparisons, the prototypical nature of each implementation makes it very difficult to fairly compare performance of different methods. The use of different computing platforms and differences in the algorithms themselves make a comparison difficult. For example, Kemper has reported the time for a single Jacobi iteration for a particular model, where an iteration is defined as a sweep through the entire state space. Our new method (MAGS) uses a different notion of iteration. The algorithm does a dynamic number of basic Gauss-Seidel steps, that is determined by certain parameter values and an depending on an "effectiveness" criterion, rather than by a simple sweep through the state space. Furthermore, recently used rates are cached using our technique, and we expect that each operation in our solution technique will be more effective in leading the solver to convergence.

In spite of these difficulties, we will try to make a comparison between the Kronecker-based implementation by Kemper and our prototype on-the-fly method implementation. In doing so, we make the assumption that the time taken by the iterative solver in actually performing vector-vector multiplications is negligible relative to the cost of generating the required data for both the Kronecker and on-the-fly methods. This is reasonable, since both methods perform many more operations in obtaining the required rates than in using them. Our computer (a 120 MHz Hewlett-Packard Model C110) can perform Gauss-Seidel with over-relaxation by accessing A at a rate of 50 MB/s, for example, if the state-transition-rate matrix is stored explicitly in memory.

Since the methods have a different notion of iteration and use the obtained rates very differently (our algorithm should typically converge with fewer iterations, using a consistent notion of iteration), it is not possible to compare iteration costs. Instead, we compare data generation rates of the two implementations. To calculate the data generation rate for Kemper's implementation, we divide the time to do an iteration by the number of non-zero entries in the matrix, and we obtain a data generation rate of approximately 700 Kbytes/second on an 85 MHz Sparc 4 machine. Measurements of an SPN model on our machine show that we can generate data at a rate of about 440 KByte/second.

We guess that our machine is roughly three times faster than the one Kemper used, resulting in a data generation rate for the Kronecker-based implementation that we guess is five times faster than the on-the-fly methods. Thus (as might be expected), it takes longer to generate data for a completely general model representation than for one that exhibits the special structure needed for Kronecker-based solution methods. However, we reuse data that is generated (in a small, fixed-size cache) and use a solution algorithm that should be faster than Jacobi for most models. Thus, the solution times for the prototype on-the-fly implementation are roughly similar to those for Kronecker-based methods, and since our methods are applicable to a much wider class of models and can use more effective iterative solvers, it is reasonable to use them for models that do not exhibit the special structure necessary for the Kronecker approach.

6 Conclusion

We make three important contributions in this paper. First, we propose a technique that can solve very general models of approximately the same size as Kemper and Ciardo [6, 10, 11], using approximately the same amount of memory. Instead of requiring the special model structure needed for a Kronecker-based solution, we generate the required row and/or columns of the state-transition-rate matrix on-the-fly, allowing for a completely general structure. In particular, we develop algorithms for doing this for SPN and GSPN models.

Second, we recognized that for all such methods (both ours and Kronecker-based ones), generation of rates from the state-transition-rate matrix is much slower than the iterative solution process. To account for this, we developed a new iterative solution process, called modified adaptive Gauss-Seidel, that exhibits locality in its use of rates from the state-transition-rate matrix, and hence does not require as high a data generation rate to be competitive. Because of this, we can cache recently generated rates in memory, minimizing the bottleneck of rate generation.

Third, we showed how to solve $Ax = b$ using Gauss-Seidel and variants by accessing A only by columns. This method is no more computationally expensive than Gauss-Seidel (except for initialization, which is a very small part of the cost) and takes no more (or even less) memory than Jacobi. This result is very important, since it shows that if we have the memory to hold a second vector of size equal to the number of states in the model, (the same requirement that the Jacobi method has,) we can perform all iterative solution techniques that are based on Gauss-Seidel iteration steps without the cost of backward execution.

Finally, we compare the performance of a prototype implementation of our method to a prototype implementation using the Kronecker approach. We saw that our implementation generates data (transition rates) at a speed about five times slower than the Kronecker-based prototype did. However, we believe that the faster convergence rate of MAGS and the locality of its use of rate information, combined with column-only Gauss-Seidel, makes the on-the-fly approach viable for the solution of large models that can not be solved by the Kronecker approach.

References

[1] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia, PA, 1994.

[2] G. Chiola, C. Dutheillet, G. Franceschinis, and S. Hadaad, "On well-formed coloured nets and their symbolic reachability graph," in *Proc. Eleventh Conference on Application and Theory of Petri Nets*, Paris, France, June 1990. Reprinted in K. Jensen and G. Rozenberg, ed., *High-Level Petri Nets. Theory and Application*, Springer Verlang, 1991.

[3] G. Chiola and G. Franceschinis, "Colored GSPN models and automatic symmetry detection," in *Proc. Third Int. Workshop on Petri Nets and Performance Models (PNPM'89)*, pp. 50–60, Kyoto, Japan, Dec. 1989.

[4] 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*, pp. 61–65, Bloomingdale, IL, Sept. 7–8, 1996.

[5] G. Ciardo, A. Blakenmore, P. F. J. Chimento, J. K. Muppala, and K. S. Trivedi, "Automatic generation and analysis of Markov reward models using Stochastic Reward Nets," in C. Meyer and R. J. Plemmons, ed., *Linear Algebra, Markov Chains, and Queueing Models*, vol. 48 of *IMA Volumes in Mathematics and its Applications*, pp. 141–191, Springer-Verlag, 1993.

[6] 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.

[7] 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)*, pp. 258–277, Springer-Verlag, June 1994.

[8] G. Horton, "Adaptive relaxation for the steady-state analysis of Markov chains," in William J. Stewart, ed., *Computations with Markov Chains*, pp. 585–586, Kluwer Academic Publishers, Boston, 1995.

[9] 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.

[10] P. Kemper, "Numerical analysis of superposed GSPNs," in *Proc. Int. Workshop on Petri Nets and Performance Models (PNPM'95)*, pp. 52–61, Durham, NC, Oct. 1995.

[11] P. Kemper, "Numerical Analysis of Superposed GSPNs," in *IEEE Transactions on Software Engineering*, vol. 22, no. 9, Sept. 1996.

[12] M. A. Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franseschinis, *Modeling with generalized stochastic Petri nets*, John Wiley & Sons, 1995.

[13] J. F. Meyer, A. Movaghar, and W. H. Sanders, “Stochastic activity networks: Structure, behavior, and application,” In *Proc. International Workshop on Timed Petri Nets*, pp. 106–115, Torino, Italy, July 1985.

[14] A. Movaghar and J. F. Meyer, “Performability modeling with stochastic activity networks,” In *Proc. 1984 Real-Time Systems Symp.*, pp. 215–224, Austin, TX, Dec. 1984.

[15] M. A. Qureshi, W. H. Sanders, A. P. A. van Moorsel, and R. German, “Algorithms for the Generation of State-Level Representations of Stochastic Activity Networks with General Reward Structures,” In *Proc. Int. Workshop on Petri Nets and Performance Models (PNPM’95)*, pp. 180–190, Durnham, NC, Oct., 1995.

[16] U. Rüde, “On the multilevel adaptive iterative method,” in *Preliminary Proceedings of the Second Copper Mountain Conference on Iterative Methods*, April 9–14, 1992. Also in T. Manteuffel, ed., *SIAM J. Sci. Comput.*, 15, 1994.

[17] W. H. Sanders and J. F. Meyer, “Reduced Base Model Construction Methods for Stochastic Activity Networks,” in *IEEE Journal on Selected Areas in Communications*, vol. 9, no. 1, pp. 25–36, Jan. 1991.

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