ITERATIVE METHODS
BASED ON SPLITTINGS
FOR STOCHASTIC AUTOMATA NETWORKS

A THESIS
SUBMITTED TO THE DEPARTMENT OF COMPUTER
ENGINEERING AND INFORMATION SCIENCE
AND THE INSTITUTE OF ENGINEERING AND SCIENCE
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

By
Ertuğrul Uysal
June, 1997

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Asst. Prof. Dr. Tuğrul Dayar(Principal Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assoc. Prof. Dr. Cevdet Aykanat

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Asst. Prof. Dr. Mustafa Pınar

Approved for the Institute of Engineering and Science:

Prof. Dr. Mehmet Baray, Director of Institute of Engineering and Science

ABSTRACT

ITERATIVE METHODS
BASED ON SPLITTINGS
FOR STOCHASTIC AUTOMATA NETWORKS

M.S. in Computer Engineering and Information Science
Supervisor: Asst. Prof. Dr. Tuğrul Dayar
June, 1997

This thesis presents iterative methods based on *splittings* (Jacobi, Gauss–Seidel, Successive Over Relaxation) and their block versions for *Stochastic Automata Networks* (SANs). These methods prove to be better than the power method that has been used to solve SANs until recently. Through the help of three examples we show that the time it takes to solve a system modeled as a SAN is still substantial and it does not seem to be possible to solve systems with tens of millions of states on standard desktop workstations with the current state of technology. However, the SAN methodology enables one to solve much larger models than those could be solved by explicitly storing the global generator in the core of a target architecture especially if the generator is reasonably dense.

Keywords: Markov processes; Stochastic automata networks; Tensor algebra; Splittings; Block methods

ÖZET

Ertuğrul Uysal
Bilgisayar ve Enformatik Mühendisliği, Yüksek Lisans
Tez Yöneticisi: Yrd. Doç. Dr. Tuğrul Dayar
Haziran, 1997
RASSAL ÖZDEVİNİMLİ AĞLAR İÇİN
BÖLÜNME TABANLI
İTERATİF YÖNTEMLER

Bu tezde Rassal Özdevinimli Ağlar için bölünme tabanlı dolaylı yöntemler (Jacobi, Gauss-Seidel, Succesive Over Relaxation) ve bunların blok çeşitleri geliştirilmiştir. Bu yöntemlerin, yakın zamana kadar Rassal Özdevinimli Ağları çözmekte kullanılan power yönteminden daha iyi oldukları gösterilmiştir. Üç örnek yardımıyla, Rassal Özdevinimli Ağlar kullanılarak geliştirilmiş bir modelin çözülmesi için gerekli sürenin hala oldukça yüksek olduğunu, ve şu anki teknolojik imkanlarla, on milyonlar mertebesinde duruma (state) sahip bir modelin standart masaüstü bilgisayarlarla çözülmesinin pek mümkün gözükmediğini buluyoruz. Diğer taraftan Rassal Özdevinimli Ağlar yöntemi ile, tüm sistemi ifade eden matrisi bilgisayarın ana hafızasında seyrek şekilde saklayarak çözülebilecek modellerden çok daha büyük modellerin çözülebileceği görülmüştür. Bu durum, özellikle tüm sistemi ifade eden matrisin yoğun olduğu durumda geçerlidir.

Anahtar kelimeler: Markov süreçleri, Rassal özdevinimli ağlar, Tensör cebri, Bölünmeler, Blok yöntemler.

To my parents and my sister

ACKNOWLEDGMENTS

The most I owe to my supervisor, Asst. Prof. Dr. Tuğrul Dayar for his guidance and support during this study. I would also like to thank my family for giving more than one can imagine, and all my friends including but not limited to Ertuğrul Mescioğlu , Halit Yıldırım, Kemal Civelek and Yusuf Vefalı, meeting whom has been and is going to be a source of happiness.

Finally, I would like to thank my committee members Assoc. Prof. Dr. Cevdet Aykanat and Asst. Prof. Dr. Mustafa Pınar for their valuable comments on my thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Markov chains [16] are one of the most widely used modeling techniques in the scientific community. The range of application domains is wide, including natural sciences and engineering disciplines. The simple requirement for a system to be modeled as a Markov chain is that the system's next action (transition) depend only on the current state of the system, named as the *memoryless* or the *Markov* property [16, p. 4]. Several natural phenomena that arise in biology, physics and chemistry can be modeled as Markov chains. In engineering sciences Markov chains have a wide use in several branches of industrial engineering, electronics and computer engineering. Performance evaluation and reliability modeling is the field that Markov chains find the most use in computer engineering.

The random behavior of a system should posses a geometric or exponential probability distribution in order to be modeled as a Markov chain, since these are the only probability distributions that carry the memoryless property [16, p. 4]. Fortunately, the number of systems that show this structure is large and we have methods for fitting the random characteristics of most systems into exponential or geometric distributions.

After modeling a system as a Markov chain, one seeks quantitative information from the built model. One attractive feature of Markov chain models is that most interesting properties of a Markov model can be obtained by solving a linear system of equations. Much research result is available concerning the numerical solution of Markov chain models. In addition to this, interest in this field of research is still alive. Most methods for solving systems of linear equations may be used for Markov chain models effectively. Direct methods do not seem to be suitable for solving large and sparse systems which arise in Markov chain models. Several types of iterative methods are applied to Markov chains and their properties in the context of Markov chain models are studied. However, much research needs to be done, for understanding the behavior of iterative methods, especially non-stationary iterative methods like GMRES and Arnoldi[8].

In Markov chain applications, the problem size increases very quickly as the applications get more interesting. This problem is referred to as the *state space explosion*[16, 14, 4] problem and has initiated different approaches to the Markov chain problem. Approximate solutions and bounds for the solution vector[14] are studied for reducing the complexity of the problem. In plain words the coefficient matrix, constructed for solving the linear system of equations, becomes very large and prohibits one to solve interesting problems beyond a certain limit. Stochastic Automata Networks (SANs) [16, 10, 11] are developed to overcome the difficulties that accompany the state space explosion problem. Although it is possible to apply SAN methodology to different domains, performance modeling of parallel and distributed computer systems are especially suited to this type of approach[5]. In SAN methodology, a system is modeled as a set of components interacting with each other. Characteristics of each component is captured separately from the interactions among the components, and formulated in compact form, which leads to considerable reduction in the amount of storage needed for the model. Methods available for solving Markov chain problems obtained from a SAN formalism, appear in two forms. One might prefer to store the coefficient matrix of the linear system of equations in sparse format. However, this approach does not make use of the storage reduction provided by the SAN methodology. On the other hand, it is possible to solve the system by only referring to the compact storage of the model. Currently, the power method and non-stationary methods of GMRES and Arnoldi are implemented for SANs in compact form, but there are no results available concerning the solution of a real life problem obtained from these methods[17, 6].

In this thesis, we introduce the concept of a splitting for a SAN in compact form and develop the stationary methods of Jacobi, Gauss-Seidel and SOR based on this splitting[16, 7]. We also implement these iterative methods based on splittings and their block versions. In addition to this, we experiment with these methods on real life problems. We investigate the performance of these methods and their sparse counterparts compared to the power method.

In the following chapter, we introduce several concepts related to Markov chains and give the formulation of the problem of solving a Markov chain model as a linear system of equations. Stationary methods for Markov chain problems are also introduced in this chapter.

The third chapter discusses SANs. The concept of a SAN model in compact form is explained with an example and the necessary algebraic framework for building SAN models is also provided. This chapter ends with an algorithm and a theorem regarding the complexity of the algorithm, that proves to be useful for SAN models in compact form.

The stationary iterative methods based on splittings for solving SANs in compact form are introduced in the following chapter. The algorithms provided for the methods are explained and a section on numerical results present the performance of the methods on three problems. Some interesting properties of block methods that have gone unnoticed so far are included in this chapter. An upper bound on the number of multiplications for the Gauss-Seidel and SOR methods are derived at the end of the chapter.

The last chapter contains conclusive remarks about the methods investigated. Observations and comments about methods, the SAN methodology and Markov chains based on our work are provided in the chapter.

We included an appendix that describes how to incorporate new models into the Peps package[12], which is the software tool developed in France for solving SAN models in compact form, since we implemented our methods as an extension to Peps.

Chapter 2

Markov Chains

2.1 Preliminaries

In our attempt to understand the characteristics of natural and artificial phenomena, mathematical models of systems are developed. It is possible to build models using the concept of the system being in a number of states. Generally, the system is thought to be in an initial state, and its behavior is modeled as transitions from one state to another. It is also possible to classify systems according to certain properties they might hold. One such property that the system modeled as a process changing states might have is the *memoryless* property, i.e., it only remembers its current state [16, p. 4]. In other words, the system's transition from one state to the other is independent from the previous states that the system has visited.

In many of the models arising from diverse fields including natural sciences such as physics and biology, and engineering sciences such as industrial, electrical and computer engineering, the system either has or can be modeled as having memoryless property [16, p. 3]. The systems that posses the memoryless property may be modeled as a Markov process [16, p. 4].

A system modeled as a Markov process has a number of possible states. The actual number of possible states can be infinite, however the system can be at only one of the possible states at any time instant [16, p. 4]. In addition to this, it is assumed that the transition time, the time it takes the system to go from one state to the other is negligible. That is, the transitions are said to take place instantaneously. [16, p. 3]

It is possible to have a continuous state space for a Markov process. For instance, if the output voltage of an electric circuitry can take all values within a range, and if the system can be modeled as a Markov process, it can be modeled as a Markov process with a continuous state space, having the output voltage as states of the system. If, for instance, the circuitry's output voltage raises from 0.6 volts to 3.7 volts, one would view the model as making a transition from state 0.6 to state 3.7. On the other hand, if the output voltage can take only certain potential values, and if the system can be modeled as a Markov process, the system can be modeled as a discrete state space Markov process. Note that the actual values of the voltages do not effect the discrete or continuous character of the system.

Markov processes with discrete state spaces are called Markov chains [16, p. 5], and they are what our work is based on.

In the next section, we give the definition of a Markov chain in a formal context. The following two sections introduce two different types of Markov chains that arise in Markov chain modeling. Stationary distribution of a Markov chain[16, p. 15] is an important quantity for determining certain characteristics of the model under consideration, and is introduced in the next section. Finally, methods developed for solving Markov chain models are discussed in the last section.

2.2 Formal Definition of Markov Chains

A Markov chain is a special case of a Markov process and, a Markov process is a stochastic process satisfying certain requirements. Hence, we give the definitions of stochastic processes in general, then Markov processes and Markov chains based on this.

Definition 2.2.1 *[16, p. 4] A stochastic process is defined as a family of random variables $\{X(t), t \in T\}$ defined on a given probability space indexed by the index parameter t, where t varies over some index set (parameter space) T.*

In general, t takes values from the range $(-\infty, +\infty)$. In applications, the index set T is thought of as the set of time points at which observation about the system is made. In other words, the index t of the random variable is defined as the time point that $X(t)$ takes the observed value. In such cases, t takes values from the range $[0, +\infty)$. Depending on the characteristics of the values t takes, the process is either a continuous parameter (time) stochastic process or a discrete parameter (time) stochastic process. If t can take discrete values only, or similarly, if observation about the system is made only certain equidistant time points, the process is called a discrete–time parameter process. If the range of values of t is $[0, +\infty)$ without any restrictions, or the system is observed at time points that are not equidistant, the process is referred to as a continuous–time stochastic process.

Definition 2.2.2 *Markov property: [16, p. 4] Let $\{X(t), t \in T\}$ be a stochastic process defined on a given probability space indexed by the time index parameter t, where t varies over time index set T. Let the system be observed at time points $t_0, t_1, , \ldots, t_n$ and let $t_0 < t_1 < \ldots < t_n$. The stochastic process $\{X(t), t \in T\}$ is said to have the Markov property if and only if*

$$\begin{aligned} Prob\{X(t) \leq x | X(t_0) &= x_0, X(t_1) = x_1, \ldots, X(t_n) = x_n\} \\ &= Prob\{X(t) \leq x | X(t_n) = x_n\}. \end{aligned}$$

In plain words, Markov property states that the next transition of the system from the current state $X(t_n) = x_n$ to the next state $X(t) = x$, depends only on the current state $X(t_n) = x_n$ and is independent of its previous state history, i.e., it is independent of the states $X(t_0) = x_0, X(t_1) = x_1, \ldots, X(t_{n-1}) = x_{n-1}$. In other words, the current state of the process provides sufficient information to make the next transition.

Definition 2.2.3 *A Markov process is a stochastic process which satisfies the Markov property.*

For any stochastic process, and hence for any Markov process, the values that the random variables $X(t)$ take, define the state space of the process. As with the index set parameter t, the state space of a process can be continuous or discrete, finite or infinite.

Definition 2.2.4 *A Markov chain is a Markov process whose state space is discrete.*

In Markov chain terminology, the state space of a chain is generally associated with the set of natural numbers. In other words the states are referred as state 0, state 1, etc.

When a stochastic process possesses a certain condition on the random variables and the index parameter (namely the Markov property), the process is said to be a Markov process. Similarly, the time index set and the state space characteristics of a Markov chain give rise to several types of Markov chains.

In a *homogeneous* Markov chain, the transitions of the system are independent of the time parameter t. The Markov property requires that the next transition be independent of the *previous* state history of the process. However, it is possible that the process makes a transition which is dependent both on the current state of the system and the value of the time parameter t. Such a Markov chain, in which the transitions out of a state are dependent on the time parameter t, is called a *non-homogeneous* Markov Chain.

Similar to the state space of the process, the index set (the time parameter) can be continuous or discrete. If the time parameter of a chain takes its values from a discrete set, the Markov chain is called a Discrete Time Markov Chain (DTMC). If the values of t are are continuous, the Markov chain is called a Continuous Time Markov Chain (CTMC).

In summary, there are four parameters that describe a stochastic process. First, the continuous or discrete character of the state space, is a determining property of the process. Second, the continuous or discrete character of the time parameter introduce another classification dimension for the processes. Third, the time homogeneity of the process, is also an important quantity, in capturing the properties of a process. Finally, the characteristics of the relations between the index set and and the random variables, i.e., the dependencies among them, define classes of stochastic processes. The classification that is determined by the character of the state space is important and is discussed in more detail in the following chapters.

2.3 Discrete Time Markov Chains

If the index set of the Markov chain is countable, i.e., it is in one to one correspondence with the set of natural numbers, the Markov chain is called a Discrete Time Markov Chain. In this case, the index set is in general taken to be the set of natural numbers and the random variables are numbered accordingly, i.e., as $X_0, X_1, \ldots, X_n$.

For a non-homogeneous Discrete Time Markov chain, the Markov property is described as [16, p. 5]

$$\begin{aligned}\text{Prob}\{X_{n+1} \leq x_{n+1} | X_0 = x_0, X_1 = x_1, \ldots, X_n = x_n\} \\ = \text{Prob}\{X_{n+1} \leq x_{n+1} | X_n = x_n\}.\end{aligned}$$

The conditional probability that the process makes a transition to a new state j, given that it is in current state i, is called the *single step transition probability.* It is expressed as [16, p. 5]

$$p_{ij}(n) = \text{Prob}\{X_{n+1} = j | X_n = i\}.$$

Note that since the Markov chain is a Discrete Time Markov chain, the state indices i and j are natural numbers.

For a homogeneous Markov chain, the next transition of the process is independent of the index parameter n. The single step transition probabilities are written as

$$p_{ij} = \text{Prob}\{X_{n+1} = j | X_n = i\}.$$

The random variables X_n should be geometrically distributed in order to satisfy the Markov property. In other words there is no other discrete probability

$$\begin{bmatrix} 0 & 0.3 & 0.7 \\ 0.2 & 0.4 & 0.4 \\ 0.1 & 0.9 & 0 \end{bmatrix}$$

Figure 2.1: A transition probability matrix

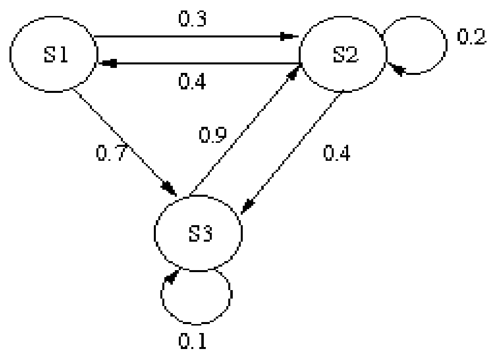

Figure 2.2: A time homogeneous discrete time Markov chain

distribution that satisfy the Markov property. A homogeneous Discrete Time Markov chain's behavior can be expressed as a *transition probability matrix.* Such a matrix is also called a chain matrix and is formed by putting the transition probability from state i to state j to the ith row and jth column of the matrix. Since the sum of the probabilities of making a transition from a state to all other states is one, the sum of the elements in any row is one. Such a matrix in which the sum of elements in any row add up to one is called a *stochastic matrix.* In Markov chain literature, the transition probability matrix is labeled P. Figure 2.1 demonstrates a transition probability matrix of a homogeneous discrete time Markov chain with three states, that is described in Figure 2.2 with transition state diagrams.

2.4 Continuous Time Markov Chains

When the time indices of a Markov chain is continuous, the chain is called a Continuous Time Markov Chain. Note that, as in the case of DTMC, the state space is still discrete. In other words, the random variables describing the process might take discrete values; however the system might be observed, i.e., make transitions, at any instant in time.

The Markov property for a non-homogeneous Continuous Time Markov Chain is expressed as [16, p. 17]

$$\begin{aligned}\text{Prob}\{X(t_{n+1}) \leq x_{n+1} \mid X(t_0) = x_0, X(t_1) = x_1, \ldots, X(t_n) = x_n\} \\ = \text{Prob}\{X(t) \leq x | X(t_n) = x_n\}\end{aligned}$$

for any sequence of time points $t_0 < t_1 < \ldots < t_n < t_{n+1}$.

The transition probability of a non-homogeneous CTMC is given by

$$p_{ij}(s,t) = \text{Prob}\{X(t) = j | X(s) = i\}, \text{ for } t > s.$$

In the homogeneous case, since the probabilities are independent of the actual values of s and t, the transition probability is expressed in terms of the difference $\tau = (t - s)$, i.e.,

$$p_{ij}(\tau) = Prob\{X(s + \tau) = j | X(s) = i\}.$$

In this case a single probability transition matrix is not sufficient to express the behavior of the matrix, we need a set of matrices parameterized by τ. Instead, a new matrix, called the *transition rate matrix* or the *infinitesimal generator matrix*, is introduced. The matrix is constructed in a similar way and is generally labeled Q. Yet, this time the entries are not probabilities of making a transition from one state to another, but each element at row i, column j of the matrix denotes an *instantaneous* transition rate. That is, the entries of the generator matrix are given by the rate of making transitions from state i to state j, when τ is chosen to be sufficiently small so that the probability of observing more than one transition within the observation period τ is negligible. A more rigorous derivation of the rate matrix from the transition probabilities can be found in [16, p. 18]. In Figure 2.4 a CTMC is shown as a state diagram. Figure 2.3 gives the corresponding infinitesimal generator matrix. Note that the diagonal entries in each row are equal to the negative of the sum of the off-diagonal entries, i.e.,

$$q_{ii} = -\sum_{j \neq i} q_{ij}.$$

$$\begin{bmatrix} -6.5 & 4.0 & 2.5 & 0 \\ 3.0 & -8.9 & 2.2 & 3.7 \\ 0 & 1.5 & -3.7 & 2.2 \\ 0 & 3.7 & 3.2 & -6.9 \end{bmatrix}$$

Figure 2.3: A transition rate matrix

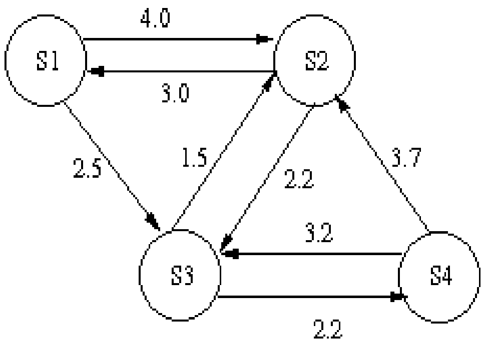

Figure 2.4: A time homogeneous continuous time Markov chain

This can easily be understood if one notices that the entries are rates representing transitions from a state to others. The transitions from one state to itself (the rate at which the process stays at that state) will decrease as the rates to the other states increase. In [16, p. 19] this property of an infinitesimal generator matrix is also derived from the transition probabilities.

For a CTMC, the random variables $X(t)$ should be exponentially distributed in order to satisfy the Markov property. Similar to the discrete case, this means that no other continuous probability distribution satisfy the memoryless property.

2.5 The Steady State Vector for a Markov Chain

The aim of modeling a system as a Markov chain, is to obtain some quantitative measures about the system. The information sought is mostly related to the states of the system. We wonder the states at which the system stays the most, how long the system occupies certain states in the long run, etc. Depending on the system being modeled, one might be interested in some states of the system more than the others. Also, in general, the states of a Markov chain are classified into several groups and determining to which group a state belongs might be of interest, see [16, p. 8]. Specifically, a *transient* state is one which the system might not return back, in the long run. A *recurrent* state is one which the system is guaranteed to return after a number of transitions. In a Markov chain, it is possible that the process makes a transition to one state, and can not leave that state, i.e., there are transitions to that state but there are not any transitions out of the state. Such states are referred to as *absorbing* states. In practice, one is more interested in states that have some desirable or undesirable properties. Thus, one might wonder the probabilities of being at those states or the average time the system spends at those states, in the long run.

It is suitable to express the state of a Markov model as a probability vector. A row vector π is used with each entry i denoting the probability of being at state i. When the system's behavior is captured as a transition rate matrix Q or a transition probability matrix P, the properties of the Markov chain can be expressed as a simple set of linear equations.

Now we introduce two important quantities that have desirable properties in the sense that they answer or provide the necessary information to answer several questions sought from a Markov chain model.

Definition 2.5.1 *Limiting Distribution of a DTMC :[16, p. 15] Given an initial probability distribution $\pi(0)$, if the limit*

$$\lim_{n\to\infty} \pi(n)$$

exists, then this limit is called the limiting distribution, and we write

$$\pi = \lim_{n\to\infty} \pi(n)$$

Definition 2.5.2 *Limiting Distribution of a CTMC :Given an initial probability distribution $\pi(0)$, if the limit*

$$\lim_{t\to\infty} \pi(t)$$

exists, then this limit is called the limiting distribution, and we write

$$\pi = \lim_{t \to \infty} \pi(t)$$

Definition 2.5.3 *Stationary Distribution of a DTMC :[16, p. 15] Let P be the transition probability matrix of a DTMC, and let the vector z whose elements z_j denote the probability of being in state j be a probability distribution; i.e.,*

$$z_j \in \mathcal{R}, 0 \leq z_j \leq 1, \ \ and \sum_{all\ j} z_j = 1.$$

Then z is said to be a stationary distribution if and only if $zP = z$.

Definition 2.5.4 *Stationary Distribution of a CTMC : Let Q be the transition rate matrix of a CTMC, and let the vector z whose elements z_j denote the probability of being in state j be a probability distribution; i.e.,*

$$z_j \in \mathcal{R}, 0 \leq z_j \leq 1, \ \ and \sum_{all\ j} z_j = 1.$$

Then z is said to be a stationary distribution if and only if $zQ = 0$.

For a certain class of Markov chains, (see [16, pp. 15–16]), if the limiting distribution exists, it is equivalent to the stationary distribution. Furthermore it is independent of the initial distribution, i.e., in the long run the effects of the initial distribution disappears.

The popularity and power of the Markov chain modeling paradigm comes from the fact that, most of the interesting properties of the system being modeled can be derived from a set of simple linear equations. In the discrete case, the equations

$$\pi P = \pi, \| \pi \|_1 = 1,$$

and in the continuous case, the equations

$$\pi Q = 0, \| \pi \|_1 = 1,$$

let one to calculate quantitative measures about the system being modeled.

Also the reformulation of the equation $\pi P = \pi$, as $\pi(I - P) = 0$ show that the problem of finding the stationary distribution of a discrete–time Markov chain, can be viewed as similar to a continuous–time problem. Conversely, a matrix P can be obtained from Q by

$$P = I + \Delta t Q, \text{ where } \Delta t \leq \frac{1}{\max|q_{ii}|}.$$

The problem of finding the stationary distribution of a Markov chain can be thus formulated in three ways. First, it can be seen as an eigenvalue problem, i.e., $\pi P = \pi$; second, it can be formulated as a null space problem, i.e., $\pi Q = 0$; and finally, it can be seen as a linear system that can be obtained in a variety of ways, from $\pi Q = 0, \| \pi \|_1 = 1$.

We conclude this section by noting that all the discussed formulations of the problem imply that the Markov chains involved are time homogeneous, and this will be our assumption in the rest of the thesis.

2.6 Methods for Numerically Solving Markov Chain Problems

2.6.1 An Overview

As the problem described in the previous section can be formulated in different ways, there are a large number of methods one may use to attack the problem. In general terms, direct methods refer to those methods that calculate the solution vector in a predetermined number of steps [16, p. 61]. Iterative methods are provided with an initial approximation to the solution and they compute a new approximation to the new solution using the previous approximation in the previous iteration. The new approximation is supposed to become more and more close to the actual solution at each step.

Direct methods applied to Markov chain problems include Gaussian elimination and LU decomposition. We note that in the case of a Markov chain problem, a nontrivial solution other than the zero vector, to the system $\pi Q = 0$ is always available since it can be verified that Q is singular [16, p. 71].

Iterative methods can be grouped into two. First group of methods referred to as stationary methods include the power method, the method of Jacobi, the method of Gauss-Seidel and Successive Overrelaxation (SOR). The second group of methods are non-stationary methods, also referred to as Krylov subspace methods, include the method of Arnoldi, Generalized Minimum Residual Method (GMRES) and the full orthogonalization method [16, pp. 117–230], [13]. In this work we concentrate on stationary iterative methods. Here we first give a comparison of direct and iterative methods in the context of Markov chains [16, pp. 61–62].

The value of a Markov chain model increases as the system being modeled becomes more and more complex. The increase in the complexity of the model is generally reflected as an increase in the number of the states of the Markov model. This phenomenon is referred to as the *state–space explosion* problem. The increase in the number of states results in an increase in the size of the generator matrix. Beyond a certain limit, it becomes necessary to use a sparse storage scheme for storing the infinitesimal generator matrix. In addition to this, the matrices arising in Markov chain models are sparse, i.e., they contain only a few entries in each row. It is basically because of this reason that direct methods are considered disadvantageous, when compared to iterative solution techniques. Direct methods usually involve introducing new nonzero elements (fill–ins) into the matrix during factorization, which makes them inefficient and difficult to deal with. Also, beyond a certain limit, especially for large problems, it might not be possible to store the newly altered matrix in core memory. In contrast, iterative methods involve only matrix–vector multiplications or equivalent operations, which do not alter the nonzero structure of the matrix. In addition to this, by not altering the matrix, we avoid the round-off errors which are observed in direct methods.

In certain cases, it might not be necessary to compute the stationary vector of a Markov chain, to high accuracy. In such uses, iterative methods allow one to stop the computation at a predefined error term.

On the other hand, iterative methods are usually accompanied with a slow convergence rate to the solution. It is this reason that one may use a direct method for Markov chain problems whenever the method is not limited impractically by memory constraints. However, iterative methods are still the dominant choice, unless a practically implementable direct method gives the solution in less time.

Stationary methods have been the subject of much research. Although the non-stationary methods seem promising, much research needs to be done on their convergence properties and to predict the number of iterations required to find the solution of a problem. In the following sections we introduce the power method, method of Jacobi, Gauss-Seidel and SOR.

2.6.2 Power Method

The power method is used to find the right–hand eigenvector of an ordinary matrix corresponding to a dominant eigenvalue of the matrix. Thus, when the Markov chain problem is formulated as one of an eigenvalue problem, i.e., $\pi P = \pi$, power method might be used to solve the problem. Let the initial probability distribution among the states of a Markov chain be $\pi(0)$, and let the probability transition matrix of the same chain be P. Then after the process makes a transition, (at the next step), the probability distribution becomes $\pi^{(1)} = \pi^{(0)}P$. At the second step the probability distribution becomes $\pi^{(2)} = \pi^{(1)}P = \pi^{(0)}P^2$. At the k^{th} step, the probability distribution is found by $\pi^{(k)} = \pi^{(0)}P^{k-1}$. Note that, $\pi^{(k)}$ is the new approximation to the solution at step k. For certain classes of Markov chains [16, p. 16], the vector $\pi^{(k)}$ approaches to the stationary distribution, i.e.,

$$\lim_{k\to\infty} \pi^{(k)} = \pi, \text{ where } \pi = \pi P.$$

Power method is multiplying the approximation at each iteration by the probability transition matrix P, to obtain a new approximation. The convergence of power method is in general slow. Further properties of the method in the context of Markov chains can be found in [16, pp. 121–125].

2.6.3 Methods Based on Splittings

The stationary methods based on splittings are used for solving a system of linear equations. In the Markov chain context, when the problem is formulated as a linear system or a null–space problem, i.e., $\pi Q = 0$, these methods may be used. The methods Jacobi, Gauss-Seidel (GS) and SOR are based on splitting the infinitesimal generator matrix Q into $D - L - U$ where D is a strictly diagonal matrix, L is a strictly lower triangular matrix, and U is a strictly upper triangular matrix. The matrix D consists of the diagonal elements of Q, and the matrices L and U consist of negative of the strictly lower and strictly upper triangular elements of Q, respectively.

The Method of Jacobi

The problem of solving $\pi Q = 0$ can be formulated as

$$\begin{aligned} \pi Q &= 0 \\ \pi(D - L - U) &= 0 \\ \pi D &= \pi(L + U). \end{aligned}$$

From this we can obtain the iteration matrix of the Jacobi and the method of Jacobi

$$\pi^{(k+1)} = \pi^{(k)}(L + U)D^{-1}.$$

Hence, the method of Jacobi is equivalent to power method with the iteration matrix being $(L + U)D^{-1}$.

The Method of Gauss-Seidel

In a similar way to the method of Jacobi, the Gauss-Seidel method can be derived from the formulation

$$\begin{aligned} \pi Q &= 0 \\ \pi(D - L - U) &= 0 \end{aligned}$$

$$\begin{aligned} \pi(D-U) &= \pi L \\ \pi &= \pi L(D-U)^{-1}. \end{aligned}$$

From this we obtain the Gauss-Seidel Method as

$$\pi^{(k+1)} = \pi^{(k)} L(D-U)^{-1}.$$

Hence, it is equivalent to power method with the iteration matrix being $L(D-U)^{-1}$. The above formulation of the Gauss-Seidel method is referred as a forward Gauss-Seidel, because when the equations regarding individual entries of the vector are considered, the elements are calculated starting from the first element to the last element of the vector $\pi^{(k+1)}$.

Another formulation is possible, which may be expressed as

$$\pi^{(k+1)} = \pi^{(k)} U(D-L)^{-1}.$$

In this case the order of solving the equations for individual entries is from the last element to the first element of the vector $\pi^{(k+1)}$. Hence the method is called a backward Gauss-Seidel.

The Gauss-Seidel method is different from the method of Jacobi as it makes use of the elements that have already been computed. For instance while calculating the ith element of the $(k+1)$st approximation vector $\pi^{(k+1)}$, it makes use of the first $i-1$ elements that have been computed so far, in the case of forward Gauss-Seidel. A backward Gauss-Seidel makes use of the previously computed $n-i$ elements ranging from index $i+1$ to n, for a vector of size n, while calculating the ith element.

Successive Overrelaxation

The method of Successive Overrelaxation (SOR) is an extrapolation on the solution of the Gauss-Seidel. A parameter w is introduced to weigh the solution vector obtained from a Gauss-Seidel iteration with the previous approximation vector. When considered in this manner, the method can be expressed as

$$\pi_{SOR}^{(k+1)} = (1-w)\pi_{SOR}^{(k)} + w\pi_{GS}^{(k)}$$

where $\pi_{GS}^{(k)}$ is the resulting vector after applying the Gauss-Seidel algorithm to the kth approximation vector of SOR. Note that SOR is also called a forward SOR when the Gauss-Seidel iteration involved is a forward Gauss-Seidel, and a backward SOR when the Gauss-Seidel iteration involved is a backward Gauss-Seidel.

Hence forward SOR in matrix notation is

$$\pi^{(k+1)} = (1-w)\pi^{(k)} + w(\pi^{(k)}U(D-L)^{-1}),$$

and backward SOR in matrix notation is

$$\pi^{(k+1)} = (1-w)\pi^{(k)} + w(\pi^{(k)}L(D-U)^{-1}).$$

Note that an SOR iteration with $w = 1$ is equivalent to a Gauss-Seidel iteration. Sometimes SOR, is referred as Successive Under Relaxation method when $0 < w < 1$.

In addition to forward and backward versions of SOR, a Symmetric SOR (SSOR) has been introduced, which is simply a forward SOR followed by a backward SOR. In the case of Markov chain problems, there is little benefit in using a SSOR instead of SOR and this can be observed only in rare examples [16, p. 132].

Convergence characteristics of stationary methods in a general context can be found in [7] and references therein. In the Markov chain context, [16, pp. 133–176], [4, pp. 125–132] and [1, pp. 26–28] [16, pp. 138–142] provide discussions of these and other methods.

Block Versions of Iterative Methods Based on Splittings

Stationary block iterative methods are based on block partitioning of the generator matrix Q. Following [16, p. 139] we can demonstrate a block partitioning of the vector π and the matrix Q as

$$(\pi_1, \pi_2, \ldots, \pi_N), \begin{bmatrix} Q_{11} & Q_{12} & \ldots & Q_{1N} \\ Q_{21} & Q_{22} & \ldots & Q_{2N} \\ \ldots & \ldots & \ldots & \ldots \\ Q_{N1} & \ldots & \ldots & Q_{NN} \end{bmatrix}.$$

In this case, a block splitting of Q can be obtained as $Q = (D_N - L_N - U_N)$. D_N takes the form of a block diagonal matrix, L_N takes the form of a strictly lower block triangular matrix and U_N is a strictly upper block triangular matrix, i.e.,

$$D_N = \begin{bmatrix} D_{11} & 0 & \ldots & 0 \\ 0 & D_{22} & \ldots & 0 \\ \ldots & \ldots & \ldots & \ldots \\ 0 & \ldots & \ldots & D_{NN} \end{bmatrix},$$

$$L_N = \begin{bmatrix} 0 & 0 & \ldots & 0 \\ L_{21} & 0 & \ldots & 0 \\ \ldots & \ldots & \ldots & \ldots \\ L_{N1} & \ldots & L_{NN-1} & 0 \end{bmatrix}, U_N = \begin{bmatrix} 0 & U_{12} & \ldots & U_{NN} \\ 0 & 0 & \ldots & \ldots \\ \ldots & \ldots & \ldots & U_{N-1N} \\ 0 & \ldots & \ldots & 0 \end{bmatrix}.$$

By defining $\pi_i^{(k+1)}$ as the ith portion of π as shown, we may define block Jacobi as

$$\pi_i^{(k+1)} = \pi_i^{(k)}(\sum_{j=1}^{i-1} L_{ji} + \sum_{j=i+1}^{N} U_{ji})D_{ii}^{-1} \text{ for all } i,$$

forward block Gauss-Seidel as

$$\pi_i^{(k+1)} = \pi_i^{(k)}(\sum_{j=1}^{i-1} L_{ji})(D_{ii} - \sum_{j=i+1}^{N} U_{ji})^{-1} \text{ for all } i,$$

and backward block Gauss-Seidel as

$$\pi_i^{(k+1)} = \pi_i^{(k)}(\sum_{j=i+1}^{N} U_{ji})(D_{ii} - \sum_{j=1}^{i-1} L_{ji})^{-1} \text{ for all } i.$$

The difference between the point and the block versions of the algorithms is that in the block versions, all elements of $\pi_i^{(k+1)}$ in a portion of $\pi^{(k+1)}$ are solved simultaneously. It is possible to use a direct method or another iterative method to solve the individual blocks. In this way, one can obtain a more accurate approximation at each iteration, obviously with an extra cost being introduced at each iteration. In [1, pp. 26–28] block iterative methods are discussed within the context of Markov chains.

Chapter 3

Stochastic Automata Networks

3.1 Preliminaries

In the previous chapter we have seen that if a Markov chain model of a system is available, quantitative measures about the system can be obtained from the system of equations

$$\pi Q = 0, \parallel \pi \parallel_1 = 1.$$

There are a number of methodologies for developing a Markov chain model of a system. Petri nets [8] are such a formalism for generating Markov chain models of systems. Alternatively, there are special software tools for generating Markov chain models [15]. Independent of the paradigm used, the problem of state–space explosion is observed in almost all applications. In some cases, as the applications become more interesting, the size of the Markov chain gets so large that it is impractical to find a solution.

A Stochastic Automata Network (SAN) is another formalism for generating a Markov model. They are most suitable for performance modeling of parallel and distributed systems. The model is generated by considering an individual automaton for each component of the system. Each individual component is modeled by a single stochastic automaton and the interactions between the components are incorporated into the model. The main advantage of the SAN methodology is that the model is stored very efficiently, i.e., the memory occupied by the model is very small compared to the size of the model generated.

Before getting into the formal definitions of SANs and their properties, we give a basic overview of tensor algebra which is a building block for SAN methodology.

3.2 Tensor Algebra

3.2.1 Ordinary Tensor Algebra

We now list several definitions regarding tensor algebra. These and more properties of tensor algebra concepts can be found in [2].

In the following, we use $A_{m\times n}$ for a matrix of dimension $m \times n$, B_{kl} for a matrix of dimension $k \times l$, $C_{mk\times nl}$ and $D_{mk\times nl}$ for matrices of size $mk \times nl$.

Definition 3.2.1 *Ordinary Tensor Product:(OTP) Let A_{mn} and B_{kl} be two matrices, as*

$$A = \begin{bmatrix} a_{11} & \dots & \dots & a_{1n} \\ \vdots & \dots & \dots & \vdots \\ a_{m1} & \dots & \dots & a_{mn} \end{bmatrix}, B = \begin{bmatrix} b_{11} & \dots & b_{1l} \\ \vdots & \dots & \vdots \\ \vdots & \dots & \vdots \\ b_{k1} & \dots & b_{kl} \end{bmatrix}$$

then the ordinary tensor product of A and B, $C_{mk\ nl} = A \otimes B$ is given by

$$C = \begin{bmatrix} a_{11}B & \dots & \dots & a_{1n}B \\ \vdots & \dots & \dots & \vdots \\ a_{m1}B & \dots & \dots & a_{mn}B \end{bmatrix}.$$

and the ordinary tensor product of B and A, $D_{mk\ nl} = B \otimes A$ is given by

$$D = \begin{bmatrix} b_{11}A & \dots & b_{1l}A \\ \vdots & \dots & \vdots \\ \vdots & \dots & \vdots \\ b_{k1}A & \dots & b_{kl}A \end{bmatrix}.$$

Notice that $C \neq D$.

Definition 3.2.2 *Ordinary Tensor Sum: The tensor sum of two* square *matrices A_{nn} and B_{mm}, $C_{nm\ nm} = A \oplus B$ is defined as*

$$C = A \otimes I_m + I_n \otimes B$$

Further important properties of tensor algebra as they appear in [5, pp. 4–5] are listed below. Note that all matrices are square.

- Associativity :

$A \otimes (B \otimes C) = (A \otimes B) \otimes C)$ and $A \oplus (B \oplus C) = (A \oplus B) \oplus C)$.

- Distributivity over ordinary matrix addition :

$(A + B) \otimes (C + D) = A \otimes C + B \otimes C + A \otimes D + B \otimes D$.

- Compatibility with ordinary matrix multiplication :(case I)

$(A \times B) \otimes (C \times D) = (A \otimes C) \times (B \otimes D)$.

- Compatibility with ordinary matrix multiplication :(case II)

$$\begin{aligned} \bigotimes_{i=1}^{N} A^{(i)} &= \prod_{i=1}^{N} I_{n_1} \otimes \ldots \otimes I_{n_{i-1}} \otimes A^{(i)} \otimes I_{n_{i+1}} \otimes \ldots \otimes I_{n_N} \\ &= \prod_{i=1}^{N} I_{1:i-1} \otimes A^{(i)} \otimes I_{i+1:N}, \end{aligned}$$

where $I_{i:j}$ is the identity matrix of size $\prod_{k=i}^{j} n_k$.

- Compatibility with ordinary matrix inversion :

$(A \times B)^{-1} = A^{-1} \otimes B^{-1}$.

- Pseudo Commutativity :

$A \otimes B = P_\tau (B \otimes A) P_\tau^T$,
where P_τ is a permutation matrix of order $n_1 \times n_2$, n_1 is the size of matrix A and n_2 is the size of matrix B.

Note that no commutativity other than the given pseudo commutativity property holds for ordinary tensor products.

It is straightforward to extend these properties to N term tensor products and sums. For our purpose of illustrating several algorithms, noting that

$$\bigotimes_{k=1}^{N} A^{(k)} = \sum_{k=1}^{N} A^{(k)} I_{n_1} \otimes \ldots \otimes I_{n_{k-1}} \otimes A^{(k)} \otimes I_{n_{k+1}} \otimes \ldots \otimes I_{n_N},$$

where I_{n_k} is defined to be the identity matrix of size n_k which is the size of the k^{th} term of the tensor product , $A^{(k)}$, is sufficient.

3.2.2 Generalized Tensor Algebra

Ordinary tensor algebra is used in other fields of science as well as SAN modeling. However, it does not allow one to handle certain constructs that arise in SAN models. Since such constructs are essential for any meaningful model, tensor algebra has been extended in order to cope with them. Generalized tensor algebra refers to tensor algebra where the elements of the matrices may be real valued functions. In SAN context, the functional elements are functions of the states of one or more automata. We now give several definitions and properties of generalized tensor algebra. These with more detailed discussions and proofs can be found in [5, pp. 13-20]. We follow the conventions there and assume that all matrices are square, which is the case for us. A matrix of the form $B[A]$ refers to a functional matrix B which contains entries that are dependent on the state of automata with transition rate matrix A. In general an expression of the form $A^{(m)}[\mathcal{A}^{(1)}, \mathcal{A}^{(2)}, \ldots, \mathcal{A}^{(m-1)}]$ denotes a matrix $A^{(m)}$ that contain functional entries that depend on the states of the automata $\mathcal{A}^{(1)}, \mathcal{A}^{(2)}, \ldots, \mathcal{A}^{(m-1)}$. Note that the state of an automaton is determined by the row of the generator matrix, i.e., elements on row i of the matrix are transition rates out of state i, except of course the diagonal element which is interpreted as the rate of staying in the same state. The operator $\otimes_g$ is used for generalized tensor products.

Definition 3.2.3 *Generalized Tensor Product:(GTP Case I) Let A and $B[A]$ be two square matrices with sizes n_a and n_b respectively, and let B contain functional entries that depend on the state of A. Then the generalized tensor* *product of A and B, $C = A \otimes_g B[A]$ is given by*

$$\begin{bmatrix} a_{11}B(a_1) & a_{12}B(a_1) & \dots & a_{1n_a}B(a_1) \\ a_{21}B(a_2) & a_{22}B(a_2) & \dots & a_{2n_a}B(a_2) \\ \vdots & \vdots & \ddots & \vdots \\ a_{n_a1}B(a_{n_a}) & a_{n_a2}B(a_{n_a}) & \dots & a_{n_an_a}B(a_{n_a}) \end{bmatrix}$$

Definition 3.2.4 *Generalized Tensor Product:(Case II) Let $A[B]$ and B be two square matrices with sizes n_a and n_b respectively, and let A contain functional entries that depend on the state of B. Then the generalized tensor product of A and B, $C = A[B] \otimes_g B$ is given by*

$$\begin{bmatrix} a_{11}[B]I_{n_b} \times B & a_{12}[B]I_{n_b} \times B & \dots & a_{1n_a}[B]I_{n_b} \times B \\ a_{21}[B]I_{n_b} \times B & a_{22}[B]I_{n_b} \times B & \dots & a_{2n_a}[B]I_{n_b} \times B \\ \vdots & \vdots & \ddots & \vdots \\ a_{n_a1}[B]I_{n_b} \times B & a_{n_a2}[B]I_{n_b} \times B & \dots & a_{n_an_a}[B]I_{n_b} \times B \end{bmatrix}$$

where $a_{ij}[B]I_{n_b}$ is defined as $diag\{a_{ij}(b1), a_{ij}(b2), \dots, a_{ij}(b_{nb})\}$ and $a_{ij}(b_k)$ is the functional element of A with its function being evaluated at state k of B.

Definition 3.2.5 *Generalized Tensor Product:(Case III) Let $A[B]$ and $B[A]$ be two square matrices with sizes n_a and n_b respectively, and let A contain functional entries that depend on the state of B. Let B also contain functional entries that depend on the state of A. Then the generalized tensor product of A and B, $C = A[B] \otimes_g B[A]$ is given by*

$$\begin{bmatrix} a_{11}[B]I_{n_b} \times B(a_1) & a_{12}[B]I_{n_b} \times B(a_1) & \dots & a_{1n_a}[B]I_{n_b} \times B(a_1) \\ a_{21}[B]I_{n_b} \times B(a_2) & a_{22}[B]I_{n_b} \times B(a_2) & \dots & a_{2n_a}[B]I_{n_b} \times B(a_2) \\ \vdots & \vdots & \ddots & \vdots \\ a_{n_a1}[B]I_{n_b} \times B(a_{n_a}) & a_{n_a2}[B]I_{n_b} \times B(a_{n_a}) & \dots & a_{n_an_a}[B]I_{n_b} \times B(a_{n_a}) \end{bmatrix}$$

where $a_{ij}[B]I_{n_b}$ and $a_{ij}(b_k)$ are defined as in case II.

Now let us see some of the properties of generalized tensor algebra that are of interest to us.

- Associativity :

$A[B,C] \otimes_g (B[A,C] \otimes_g C[A,B]) = (A[B,C] \otimes_g B[A,C]) \otimes_g C[A,B])$.

- Distributivity over ordinary matrix addition :

$$(A_1[B] + A_2[B]) \otimes_g (B_1[A] + B_2[A]) =$$
$$A_1[B] \otimes_g B_1[A] + A_1[B] \otimes_g B_2[A] + A_2[B] \otimes_g B_1[A] + A_1[B] \otimes_g B_2[A].$$

- Compatibility with ordinary matrix multiplication :

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

- Pseudo Commutativity :

$$\bigotimes_{g \ k=1}^{N} A^{(k)}[\mathcal{A}^{(1)}, \ldots, \mathcal{A}^{(N)}] = P_\tau \bigotimes_{g \ k=1}^{N} A^{(\tau(k))}[\mathcal{A}^{(1)}, \ldots, \mathcal{A}^{(N)}] P_\tau^T,$$

where τ is a permutation of integers $[1, 2, \ldots, N]$, and P_τ is a permutation matrix of order $\prod_{i=1}^{N} n_i$.

3.3 Stochastic Automata Networks

Let us consider two stochastic automata initially without any interaction among them. The system being modeled has two components, each of which can be represented by a single automaton. A Stochastic Automata Network, describing the system, is represented by two automata, $\mathcal{A}^{(1)}$ and $\mathcal{A}^{(2)}$. If the automata are models obtained from DTMCs, i.e., they are defined by probability transition matrices $P^{(1)}$ and $P^{(2)}$, the whole system is defined by the transition probability matrix obtained from a tensor product, i.e., $P^{(1)} \otimes P^{(2)}$. On the other hand if the automata are models obtained from a CTMC, i.e., have transition rate matrices $Q^{(1)}$ and $Q^{(2)}$, the transition rate matrix of the whole system is obtained from the tensor sum of $Q^{(1)}$ and $Q^{(2)}$ as $Q^{(1)} \oplus Q^{(2)}$. If the probability distribution of the states of the first automaton at time t, is represented by vector $\pi^{(1)}(t)$, and similarly if the probability distribution of the second automaton are represented by $\pi^{(2)}(t)$ at time t, the probability distribution describing the state of the whole system at time t, is given by $\pi^{(1)}(t) \otimes \pi^{(2)}(t)$. If the first automaton has n_1 states and the second automaton has n_2 states, the whole system has $n_1 \times n_2$ states. Each state of the *global* system is a combination of the states of the two automata. The global state of the system can be represented by a 2–tuple, i.e., if the first automaton is at state i, and the second automaton is at state j, the global system is at state (i, j). It can be easily verified that each row of the global generator matrix and the global state distribution vector corresponds to a state of the global system represented as a 2–tuple. A consequence of these results is that the stationary distribution of the global system can be obtained from the tensor product of the stationary vectors of the individual automata. Hence, it is straightforward to find the stationary vector of a SAN with noninteracting automata, i.e, first solve for the stationary vectors of the individual automata then calculate the tensor product of them.

In case the global system is modeled by N automata, $\mathcal{A}^{(1)}, \mathcal{A}^{(2)}, \ldots, \mathcal{A}^{(N)}$, the global generator is obtained by

$$Q = \bigoplus_{i=1}^{N} Q^{(i)}$$

in the continuous case, and by

$$P = \bigotimes_{i=1}^{N} P^{(i)}$$

in the discrete case.

In both cases the global state distribution vector is obtained by

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

3.4 Capturing the Interactions

3.4.1 Functional Transitions

In order to be able to model complex systems, especially parallel and distributed systems that have interacting components, one needs to model the interaction between the components. One extension that enables us to incorporate such interactions between individual components is by means of *functional transitions*. The stochastic automaton, modeling a component, is allowed to have transitions whose rate is a function of the states of several automata. Now, the entries of the transition rate matrix might be functional, i.e., the transition rate matrix is not an ordinary real valued matrix but it is a matrix whose entries may contain a real valued function. Note that if the rate of a transition is dependent only on the automaton that makes the transition, the transition is considered to be a constant transition, not a functional one.

Similar to noninteracting automata, the global generator matrix of the global system composed of CTMC can be described as a tensor sum of the generator matrices of the individual automata, yet this time as a generalized tensor sum of the individual matrices, i.e.,

$$Q = \bigoplus_{i=1}^{N}{}_{g}\; Q^{(i)}.$$

In the discrete case a generalized tensor product is needed, i.e.,

$$P = \bigotimes_{i=1}^{N}{}_{g}\; P^{(i)}.$$

Two important points to note about functional transitions in SAN descriptors is that; first, still the transitions of an automaton effect only the automaton itself even if the transition is a functional one, second, the nonzero structure of the generator matrix is still in a suitable form to store in sparse format, i.e., no zero entry may become nonzero during function evaluation yet some nonzero entries might evaluate to zero.

3.4.2 Synchronizing Events

Another concept, introduced to extend the modeling capability of a stochastic automata network, is one of a synchronizing event. A synchronizing event is either a transition of one automaton that force one or more automata to specific states, or an event in which an automaton being (or not being) in a state force some other automata to get into or stuck at certain states. The transitions that are involved in a synchronizing event may also contain functional rates. It is possible to have a synchronizing event in which a transition of an automaton cause several or all automata to make or block transitions. Note that, a synchronizing event causes the state of the global system to be altered, whereas a functional transition affects only the automaton that contains the transition.

In SAN terminology, the automaton that initiates a transition in the other automata in a synchronizing event is called the *master* automaton of the synchronizing event. The automata that are effected by the master automaton's transition are called the *slave* automata. Note that a transition in the master automaton has a rate associated with it, whereas the induced transitions in the slaves happen instantaneously with the master automaton's transition.

3.5 Descriptor of a SAN

In this section we introduce the concept of a descriptor for a SAN. Here and hereafter, we consider only continuous–time stochastic automata and hence all matrices are transition rate matrices. The extension of the concepts introduced, to discrete–time automata are possible. The effect of a synchronizing event on a SAN is captured by introducing new tensor product expressions. If there is a synchronizing event labeled e that appears in a SAN with N automata, one new tensor product of the form

$$\bigotimes_{i=1}^{N} Q_e^{(i)}$$

and another one in the form

$$\bigotimes_{i=1}^{N} \bar{Q}_e^{(i)}$$

are introduced. The last term is referred to as the diagonal corrector of the synchronizing event and is introduced to maintain the global generator as a transition rate matrix. In the most general case, where the transitions involved in the synchronizing event, say e_i, are functional, the tensor products are generalized tensor products and the expressions introduced are in the forms

$$\bigotimes_{g}{}_{i=1}^{N} Q_e^{(i)}$$

and

$$\bigotimes_{g}{}_{i=1}^{N} \bar{Q}_e^{(i)}.$$

For a SAN model with N automata, there are N matrices in the tensor products, each of which correspond to one automaton in the SAN. For each synchronizing event, the order of the terms in each tensor product are explicit as described by

$$Q_e^{(1)} \otimes_g Q_e^{(2)} \otimes_g \ldots \otimes_g Q_e^{(N)}$$

and

$$\bar{Q}_e^{(1)} \otimes_g \bar{Q}_e^{(2)} \otimes_g \ldots \otimes_g \bar{Q}_e^{(N)}.$$

This is important since neither ordinary nor generalized tensor products are commutative.

Since for each synchronizing event, two new tensor products are introduced, for a SAN model with E synchronizing events, $2E$ tensor products are introduced. The global generator of a SAN with N automata and E synchronizing events is obtained from the equation

$$\begin{aligned}
Q &= \bigoplus_{g}{}_{i=1}^{N} Q^{(i)} + \sum_{i=1}^{E} \bigotimes_{g}{}_{i=1}^{N} Q_e^{(i)} + \sum_{j=1}^{E} \bar{Q}_{e_j}^{(i)} \\
&= \sum_{i=1}^{N} I_{n_1} \otimes \ldots \otimes I_{n_{i-1}} \otimes Q^{(i)} \otimes I_{n_{i+1}} \otimes \ldots \otimes I_{n_N} + \sum_{j=1}^{2E} \bigotimes_{g}{}_{i=1} Q_j^{(i)} \\
&= \sum_{j=1}^{N+2E} \bigotimes_{g}{}_{i=1} Q_j^{(i)}
\end{aligned}$$

and the form of it as in the last line is referred to as the *descriptor* of the SAN. The first set of N tensor products are referred to as the local generator matrices, the E tensor products of the form $\sum_{j=1}^{E} \bigotimes_{i=1 \quad g}^{N} Q_{e_j}^{(i)}$ are referred as the synchronizing event matrices, the final E tensor products of the form $\sum_{j=1}^{E} \bigotimes_{i=1 \quad g}^{N} \bar{Q}_{e_j}^{(i)}$ are referred as the corrector matrices. The synchronizing event matrices reflect the interaction among the automata involved in the event. The corrector matrices are diagonal matrices introduced to make the global generator matrix a transition rate matrix. Further information about the rationale behind these matrices with the related proofs might be found in [9], [10], [11] . We now give an example SAN to illustrate the concepts introduced in this chapter. The example SAN appears in [16, pp. 470–472].

The SAN has two automata, one with two states and the other with three states. It has two synchronizing events and there are also functional rates. There is a functional transition in the second automaton $\mathcal{A}^{(2)}$, the transition from state 2 to state 3 occur with rate $\hat{\mu}_2$ if the first automaton, $\mathcal{A}^{(1)}$,is in state 1 and with $\tilde{\mu}_2$ if the first automaton is in state 2. The local generator matrix of $\mathcal{A}^{(1)}$ is given by

$$Q_l^{(1)} = \begin{bmatrix} -\lambda_1 & \lambda_1 \\ 0 & 0 \end{bmatrix}$$

and the local generator matrix of $\mathcal{A}^{(2)}$ is given by

$$Q_l^{(2)} = \begin{bmatrix} -\mu_1 & \mu_1 & 0 \\ 0 & -f & f \\ 0 & 0 & 0 \end{bmatrix}.$$

For the second automaton, the functional transition rate f is defined by

$$f = \begin{cases} \hat{\mu}_2 & \text{if } s\mathcal{A}^{(1)} = 1 \\ \tilde{\mu}_2 & \text{if } s\mathcal{A}^{(1)} = 2 \end{cases},$$

where $s\mathcal{A}^{(i)}$ is a function that maps automaton $\mathcal{A}^{(i)}$ to its state.

The first synchronizing event e_1, occurs by a transition of the first automaton, $s\mathcal{A}^{(1)}$, from state 2 to state 1, which happens at a rate λ_2, causing the second automaton, $s\mathcal{A}^{(1)}$, to state 1. The synchronizing event matrix and the corrector matrix corresponding to the first synchronizing event e_1, for the first automaton are given by

$$Q_{e_1}^{(1)} = \begin{bmatrix} 0 & 0 \\ \lambda_2 & 0 \end{bmatrix}, \ \bar{Q}_{e_1}^{(1)} = \begin{bmatrix} 0 & 0 \\ 0 & -\lambda_2 \end{bmatrix},$$

and the synchronizing event matrix and the corrector matrix corresponding to the first synchronizing event e_1, for the second automaton are given by

$$Q_{e_1}^{(2)} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}, \ \bar{Q}_{e_1}^{(2)} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

The second automaton is the master of the second synchronizing event. Whenever the second automaton makes a transition from state 3 to state 1, which happens with rate μ_3, it causes the first automaton to state 1. The first automaton has

$$Q_{e_2}^{(1)} = \begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix},$$

as the synchronizing event matrix and has

$$\bar{Q}_{e_2}^{(1)} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix},$$

as the corrector matrix. The second automaton has

$$Q_{e_2}^{(2)} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ \mu_3 & 0 & 0 \end{bmatrix},$$

as the synchronizing event matrix and has

$$\bar{Q}_{e_2}^{(2)} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & -\mu_3 \end{bmatrix},$$

as the corrector matrix.

The descriptor of the SAN may be expanded as

$$\begin{aligned} Q &= \bigoplus_{i=1}^{N} Q_l^{(i)} + \sum_{e=1}^{E} \bigotimes_{i=1}^{N} Q_e^{(i)} + \sum_{e=1}^{E} \bigotimes_{i=1}^{N} \bar{Q}_e^{(i)} \\ &= Q_l^{(1)} \oplus Q_l^{(2)} + Q_{e_1}^{(1)} \otimes Q_{e_1}^{(2)} + Q_{e_2}^{(1)} \otimes Q_{e_2}^{(2)} + \bar{Q}_{e_1}^{(1)} \otimes \bar{Q}_{e_1}^{(2)} + \bar{Q}_{e_2}^{(1)} \otimes \bar{Q}_{e_2}^{(2)}, \end{aligned}$$

and from this we obtain the global generator as

$$\left[\begin{array}{ccc|ccc} -(\lambda_1+\mu_1) & \mu_1 & 0 & \lambda_1 & 0 & 0 \\ 0 & -(\lambda_1+\hat{\mu}_2) & \hat{\mu}_2 & 0 & \lambda_1 & 0 \\ \mu_3 & 0 & -(\lambda_1+\mu_3) & 0 & 0 & \lambda_1 \\ \hline \lambda_2 & 0 & 0 & -(\lambda_2+\mu_1) & \mu_1 & 0 \\ \lambda_2 & 0 & 0 & 0 & -(\lambda_2+\tilde{\mu}_2) & \tilde{\mu}_2 \\ \lambda_2+\mu_3 & 0 & 0 & 0 & 0 & -(\lambda_2+\mu_3) \end{array}\right].$$

3.6 Efficient Tensor Product Vector Multiplication

The problem of finding the stationary distribution vector of a SAN with N automata and E synchronizing events involve solving the linear system of equations

$$\pi \sum_{j=1}^{N} \bigotimes_{i=1}^{N} Q_j^{(i)} = 0.$$

All iterative methods developed for solving this system of equations need to do a tensor product vector multiplication of the form

$$\pi \bigotimes_{i=1}^{N} Q^{(i)}.$$

Note that it is possible to first expand the tensor product and then do an ordinary vector matrix multiplication. However, in such an implementation the number of multiplications for finding the matrix resulting from the tensor product is $\prod_{i=1}^{N} n_i$, and another set of $\prod_{i=1}^{N} n_i$ multiplication operations is necessary for carrying out the matrix vector multiplication. Obviously, this is very inefficient both storage-wise and time-wise. Instead, in [17] Stewart et. al., suggest an algorithm with a lower computational complexity and without the need for expanding the tensor product, if there are no functional rates in the matrices, i.e., the tensor product is an ordinary tensor product. We give the theorem indicating the complexity of the multiplication operation and the algorithm (see Table 3.1). The proof of the theorem and a detailed discussion of the algorithm can be found in [17, pp. 516–517].

1. Initialize: $nleft = n_1 n_2 \dots n_{N-1}$; $nright = 1$.
2. For $i = N, \dots, 2, 1$ do
   - $base = 0$; $jump = n_i \times nright$
   - For $k = 1, 2, \dots, nleft$ do
     - ∘ For $j = 1, 2, \dots, nright$ do
       - ⋆ $index = base + j$
       - ⋆ For$l = 1, 2, \dots, n_i$ do
         - · $z_l = \pi_{index}$; $index = index + nright$
       - ⋆ Multiply: $z^{'} = z \times Q^{(i)}$
       - ⋆ $index = base + j$
       - ⋆ For $l = 1, 2, \dots, n_i$ do
         - · $\pi^{'}_{index} = z^{'}_l$; $index = index + nright$
     - ∘ $base = base + jump$
   - $nleft = nleft / n_{i-1}$
   - $nright = nright \times n_i$
   - $\pi = \pi^{'}$

Figure 3.1: Vector multiplication with an ordinary tensor product

Theorem 3.6.1 *The product*

$$\pi \bigotimes_{i=1}^{N} Q^{(i)}$$

where $Q^{(i)}$, of order n_i, contains only constant terms and π is a real vector of length $\prod_{i=1}^{N} n_i$, may be computed in ρ_N multiplications , where

$$\rho_N = n_N \times (\rho_{N-1} + \prod_{i=1}^{N} n_i) = \prod_{i=1}^{N} n_i \times \sum_{i=1}^{N} n_i.$$

When there are functional rates in the automata, the tensor products become generalized tensor products. In this case a slightly modified version of the algorithm is applicable with a restriction on the ordering of the automata and their dependencies. The following theorem and algorithm in Figure 3.2 are applied in such cases. Again, more detailed information about this version of the algorithm and the theorem are provided in [17].

Theorem 3.6.2 *The multiplication*

$$\pi \times (Q^{(1)} \otimes_g Q^{(2)}[\mathcal{A}^{(1)}] \otimes_g Q^{(3)}[\mathcal{A}^{(1)}, \mathcal{A}^{(2)}] \otimes_g \dots \otimes_g Q^{(N)}[\mathcal{A}^{(1)}, \dots, \mathcal{A}^{(N-1)}])$$

1. Initialize: $nleft = n_1 n_2 \dots n_{N-1}; nright = 1.$
2. For $i = N, \dots, 2, 1$ do
   - $base = 0; jump = n_i \times nright$
   - For $k = 1, 2, \dots, nleft$ do
     - ∘ For $j = 1, 2, \dots, i-1$ do
       - ⋆ $k_j = \left(\left[(k-1) / \prod_{l=j+1}^{i-1} n_l \right] mod \left(\prod_{l=j+1}^{i-1} n_l \right) \right) + 1$
     - ∘ For $j = 1, 2, \dots, nright$ do
       - ⋆ $index = base + j$
       - ⋆ For$l = 1, 2, \dots, n_i$ do
         - · $z_l = \pi_{index}; index = index + nright$
       - ⋆ Multiply: $z' = z \times Q^{(i)}[a_{k_1}^{(1)}, \dots, a_{k_{i-1}}^{(i-1)}]$
       - ⋆ $index = base + j$
       - ⋆ For $l = 1, 2, \dots, n_i$ do
         - · $\pi'_{index} = z'_l; index = index + nright$
     - ∘ $base = base + jump$
   - $nleft = nleft / n_{i-1}$
   - $nright = nright \times n_i$
   - $\pi = \pi'$

Figure 3.2: Vector multiplication with a generalized tensor product

where $Q^{(i)}$, of order n_i and π is a real vector of length $\prod_{i=1}^{N} n_i$, may be computed in ρ_N multiplications , where

$$\rho_N = n_N \times (\rho_{N-1} + \prod_{i=1}^{N} n_i) = \prod_{i=1}^{N} n_i \times \sum_{i=1}^{N} n_i.$$

It should be clear that the dependency list for an automaton as it appears in the definitions and properties is not strict, i.e., actually an automaton might depend on a subset of the automata in its parameter list. Notice the order of dependencies among the automata for the compatibility of generalized tensor products with ordinary matrix multiplications, the first automaton should be independent of the rest, the second automaton may only depend on the first one, and each automaton may depend on a subset of the automata that precede it. The final automaton might depend on all the remaining automata.

Chapter 4

Stationary Iterative Methods for a SAN

4.1 The splitting of a SAN descriptor

In order to use stationary iterative methods such as Jacobi, GS, and SOR for solving a SAN, the corresponding descriptor needs to be split. Here we give a suitable splitting for a SAN descriptor in the form $D - L - U$ [16, p. 126]. By a suitable splitting we mean one in which L, D, and U each consists of a sum of tensor products so that iterative methods of interest may be implemented in terms of the efficient vector–tensor product multiplication algorithm.

The derivations of the splittings are based on the associativity of tensor products and distributivity of tensor product over matrix addition [2]. These two properties are valid for both OTP and GTP [5]. In other words, the splittings exist in both the nonfunctional (i.e., OTP) case and the functional (i.e., GTP) case. Obviously, limitations on the applicability of the efficient vector–descriptor multiplication algorithm still remain [5, pp. 13–24].

The descriptor of a SAN with N automata and E synchronizing events is given by

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

However we can rewrite (1) as

$$Q = Q_l + Q_e + \bar{Q}_e,$$

where

$$\begin{aligned} Q_l &= \bigoplus_{i=1}^{N} Q_l^{(i)}, \\ Q_e &= \sum_{e=1}^{E} \bigotimes_{i=1}^{N} Q_e^{(i)}, \\ \bar{Q}_e &= \sum_{e=1}^{E} \bigotimes_{i=1}^{N} \bar{Q}_e^{(i)}. \end{aligned}$$

Assuming that the ith automaton has n_i states, the global generator will have $n = \prod_{i=1}^{N} n_i$ states. The generator $Q_l^{(i)}$ is comprised of local transitions in the ith automaton.

First, we introduce some lemmas. Then we give a theorem that follows from the lemmas, for splitting the descriptor of a SAN.

Lemma 4.1.1 *The tensor product of two diagonal matrices D_1 and D_2 is a diagonal matrix $D(= D_1 \otimes D_2)$.*

Proof. By the definition of the $\otimes$ operator, D is a block diagonal matrix where each block is equal to D_2, and since D_2 is a diagonal matrix, D is also diagonal. □

Lemma 4.1.2 *The tensor product of a diagonal matrix D_1 and a strictly lower triangular matrix L_1 is a strictly lower triangular matrix $L(= D_1 \otimes L_1)$.*

Proof. By the definition of the $\otimes$ operator, L is a block diagonal matrix where each diagonal block is equal to L_1. Since L_1 is strictly lower triangular, L is a block diagonal matrix with strictly lower triangular blocks along the diagonal; hence, it is a strictly lower triangular matrix. □

Lemma 4.1.3 *The tensor product of a diagonal matrix D_1 and a strictly upper triangular matrix U_1 is a strictly upper triangular matrix $U(= D_1 \otimes U_1)$.*

Proof. By the definition of the $\otimes$ operator, U is a block diagonal matrix where each diagonal block is equal to U_1. Since U_1 is strictly upper triangular, U is a block diagonal matrix with strictly upper triangular blocks along the diagonal; hence, it is a strictly upper triangular matrix. $\square$

Lemma 4.1.4 *The tensor product of a strictly lower triangular matrix L_1 and a matrix A_1 of arbitrary nonzero structure is a strictly lower triangular matrix $L(= L_1 \otimes A_1)$.*

Proof. By the definition of the $\otimes$ operator, L is a block strictly lower triangular matrix with zero blocks of the order of A_1 in the diagonal and upper triangular parts. Thus L has zero elements in the diagonal and upper triangular parts; it is strictly lower triangular. $\square$

Lemma 4.1.5 *The tensor product of a strictly upper triangular matrix U_1 and a matrix A_1 of arbitrary nonzero structure is a strictly upper triangular matrix $U(= U_1 \otimes A_1)$.*

Proof. By the definition of the $\otimes$ operator, U is a block strictly upper triangular matrix with zero blocks of the order of A_1 in the diagonal and lower triangular parts. Thus U has zero elements in the diagonal and lower triangular parts; it is strictly upper triangular. $\square$

Lemma 4.1.6 *$\bar{Q}_e$ is a diagonal matrix.*

Proof. Since $\bar{Q}_e = \sum_{e=1}^{E} \bigotimes_{i=1}^{N} \bar{Q}_e^{(i)}$ and each $\bar{Q}_e^{(i)}$ is diagonal. Then from Lemma 4.1.1, $\bar{Q}_e$ is diagonal. $\square$

Lemma 4.1.7 *Q_l can be split as $D_l - L_l - U_l$, where D_l is diagonal, L_l is strictly lower triangular, U_l is strictly upper triangular and each of the three terms is in the form of a sum of tensor products.*

Proof. Let $Q_l^{(i)}$ be split as $D_l^{(i)} - L_l^{(i)} - U_l^{(i)}$, where $D_l^{(i)}$ is diagonal, $L_l^{(i)}$ is strictly lower triangular, and $U_l^{(i)}$ is strictly upper triangular. We use $I_{n_i:n_j}$, to represent an identity matrix of size $\prod_{k=i}^{j} n_k$ when $i \le j$, else a one. Then

$$\begin{aligned}
Q_l &= \bigoplus_{i=1}^{N} Q_l^{(i)} \\
&= \sum_{i=1}^{N} I_{n_1} \otimes I_{n_2} \otimes \cdots \otimes Q_l^{(i)} \otimes \cdots \otimes I_{n_{N-1}} \otimes I_{n_N} \\
&= \sum_{i=1}^{N} I_{n_1:n_{i-1}} \otimes Q_l^{(i)} \otimes I_{n_{i+1}:n_N} \\
&= \sum_{i=1}^{N} I_{n_1:n_{i-1}} \otimes (D_l^{(i)} - L_l^{(i)} - U_l^{(i)}) \otimes I_{n_{i+1}:n_N} \\
&= \sum_{i=1}^{N} (I_{n_1:n_{i-1}} \otimes D_l^{(i)} \otimes I_{n_{i+1}:n_N}) - \sum_{i=1}^{N} (I_{n_1:n_{i-1}} \otimes L_l^{(i)} \otimes I_{n_{i+1}:n_N}) \\
&\quad - \sum_{i=1}^{N} (I_{n_1:n_{i-1}} \otimes U_l^{(i)} \otimes I_{n_{i+1}:n_N}) \\
&= D_l - L_l - U_l
\end{aligned}$$

The last equality is a consequence of Lemmas 4.1.1, 4.1.2, 4.1.3, 4.1.4, and 4.1.5. □

Lemma 4.1.8 *Q_e can be split as $D_e - L_e - U_e$ where D_e is diagonal, L_e is strictly lower triangular, U_e is strictly upper triangular and each of the three terms are in the form of a sum of tensor products.*

Proof. Let $Q_e^{(i)}$ be split as $D_e^{(i)} - L_e^{(i)} - U_e^{(i)}$, where $D_e^{(i)}$ is diagonal, $L_e^{(i)}$ is strictly lower triangular, and $U_e^{(i)}$ is strictly upper triangular. Then

$$\begin{aligned}
Q_e &= \sum_{e=1}^{E} \bigotimes_{i=1}^{N} Q_e^{(i)} \\
&= \sum_{e=1}^{E} (D_e^{(1)} - L_e^{(1)} - U_e^{(1)}) \otimes (\bigotimes_{i=2}^{N} Q_e^{(i)}) \\
&= \sum_{e=1}^{E} \left[\left[D_e^{(1)} \otimes (\bigotimes_{i=2}^{N} Q_e^{(i)}) \right] - \left[L_e^{(1)} \otimes (\bigotimes_{i=2}^{N} Q_e^{(i)}) \right] - \left[U_e^{(1)} \otimes (\bigotimes_{i=2}^{N} Q_e^{(i)}) \right] \right] \\
&= \sum_{e=1}^{E} \left[D_e^{(1)} \otimes (\bigotimes_{i=2}^{N} Q_e^{(i)}) \right] - \sum_{e=1}^{E} \left[L_e^{(1)} \otimes (\bigotimes_{i=2}^{N} Q_e^{(i)}) \right] - \sum_{e=1}^{E} \left[U_e^{(1)} \otimes (\bigotimes_{i=2}^{N} Q_e^{(i)}) \right] \\
&= \sum_{e=1}^{E} \left[D_e^{(1)} \otimes (D_e^{(2)} - L_e^{(2)} - U_e^{(2)}) \otimes (\bigotimes_{i=3}^{N} Q_e^{(i)}) \right] - \sum_{e=1}^{E} \left[L_e^{(1)} \otimes (\bigotimes_{i=2}^{N} Q_e^{(i)}) \right]
\end{aligned}$$

$$
\begin{aligned}
& - \sum_{e=1}^{E} \left[U_e^{(1)} \otimes (\bigotimes_{i=3}^{N} Q_e^{(i)}) \right] \\
= & \sum_{e=1}^{E} \left[D_e^{(1)} \otimes D_e^{(2)} \otimes (\bigotimes_{i=3}^{N} Q_e^{(i)}) \right] - \sum_{e=1}^{E} \left[D_e^{(1)} \otimes L_e^{(2)} \otimes (\bigotimes_{i=3}^{N} Q_e^{(i)}) \right] \\
& - \sum_{e=1}^{E} \left[D_e^{(1)} \otimes U_e^{(2)} \otimes (\bigotimes_{i=3}^{N} Q_e^{(i)}) \right] - \sum_{e=1}^{E} \left[L_e^{(1)} \otimes (\bigotimes_{i=2}^{N} Q_e^{(i)}) \right] \\
& - \sum_{e=1}^{E} \left[U_e^{(1)} \otimes (\bigotimes_{i=3}^{N} Q_e^{(i)}) \right] \\
= & \cdots \\
= & \sum_{e=1}^{E} (\bigotimes_{i=1}^{N} D_e^{(i)}) - \sum_{e=1}^{E} \sum_{k=1}^{N} \left[(\bigotimes_{i=1}^{k-1} D_e^{(i)}) \otimes L_e^{(k)} \otimes (\bigotimes_{i=k+1}^{N} Q_e^{(i)}) \right] \\
& - \sum_{e=1}^{E} \sum_{k=1}^{N} \left[(\bigotimes_{i=1}^{k-1} D_e^{(i)}) \otimes U_e^{(k)} \otimes (\bigotimes_{i=k+1}^{N} Q_e^{(i)}) \right] \\
= & D_e - L_e - U_e
\end{aligned}
$$

The last equality is a consequence of Lemmas 4.1.1, 4.1.2, 4.1.3, 4.1.4, and 4.1.5. □

Theorem 4.1.9 *The descriptor of a SAN given by $Q(= Q_l + Q_e + \bar{Q}_e)$ can be split as $Q = D - L - U$, where D is diagonal, L is strictly lower triangular, and U is strictly upper triangular. In particular*

$$
\begin{aligned}
Q &= Q_l + Q_e + \bar{Q}_e \\
&= (D_l - L_l - U_l) + (D_e - L_e - U_e) + \bar{Q}_e \\
&= \underbrace{(D_l + D_e + \bar{Q}_e)}_{D} - \underbrace{(L_l + L_e)}_{L} - \underbrace{(U_l + U_e)}_{U}.
\end{aligned}
$$

Moreover, $D, L,$ and U each may be written in the form of a sum of tensor products.

Proof. The proof of Theorem 4.1.9 follows from Lemmas 4.1.6, 4.1.7, and 4.1.8. □

4.1.1 An Example Splitting

The following example, from Chapter 3 better illustrates the concept of splitting a SAN descriptor. Note that for the sake of simplicity, we replaced the functional entries in the second automaton with constant values. It is composed of two automata and two synchronizing events (i.e., $N = E = 2$) with $n_1 = 2$, $n_2 = 3$. For the first automaton, we have

$$Q_l^{(1)} = \begin{bmatrix} -\lambda_1 & \lambda_1 \\ 0 & 0 \end{bmatrix},$$

$$Q_{e_1}^{(1)} = \begin{bmatrix} 0 & 0 \\ \lambda_2 & 0 \end{bmatrix}, \bar{Q}_{e_1}^{(1)} = \begin{bmatrix} 0 & 0 \\ 0 & -\lambda_2 \end{bmatrix},$$

$$Q_{e_2}^{(1)} = \begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix}, \bar{Q}_{e_2}^{(1)} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

For the second automaton, we have

$$Q_l^{(2)} = \begin{bmatrix} -\mu_1 & \mu_1 & 0 \\ 0 & -\mu_2 & \mu_2 \\ 0 & 0 & 0 \end{bmatrix},$$

$$Q_{e_1}^{(2)} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}, \bar{Q}_{e_1}^{(2)} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix},$$

$$Q_{e_2}^{(2)} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ \mu_3 & 0 & 0 \end{bmatrix}, \bar{Q}_{e_2}^{(2)} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & -\mu_3 \end{bmatrix}.$$

The global generator of the example SAN is given by

$$\begin{aligned} Q &= Q_l + Q_e + \bar{Q}_e \\ &= \bigoplus_{i=1}^{N} Q_l^{(i)} + \sum_{e=1}^{E} \bigotimes_{i=1}^{N} Q_e^{(i)} + \sum_{e=1}^{E} \bigotimes_{i=1}^{N} \bar{Q}_e^{(i)} \\ &= Q_l^{(1)} \oplus Q_l^{(2)} + Q_{e_1}^{(1)} \otimes Q_{e_1}^{(2)} + Q_{e_2}^{(1)} \otimes Q_{e_2}^{(2)} + \bar{Q}_{e_1}^{(1)} \otimes \bar{Q}_{e_1}^{(2)} + \bar{Q}_{e_2}^{(1)} \otimes \bar{Q}_{e_2}^{(2)}. \end{aligned}$$

Hence Q is a matrix of order 6, i.e.,

$$\left[\begin{array}{ccc|ccc} -(\lambda_1+\mu_1) & \mu_1 & 0 & \lambda_1 & 0 & 0 \\ 0 & -(\lambda_1+\mu_2) & \mu_2 & 0 & \lambda_1 & 0 \\ \mu_3 & 0 & -(\lambda_1+\mu_3) & 0 & 0 & \lambda_1 \\ \hline \lambda_2 & 0 & 0 & -(\lambda_2+\mu_1) & \mu_1 & 0 \\ \lambda_2 & 0 & 0 & 0 & -(\lambda_2+\mu_2) & \mu_2 \\ \lambda_2+\mu_3 & 0 & 0 & 0 & 0 & -(\lambda_2+\mu_3) \end{array}\right]. \quad (2)$$

Due to Theorem 4.1.9, we have

$$\begin{aligned} Q &= D - L - U \\ &= (D_l + D_e + \bar{Q}_e) - (L_l + L_e) - (U_l + U_e), \end{aligned}$$

where D_l, L_l, U_l are obtained from Lemma 4.1.7 and D_e, L_e, U_e are obtained from Lemma 4.1.8. As before, we use $I_{n_i:n_j}$ to represent an identity matrix of size $\prod_{k=i}^{j} n_k$ when $i \le j$, else a one. I_k and 0_k are identity and zero matrices of order k, respectively. Then from all the lemmas, we have

$$\begin{aligned} D &= D_l + D_e + \bar{Q}_e \\ &= \sum_{i=1}^{N}(I_{n_1:n_{i-1}} \otimes D_l^{(i)} \otimes I_{n_{i+1}:n_N}) + \sum_{e=1}^{E}\bigotimes_{i=1}^{N} D_e^{(i)} + \sum_{e=1}^{E}\bigotimes_{i=1}^{N} \bar{Q}_e^{(i)} \\ &= D_l^{(1)} \otimes I_3 + I_2 \otimes D_l^{(2)} + D_{e_1}^{(1)} \otimes D_{e_1}^{(2)} + D_{e_2}^{(1)} \otimes D_{e_2}^{(2)} + \bar{Q}_{e_1}^{(1)} \otimes \bar{Q}_{e_1}^{(2)} \\ &\quad + \bar{Q}_{e_2}^{(1)} \otimes \bar{Q}_{e_2}^{(2)} \\ &= \begin{bmatrix} -\lambda_1 & 0 \\ 0 & 0 \end{bmatrix} \otimes I_3 + I_2 \otimes \begin{bmatrix} -\mu_1 & 0 & 0 \\ 0 & -\mu_2 & 0 \\ 0 & 0 & 0 \end{bmatrix} + 0_2 \otimes \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \\ &\quad + \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \otimes 0_3 + \begin{bmatrix} 0 & 0 \\ 0 & -\lambda_2 \end{bmatrix} \otimes I_3 + I_2 \otimes \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & -\mu_3 \end{bmatrix}. \\ &= \left[\begin{array}{ccc|ccc} -\lambda_1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -\lambda_1 & 0 & 0 & 0 & 0 \\ 0 & 0 & -\lambda_1 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right] + \left[\begin{array}{ccc|ccc} -\mu_1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -\mu_2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & -\mu_1 & 0 & 0 \\ 0 & 0 & 0 & 0 & -\mu_2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right] \\ &\quad + 0_6 + 0_6 \end{aligned}$$

$$
+\left[\begin{array}{ccc|ccc}
0 & 0 & 0 & 0 & 0 & 0\\
0 & 0 & 0 & 0 & 0 & 0\\
0 & 0 & 0 & 0 & 0 & 0\\
\hline
0 & 0 & 0 & -\lambda_2 & 0 & 0\\
0 & 0 & 0 & 0 & -\lambda_2 & 0\\
0 & 0 & 0 & 0 & 0 & -\lambda_2
\end{array}\right]
+\left[\begin{array}{ccc|ccc}
0 & 0 & 0 & 0 & 0 & 0\\
0 & 0 & 0 & 0 & 0 & 0\\
0 & 0 & -\mu_3 & 0 & 0 & 0\\
\hline
0 & 0 & 0 & 0 & 0 & 0\\
0 & 0 & 0 & 0 & 0 & 0\\
0 & 0 & 0 & 0 & 0 & -\mu_3
\end{array}\right]
$$

$$
= \left[\begin{array}{ccc|ccc}
-(\lambda_1+\mu_1) & 0 & 0 & 0 & 0 & 0\\
0 & -(\lambda_1+\mu_2) & 0 & 0 & 0 & 0\\
0 & 0 & -(\lambda_1+\mu_3) & 0 & 0 & 0\\
\hline
0 & 0 & 0 & -(\lambda_2+\mu_1) & 0 & 0\\
0 & 0 & 0 & 0 & -(\lambda_2+\mu_2) & 0\\
0 & 0 & 0 & 0 & 0 & -(\lambda_2+\mu_3)
\end{array}\right].
$$

For L, we have

$$
\begin{aligned}
L &= L_l + L_e\\
&= \sum_{i=1}^{N}(I_{n_1:n_{i-1}} \otimes L_l^{(i)} \otimes I_{n_{i+1}:n_N}) + \sum_{e=1}^{E}\sum_{k=1}^{N}(\bigotimes_{i=1}^{k-1} D_e^{(i)}) \otimes L_e^{(k)} \otimes (\bigotimes_{i=k+1}^{N} Q_e^{(i)})\\
&= L_l^{(1)} \otimes I_3 + I_2 \otimes L_l^{(2)} + L_{e_1}^{(1)} \otimes Q_{e_1}^{(2)} + D_{e_1}^{(1)} \otimes L_{e_1}^{(2)} + L_{e_2}^{(1)} \otimes Q_{e_2}^{(2)}\\
&\quad + D_{e_2}^{(1)} \otimes L_{e_2}^{(2)}\\
&= 0_2 \otimes I_3 + I_2 \otimes 0_3 + \begin{bmatrix} 0 & 0\\ -\lambda_2 & 0 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 & 0\\ 1 & 0 & 0\\ 1 & 0 & 0 \end{bmatrix} + 0_2 \otimes \begin{bmatrix} 0 & 0 & 0\\ -1 & 0 & 0\\ -1 & 0 & 0 \end{bmatrix}\\
&\quad + \begin{bmatrix} 0 & 0\\ -1 & 0 \end{bmatrix} \otimes \begin{bmatrix} 0 & 0 & 0\\ 0 & 0 & 0\\ \mu_3 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 1 & 0\\ 0 & 0 \end{bmatrix} \otimes \begin{bmatrix} 0 & 0 & 0\\ 0 & 0 & 0\\ -\mu_3 & 0 & 0 \end{bmatrix}\\
&= 0_6 + 0_6 + \left[\begin{array}{ccc|ccc}
0 & 0 & 0 & 0 & 0 & 0\\
0 & 0 & 0 & 0 & 0 & 0\\
0 & 0 & 0 & 0 & 0 & 0\\
\hline
-\lambda_2 & 0 & 0 & 0 & 0 & 0\\
-\lambda_2 & 0 & 0 & 0 & 0 & 0\\
-\lambda_2 & 0 & 0 & 0 & 0 & 0
\end{array}\right] + 0_6 + \left[\begin{array}{ccc|ccc}
0 & 0 & 0 & 0 & 0 & 0\\
0 & 0 & 0 & 0 & 0 & 0\\
0 & 0 & 0 & 0 & 0 & 0\\
\hline
0 & 0 & 0 & 0 & 0 & 0\\
0 & 0 & 0 & 0 & 0 & 0\\
-\mu_3 & 0 & 0 & 0 & 0 & 0
\end{array}\right]
\end{aligned}
$$

$$+\left[\begin{array}{ccc|ccc} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ -\mu_3 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right] = \left[\begin{array}{ccc|ccc} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ -\mu_3 & 0 & 0 & 0 & 0 & 0 \\ \hline -\lambda_2 & 0 & 0 & 0 & 0 & 0 \\ -\lambda_2 & 0 & 0 & 0 & 0 & 0 \\ -(\lambda_2+\mu_3) & 0 & 0 & 0 & 0 & 0 \end{array}\right].$$

Finally for U, we have

$$\begin{aligned} U &= U_l + U_e \\ &= \sum_{i=1}^{N}(I_{n_1:n_{i-1}} \otimes U_l^{(i)} \otimes I_{n_{i+1}:n_N}) + \sum_{e=1}^{E}\sum_{k=1}^{N}(\bigotimes_{i=1}^{k-1} D_e^{(i)}) \otimes U_e^{(k)} \otimes (\bigotimes_{i=k+1}^{N} Q_e^{(i)}) \\ &= U_l^{(1)} \otimes I_3 + I_2 \otimes U_l^{(2)} + U_{e_1}^{(1)} \otimes Q_{e_1}^{(2)} + D_{e_1}^{(1)} \otimes U_{e_1}^{(2)} + U_{e_2}^{(1)} \otimes Q_{e_2}^{(2)} + \\ &\quad D_{e_2}^{(1)} \otimes U_{e_2}^{(2)} \\ &= \begin{bmatrix} 0 & -\lambda_1 \\ 0 & 0 \end{bmatrix} \otimes I_3 + I_2 \otimes \begin{bmatrix} 0 & -\mu_1 & 0 \\ 0 & 0 & -\mu_2 \\ 0 & 0 & 0 \end{bmatrix} + 0_2 \otimes \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix} \\ &\quad + 0_2 \otimes 0_3 + 0_2 \otimes \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ \mu_3 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \otimes 0_3 \\ &= \left[\begin{array}{ccc|ccc} 0 & 0 & 0 & -\lambda_1 & 0 & 0 \\ 0 & 0 & 0 & 0 & -\lambda_1 & 0 \\ 0 & 0 & 0 & 0 & 0 & -\lambda_1 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right] + \left[\begin{array}{ccc|ccc} 0 & -\mu_1 & 0 & 0 & 0 & 0 \\ 0 & 0 & -\mu_2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & -\mu_1 & 0 \\ 0 & 0 & 0 & 0 & 0 & -\mu_2 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right] \\ &\quad + 0_6 + 0_6 + 0_6 + 0_6 \\ &= \left[\begin{array}{ccc|ccc} 0 & -\mu_1 & 0 & -\lambda_1 & 0 & 0 \\ 0 & 0 & -\mu_2 & 0 & -\lambda_1 & 0 \\ 0 & 0 & 0 & 0 & 0 & -\lambda_1 \\ \hline 0 & 0 & 0 & 0 & -\mu_1 & 0 \\ 0 & 0 & 0 & 0 & 0 & -\mu_2 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right]. \end{aligned}$$

The global generator matrix given in 2 may be verified by computing $D - L - U$. In the next section, we present three iterative methods that follow from the splitting in Theorem 4.1.9.

4.2 Iterative Methods Based on Splittings

Remember that the problem of finding the stationary vector of a Markov chain may be formulated as one of computing a nontrivial solution to a homogeneous system of linear algebraic equations with a singular coefficient matrix under a normalization constraint. That is, the $(1 \times n)$ unknown vector π in

$$\pi Q = 0, \ \ \|\pi\|_1 = 1 \tag{3}$$

is sought. The methods based on splittings amount to using the power method with an iteration matrix that corresponds to the particular splitting until a predetermined stopping criterion is met. We should also remark that the efficient vector–(generalized) tensor product multiplication algorithm used by the methods of interest has a time complexity of order $O(\prod_{i=1}^N n_i \sum_{i=1}^N n_i)$. This complexity result assumes that all matrices in a tensor product are dense. In reality, some of these matrices are identity and zero, some are diagonal, and the remaining sparse. See, for instance, the matrices forming the descriptor in the example in Subsection 4.1.1.

In the following subsections, we introduce the stationary iterative methods of Jacobi, Gauss–Seidel and SOR that we described in Chapter 2. In Section 4.3, we describe the block versions of the same methods.

4.2.1 Jacobi

In matrix notation, applying the Jacobi method to a homogeneous linear system as in (3) is equivalent to applying the power method to the iteration matrix $(L+U)D^{-1}$; that is,

$$\pi^{(k+1)} = \pi^{(k)}(L+U)D^{-1}, \quad k = 0, 1, \ldots,$$

where Q is split as $D - (L+U)$. As it can be seen from the given formulation, each iteration may be implemented in two steps. First, postmultiply the most recent approximation $\pi^{(k)}$ with $(L+U)$, which is a sum of tensor products, and obtain $y^{(k)}$. Then postmultiply $y^{(k)}$ with D^{-1}. This last step can be implemented by multiplying the reciprocal of each diagonal element in D with the corresponding element of $y^{(k)}$ to give $\pi^{(k+1)}$.

4.2.2 Gauss–Seidel

In matrix notation, applying GS to a homogeneous system as in (3) is equivalent to applying the power method to the iteration matrix $U(D-L)^{-1}$. However, in order to employ the efficient vector–tensor product multiplication algorithm, we propose a slightly different implementation of the method. A backward GS iteration corresponds to the splitting $Q=(D-L)-U$ and may be written as

$$\pi^{(k+1)}(D-L) \quad = \quad \pi^{(k)}U, \quad k=0,1,\ldots.$$

The right hand side of the iteration requires the use of vector–tensor product multiplication. Once the right hand side is computed as $b^{(k)}$, the next step involves solving the lower triangular system of equations $\pi^{(k+1)}(D-L)=b^{(k)}$. Similarly one can define forward GS using the splitting $Q=(D-U)-L$. In order to employ the efficient vector–tensor product multiplication algorithm, we should examine the nonzero structure of the matrix $(D-L)$.

D is a diagonal matrix of order $n=\prod_{i=1}^{N} n_i$ from which all the diagonal elements of $(D-L)$ come. That is, none of the nonzero elements of L, a strictly lower triangular matrix, appear along the diagonal of $(D-L)$.

By considering Lemmas 4.1.7, 4.1.8 and relabeling L_l as $L_{e=0}$, we can rewrite L as

$$\begin{aligned}
L &= \sum_{i=1}^{N} I_{n_1:n_{i-1}} \otimes L_l^{(i)} \otimes I_{n_{i+1}:n_N} + \sum_{e=1}^{E}\sum_{k=1}^{N}(\bigotimes_{i=1}^{k-1} D_e^{(i)}) \otimes L_e^{(k)} \otimes (\bigotimes_{i=k+1}^{N} Q_e^{(i)}) \\
&= \sum_{e=0}^{E}\left[\sum_{k=1}^{N}(\bigotimes_{i=1}^{k-1} D_e^{(i)}) \otimes L_e^{(k)} \otimes (\bigotimes_{i=k+1}^{N} Q_e^{(i)})\right] \\
&= \sum_{e=0}^{E} L_e^{(1)}(\bigotimes_{i=2}^{N} Q_e^{(i)}) + \sum_{e=0}^{E} D_e^{(1)}\left[\sum_{k=2}^{N}(\bigotimes_{i=2}^{k-1} D_e^{(i)}) \otimes L_e^{(k)} \otimes (\bigotimes_{i=k+1}^{N} Q_e^{(i)})\right] \\
&= \sum_{e=0}^{E} Q_e^L,
\end{aligned}$$

where all Q_e^L are strictly lower triangular matrices formed by summing similar tensor products. For Q_0^L (i.e., L_l), all matrices except $L_0^{(i)}$ in the tensor products are identity matrices.

Similarly, using Lemmas 4.1.6, 4.1.7, 4.1.8 and relabeling $\bar{Q}_e^{(i)}$ as $D_{e+E}^{(i)}$ for $e = 1, 2, \ldots, E$, we get

$$\begin{aligned} D &= \sum_{i=1}^{N} I_{n_1:n_{i-1}} \otimes D_l^{(i)} \otimes I_{n_{i+1}:n_N} + \sum_{e=1}^{E} \bigotimes_{i=1}^{N} D_e^{(i)} + \sum_{e=1}^{E} \bigotimes_{i=1}^{N} \bar{Q}_e^{(i)} \\ &= \sum_{i=1}^{N} I_{n_1:n_{i-1}} \otimes D_l^{(i)} \otimes I_{n_{i+1}:n_N} + \sum_{e=1}^{2E} \bigotimes_{i=1}^{N} D_e^{(i)}. \end{aligned}$$

Next we expose the block structure of $(D - L)$ and build the lower triangular solution on this structure.

Each Q_e^L matrix is the sum of N tensor products. All tensor products in this summation introduce nonzero entries to Q_e^L that are in mutually exclusive locations. In other words, each nonzero element in Q_e^L comes from a different tensor product. To see this, partition Q_e^L into n_1 blocks each of order $\prod_{i=2}^{N} n_i$. Its lower triangular blocks come from the term $L_e^{(1)} \otimes Q_e^{(2)} \otimes \cdots \otimes Q_e^{(N)}$ and its diagonal blocks come from the remaining terms (i.e., terms that have $D_e^{(1)}$ as the first factor). Observe that block (i, j) $i > j$ of Q_e^L can be expressed as $l_{e(i,j)}^{(1)}(\bigotimes_{k=2}^{N} Q_e^{(k)})$, where $l_{e(i,j)}^{(1)}$ is the (i, j)th element of $L_e^{(1)}$. Similarly, block (j, j) of Q_e^L can be expressed as $d_{e(j,j)}^{(1)} \left[\sum_{k=2}^{N}(\bigotimes_{i=2}^{k-1} D_e^{(i)}) \otimes L_e^{(k)} \otimes (\bigotimes_{i=k+1}^{N} Q_e^{(i)})\right]$, where $d_{e(j,j)}^{(1)}$ is the jth diagonal element of $D_e^{(1)}$.

Given the above (first level) partitioning of L, our algorithm for solving π in the system $\pi(D - L) = b$ stems from the following observation. The linear equations for the subvector of π corresponding to the jth diagonal block of $(D - L)$, denoted $\bar{\pi}_j$, can be expressed as

$$\bar{\pi}_j D_{j,j} = \bar{b}_j + \sum_{i=j+1}^{n_1} \bar{\pi}_i \left[\sum_{e=0}^{E} l_{e(i,j)}^{(1)}(\bigotimes_{k=2}^{N} Q_e^{(k)})\right], \qquad (4)$$

or as

$$\bar{\pi}_j D_{j,j} = \bar{c}_j, \quad j = n_1, \ldots, 2, 1.$$

Here $D_{j,j}$ is the jth diagonal block of $(D - L)$, $\bar{b}_j$ and $\bar{c}_j$ are respectively the jth subvectors of b and c, the new right hand side.

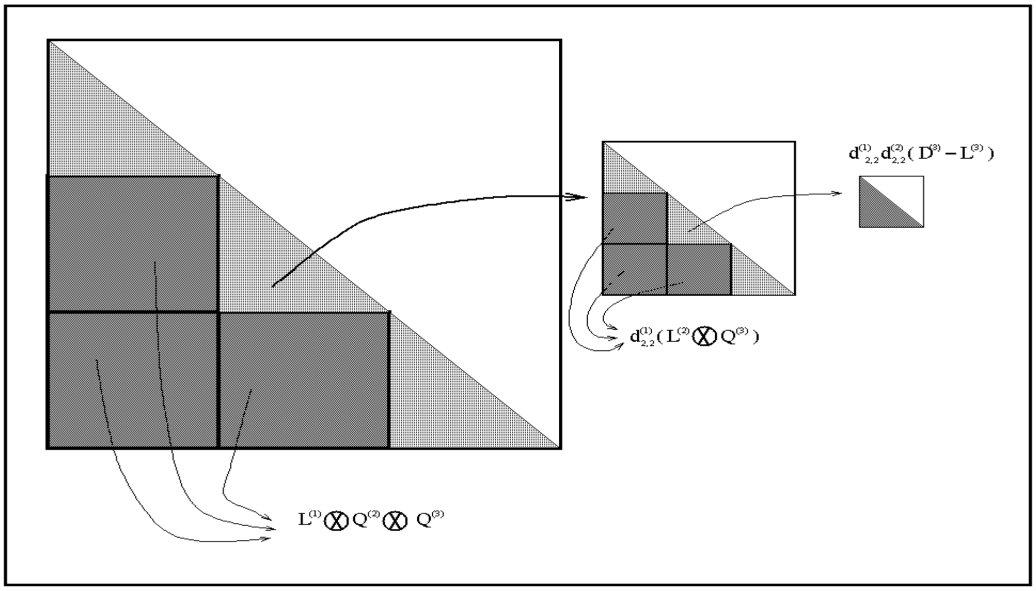

Figure 4.1: Lower triangular part of $Q_1 \otimes Q_2 \otimes Q_3$ partitioned into blocks.

At this point, we are left with the problem of solving $\bar{\pi}_j D_{j,j} = \bar{c}_j$. Fortunately, the block structure of the diagonal blocks $D_{j,j}$ is similar to that of the original matrix $(D - L)$. Each diagonal block at level 1 is a lower triangular matrix that can be expressed as a sum of tensor products. Thus,

$$\begin{aligned} D_{j,j} \;=\; & \left[d^{(1)}_{l(j,j)} \otimes I_{n_2:n_N} + \sum_{k=2}^{N} I_{n_2:n_{k-1}} \otimes D^{(k)}_l \otimes I_{n_{k+1}:n_N} \right] + \sum_{e=1}^{2E} d^{(1)}_{e(j,j)} \left(\bigotimes_{k=2}^{N} D^{(k)}_e \right) \\ & - \sum_{e=0}^{E} d^{(1)}_{e(j,j)} \left[\sum_{k=2}^{N} \left(\bigotimes_{i=2}^{k-1} D^{(i)}_e \right) \otimes L^{(k)}_e \otimes \left(\bigotimes_{i=k+1}^{N} Q^{(i)}_e \right) \right], \end{aligned}$$

where $d^{(1)}_{l(j,j)}$ is the jth diagonal element of $D^{(1)}_l$. Note that the diagonal elements of $D_{j,j}$ come from the first and the second terms. The strictly lower triangular elements come from the third term. Next we can partition each diagonal block $D_{j,j}$ into n_2 blocks each of order $\prod_{i=3}^{N} n_i$. This continues recursively until we have a system of order n_N (i.e., order of the last automaton) to solve. The first and the second terms of $D_{j,j}$ come into play only at the deepest level and the recursion is inherent in the third term. Hence, the algorithm we present for point GS is a recursive one. The lower triangular solution algorithm calls itself until the recursion ends at level N when a single iteration over the point equations is performed: the systems to be solved at level N are lower triangular.

The illustrative example in Figure 4.1 shows the partitioning of a three term tensor product. The lower triangular block structure of the tensor product $Q_1 \otimes Q_2 \otimes Q_3$ is emphasized. The dark grey shaded blocks of the product on the left come from the term $L_1 \otimes Q_2 \otimes Q_3$. The grey blocks on the left correspond to the three diagonal blocks each of order $n_2 n_3$. The partitioning of the second diagonal block $D_{2,2}$ is shown in the middle. The smallest matrix on the right is the second diagonal block of $D_{2,2}$.

Algorithm for solving $\pi(D - L) = b$

The algorithm discussed in this section solves the system $\pi(D - L) = b$ using the efficient vector–(generalized) tensor product multiplication algorithm when there are no *cyclic dependencies* in the SAN [5, pp. 20–22]. Here, D and L are respectively diagonal and strictly lower triangular matrices. In the absence of cyclic dependencies, all tensor products in a SAN (see equation (1)) may be ordered (and relabeled) such that each matrix in each tensor product has entries with functional dependencies, if at all, only to the automata that come before itself in the given ordering. A SAN that lacks cyclic dependencies may be written in the form $Q^{(1)}$,$Q^{(2)}[\mathcal{Q}^{(1)}]$,$Q^{(3)}[\mathcal{Q}^{(1)}, \mathcal{Q}^{(2)}]$,..., $Q^{(N)}[\mathcal{Q}^{(1)}, \ldots, \mathcal{Q}^{(N-1)}]$. Remember that the arguments in the square brackets of each matrix indicate dependencies that may exist among automata. For instance, transitions in automata 3 may depend only on the states of automata 1 and 2, but not on the states of others. Before we use the algorithm, we make sure the automata are ordered appropriately.

The initial call to the recursive algorithm is SolveD-L$(1, states, n, \pi, b)$. The first parameter $id (= 1)$ corresponds to the level of block partitioning. It might also be thought of as the current level automaton number. The initial call at level 1 partitions the global descriptor into n_1 blocks each of order $\prod_{i=2}^{N} n_i$. The second parameter *states*, an array of size N, stores the state of each automaton to be used in function evaluations. For instance, if we are solving the ith diagonal block (see equation (4)) in the first call (i.e., no recursive calls have been made yet), the state of automaton 1 is i. The parameter *states* is also used to determine the scalar multipliers that form the diagonal blocks. For example, in order to solve the smallest block in Figure 4.1, we need to mul-

SolveD-L$(id, states, first, \pi, b)$
1. $nright = n_{id+1} n_{id+2} \dots n_N$
2. if $(id = N)$
    - $T = 0$
    - for $e = 1$ to $2E + N$
        - $d = \prod_{i=1}^{id-1} d^{(i)}_{e(states[i],states[i])}[states]$
        - $T = T + d(D_e^{(id)}[states] - L_e^{(id)}[states])$
    - Solve $\pi_{first-n_N+1:n_N} T = b_{first-n_N+1:n_N}$
    - return

    else
    - $states[id] = n_{id}$
    - SolveD-L$(id+1, states, first, \pi, b)$
3. $\pi first = first$
4. $bfirst = first - (n_{id} nright) + 1$
5. for $irow = n_{id}$ downto 2
    - $\pi first = \pi first - nright + 1$
    - $states[id] = irow$
    - for $e = 0$ to E
        - $b' = \pi_{\pi first:nright}(\bigotimes_{i=id+1}^{N} Q_e^{(i)}[states])$
        - for $k = 1$ to $irow - 1$
            - ◇ Reset $states[(id+1), \dots, N]$ to the first indexed states of automata $(id+1)$ to N
            - ◇ for $i = 1$ to $nright$
                - ⋆ $d = \prod_{j=1}^{id-1} d^{(j)}_{e(states[j],states[j])}$
                - ⋆ $b_{bfirst+(k-1)nright+i-1} = b_{bfirst+(k-1)nright+i-1} + (d \cdot l^{(id)}_{e(irow,k)}[states] b'_i)$
                - ⋆ Update $states[(id+1), \dots, N]$ for automata $(id+1)$ to N
    - SolveD-L$(id+1, states, \pi first - nright + 1, \pi, b)$

Figure 4.2: The recursive lower triangular solution algorithm for SANs

tiply the lower triangular matrix $(D^{(3)} - L^{(3)})$ with $d^{(1)}_{2,2} d^{(2)}_{2,2}$. See also step 2 in the SolveD-L algorithm; if e corresponds to the corrector of a synchronizing event, $L_e^{(id)}[states] = 0$. Furthermore, we represent matrices arising from local automata by $e = 2E+1, \dots, 2E+N$ in step 2. We determine both the coordinates of the scalar multipliers and the current states of lower indexed automata using $states$. The initial contents of $states$ is irrelevant since it is updated when deemed necessary. The third parameter $first(= n)$ is set to the size of the unknown vector in the current call. The fourth parameter is the solution vector initially set to $\pi_i = 1/n \ \forall i$ and overwritten with the new approximation at each iteration. The last parameter b is the right hand side of the lower triangular solution. The algorithm assumes the generator matrices of automata are available globally. Since the algorithm implements a back solution and computes the last unknown (subvector) first, we use $\pi_{first:n_N}$ to denote the subvector of π with first element π_{first} and length n_N.

Vector–tensor product multiplications arising from the local and synchronizing event generator matrices (see the for–loop on e in step 5 of the algorithm) may be reduced to scalar–vector multiplications (see the third statement from the bottom in step 5). For each block in a row, a vector–tensor product multiplication possibly with functional transitions depending on the current state of the automata at that level (see *irow* in step 5) is required. An efficient approach is to loop on blocks in a row (see the for–loop on k in step 5) because in each row all blocks below the diagonal have a common vector–tensor product multiplication and all functional entries in these blocks use the same *irow* value while being evaluated (see equation (4)). It is also observed that many matrices encountered in the test problems are zero, have zero diagonals, have zero strictly lower or strictly upper triangular parts. We have taken advantage of this as well. The actual timings depend heavily on such implementation details.

Gauss–Seidel algorithm

The algorithm given in Figure 4.3 implements Gauss–Seidel for solving a SAN in the functional case assuming that a splitting $(D - L - U)$ for the SAN descriptor and an initial approximation π are available. Remember that the triangular solution overwrites the input approximation with the new approximation on return from the call. Upon termination *it* gives the number of iterations performed.

- $it = 0$
- Repeat until convergence
  - ∘ $it = it + 1$
  - ∘ Compute $b = \pi U$
  - ∘ SolveD-L$(1, states, n, \pi, b)$

Figure 4.3: The Gauss–Seidel algorithm using SolveD-L

4.2.3 Successive Overrelaxation

We now express the SOR method as $\pi_{SOR}^{(k+1)} := w\pi_{GS}^{(k+1)} + (1-w)\pi_{SOR}^{(k)}$, where $\pi_{GS}^{(k+1)}$ is the $(k+1)$st approximation of GS, $\pi_{SOR}^{(k)}$ is the kth approximation of SOR, and w is the relaxation parameter (i.e., a weighing factor) satisfying $0 < w < 2$.

4.3 Block Methods

We argued that one can perform a lower triangular back solution on the blocks of order n_N at the final depth of recursion: see the third bullet in step 2 of the SolveD-L algorithm. Instead of doing this, one may choose to solve these blocks directly, i.e., by Gaussian elimination (GE). This approach, we call block GS, is likely to decrease the number of iterations since blocks at each iteration are solved exactly. When doing this, the right hand side b that goes into SolveD-L is computed in a slightly different manner. Now one must exclude the strictly upper triangular parts of the matrices corresponding to the last automaton from the multiplication. That is,

$$\begin{aligned} b^{(k)} \;=\; & \pi^{(k)} \Bigg[\sum_{i=1}^{N-1} I_{n_1:n_{i-1}} \otimes U_l^{(i)} \otimes I_{n_{i+1}:n_N} + \\ & \sum_{e=1}^{E} \sum_{k=1}^{N-1} \Big(\bigotimes_{i=1}^{k-1} D_e^{(i)} \Big) \otimes U_e^{(k)} \otimes \Big(\bigotimes_{i=k+1}^{N} Q_e^{(i)} \Big) \Bigg]. \end{aligned}$$

What has been excluded from the new right hand side must be included at level N in step 2 of the recursive back solution algorithm. The matrix that corresponds to automaton N at step 2 must include the whole matrices that correspond to synchronizing events, their diagonal correctors and to local automata, not just the lower triangular parts. The matrices of order n_N formed in this way at the deepest level of recursion for each one of the $\prod_{i=1}^{N-1} n_i$ diagonal blocks will be solved using GE. Even though the space requirement is larger, if the decrease in the iteration count is substantial the cost of solving the blocks directly is offset by a smaller overall solution time. Another possibility is to terminate recursion earlier and solve larger blocks. Also one can choose to generate and store larger blocks at the outset, then use these at each iteration (see the concept of *grouping* in [6, pp. 13–14]).

In the experiments, we noticed an interesting feature of block methods.

Remark 4.3.1 *For a block coefficient matrix with lower (upper) triangular diagonal blocks in equation (3), backward (forward) block GS/SOR is equivalent to point GS/SOR.*

The remark follows from inspecting the linear equations in systems with the described nonzero structure.

4.4 An Upper Bound on SolveD-L

In this section, we provide an upper bound on the number of multiplications performed in the SolveD-L algorithm for point GS (see subsection 4.2.2). Remember that multiplying the approximate subvector $\bar{\pi}_j$ with block (j, i) $j > i$ of the descriptor at the first level partitioning can be expressed as $l^{(1)}_{e(j,i)} \bar{\pi}_j (\bigotimes_{k=2}^{N} Q_e^{(k)})$. If the row index j in this expression changes, the product $\bar{\pi}_j (\bigotimes_{k=2}^{N} Q_e^{(k)})$ should be reevaluated for each value of j in case there are functional dependencies among automata. At worst, the value of the functional rate remains constant for all blocks in the same row. We use the efficient vector–(generalized) tensor product multiplication algorithm that has a time complexity of $O(\prod_{i=1}^{N} n_i \sum_{i=1}^{N} n_i)$ for a tensor product with N matrices each of order n_i. This complexity result assumes that all matrices that participate in the multiplication are dense.

In the following, T_i represents the number of multiplications performed in SolveD-L when the matrix to be solved is partitioned into n_i blocks each of order $\prod_{j=i+1}^{N} n_j$.

$$\begin{aligned} T_i &= E\left[(n_i-1)\prod_{j=i+1}^{N} n_j \sum_{j=i+1}^{N} n_j + i\frac{n_i^2-n_i}{2}\prod_{j=i+1}^{N} n_j\right] \\ &\quad +\frac{n_i^2-n_i}{2}\prod_{j=i+1}^{N} n_j + n_i T_{i+1} \qquad\qquad \text{for } i<N, \\ T_N &= E\left[N\frac{n_N(n_N-1)}{2}\right] + \frac{n_N(n_N-1)}{2} + ENn_N + n_N. \end{aligned}$$

The initial call to SolveD-L views the global matrix as partitioned into n_1 blocks each of order $\prod_{i=2}^{N} n_i$. We aim at bounding T_1 given by

$$T_1 = E\left[(n_1-1)\prod_{i=2}^{N} n_i \sum_{i=2}^{N} n_i + \frac{n_1^2-n_1}{2}\prod_{i=2}^{N} n_i\right] + \frac{n_1^2-n_1}{2}\prod_{i=2}^{N} n_i + n_1 T_2.$$

The last term n_1T_2 of T_1 means in the next call we solve n_1 diagonal blocks of order $\prod_{i=2}^{N} n_i$ recursively. The term that is inside the E parentheses arises from the multiplication of the current approximate subvector with tensor products corresponding to E synchronizing events. The first term $(n_1-1)\prod_{i=2}^{N} n_i \sum_{i=2}^{N} n_i$ inside the parentheses is for the multiplication of the current approximate subvector with all blocks below the diagonal due to a synchronizing event. Remember that for each row of blocks all such multiplications are the same (hence we have n_1-1 of them), however each of the blocks below the diagonal gets multiplied with a different scalar giving the second term $(n_1^2-n_1)/2\prod_{i=2}^{N} n_i$ inside the parentheses. In the first level of partitioning, $(n_1^2-n_1)/2$ is simply the number of blocks below the diagonal and $\prod_{i=2}^{N} n_i$ is the length of the subvector. The second term of T_1 is for the number of scalar multiplications performed in computing the current approximate subvector–tensor product multiplication due to local automata. Note that the actual vector–tensor product multiplications are accounted for as the first term inside the E parentheses.

In T_N, we have the number of scalar multiplications due to synchronizing events and due to local automata as the first and the second terms, respectively. The third term is for the number of multiplications performed in computing the diagonal corrector elements (i.e., each of the n_N diagonal elements in a block gets multiplied with the diagonal elements of the previous $N-1$ levels and this happens for all E synchronizing events), and the last term is for the number of divisions made at level N to obtain the solution.

In order to find a closed form, we write

$$\begin{aligned}
T_1 &= E\left[(n_1-1)\prod_{i=2}^{N} n_i \sum_{i=2}^{N} n_i + \frac{n_1^2-n_1}{2}\prod_{i=2}^{N} n_i\right] + \frac{n_1^2-n_1}{2}\prod_{i=2}^{N} n_i \\
&\quad + n_1\left[E\left[(n_2-1)\prod_{i=3}^{N} n_i \sum_{i=3}^{N} n_i + 2\frac{n_2^2-n_2}{2}\prod_{i=3}^{N} n_i\right] + \frac{n_2^2-n_2}{2}\prod_{i=3}^{N} n_i + n_2 T_3\right] \\
&< E\left[\prod_{i=1}^{N} n_i \sum_{i=2}^{N} n_i + \frac{n_1}{2}\prod_{i=1}^{N} n_i\right] + \frac{n_1}{2}\prod_{i=1}^{N} n_i + E\left[\prod_{i=1}^{N} n_i \sum_{i=3}^{N} n_i + n_2\prod_{i=1}^{N} n_i\right] \\
&\quad + \frac{n_2}{2}\prod_{i=1}^{N} n_i + n_1 n_2 T_3 \\
&< \prod_{i=1}^{N} n_i \left[E\left[\sum_{i=2}^{N} n_i + \frac{n_1}{2}\right] + E\left[\sum_{i=3}^{N} n_i + n_2\right] + \frac{n_1}{2} + \frac{n_2}{2}\right] + n_1 n_2 T_3 \\
&< E\left[\sum_{i=2}^{N} n_i + \sum_{i=3}^{N} n_i\right]\prod_{i=1}^{N} n_i + \left[\frac{E}{2}(n_1+2n_2) + \frac{n_1+n_2}{2}\right]\prod_{i=1}^{N} n_i + n_1 n_2 T_3 \\
&< \cdots \\
&< E\left[\sum_{i=2}^{N} n_i + \cdots + \sum_{i=N}^{N} n_i\right]\prod_{i=1}^{N} n_i + \left[\frac{E}{2}\sum_{i=1}^{N-1} i n_i\right]\prod_{i=1}^{N} n_i + \frac{1}{2}\sum_{i=1}^{N-1} n_i \prod_{i=1}^{N} n_i \\
&\quad + \prod_{i=1}^{N-1} n_i T_N.
\end{aligned}$$

Noting that

$$\prod_{i=1}^{N-1} n_i T_N = \frac{1}{2}ENn_N\prod_{i=1}^{N} n_i + \frac{1}{2}EN\prod_{i=1}^{N} n_i + \frac{1}{2}n_N\prod_{i=1}^{N} n_i + \frac{1}{2}\prod_{i=1}^{N} n_i,$$

we get the (loose) bound

$$T_1 < \frac{3}{2}EN\sum_{i=1}^{N} n_i \prod_{i=1}^{N} n_i + \frac{1}{2}\sum_{i=1}^{N} n_i \prod_{i=1}^{N} n_i.$$

Similarly one can find an upper bound on the number of multiplications performed in computing the right hand side b as $(EN+1)\sum_{i=1}^{N} n_i \prod_{i=1}^{N} n_i$. Here, EN is due to synchronizing events and 1 is due to local automata. Each tensor product arising from local automata has one upper triangular matrix; all others are identity. It is not surprising to find the total number of multiplications performed in one iteration of the GS method on a SAN descriptor for the algorithm given in this paper to be $O(EN\sum_{i=1}^{N} n_i \prod_{i=1}^{N} n_i)$.

Chapter 5

Numerical Results

5.1 The Problems and the Experiments

In order to make illuminating comparisons, we implemented power, Jacobi, GS, and SOR methods. We carried out experiments using both backward and forward versions of GS (and hence SOR) together with block versions of Jacobi, GS, and SOR methods. In block implementations, we terminate recursion at the deepest level and solve the blocks of order n_N using Gaussian elimination as discussed in Section 4.3. During the experiments we used a stopping criterion of 10^{-10} between consecutive approximations. That is, we computed a residual vector as the difference between consecutive approximations, and used the 2–norm of this vector as the stopping criterion. We used a uniformly distributed probability vector as the initial approximation for all experiments. We ran all the experiments on SUN Sparcstation 4's each with 32 megabytes (MB) of RAM. All the algorithms are implemented in C++ language and the new methods are incorporated into the software package PEPS [12]. Regardless of its size, each problem produced a smaller number of iterations in either the backward or the forward approach; we present results of the better approach.

We experimented with three problems. The first two, resource sharing and three queues, are explained in [5]. The third one, the model of a mass storage system, appears in [3]. For the mass storage example, we experimented with different orderings of the automata. Obviously, ordering of automata is likely to have an effect on the iteration count. The efficient vector–tensor product multiplication algorithm itself imposes an ordering on the automata. In order to use other orderings, a permutation vector may need to be introduced to the multiplication algorithm. We experimented with orderings that do not require permutation. We also tried orderings different from the original ordering by taking advantage of the position of identity matrices in tensor products. Such orderings follow from Lemma 5.5 and its companion remark in [5, p. 16].

Modeling with SANs is still in its infancy, and only recently have researchers started considering large and complex problems. Issues related to cyclic dependencies are currently under investigation. Lemma 5.8 and Theorem 5.2 in [5] show how one can handle cyclic dependencies in generalized tensor products. If the functional dependency graph is fully connected there is not much that can be done to improve the complexity of vector-generalized tensor product multiplication. On the other hand, if the cutset of the cycles in the dependency graph has a small number of automata, then a more efficient vector-generalized tensor product multiplication algorithm can be used. However, this multiplication will still take much longer than that of a vector-generalized tensor product lacking cycles. The smaller the cutset, the better the improvement. Moreover, at the end of Section 6 in [5], it is indicated that Theorem 5.2 needs to be used only when routing probabilities associated with synchronizing events (i.e., descriptors of slave automata due to synchronizing events) are functional and result in cycles within the functional dependency graph. The occurrence of this situation is suspected to be rare by the authors of [5]. We have not seen such a case. However, it is still not impossible to have generalized tensor products with dependency cycles. We should emphasize that no attempt has been made to avoid cyclic dependencies in the modeling phase of the mass storage problem. In [6], the last paragraph of subsection 4.3 discusses the results of some experiments with artificially created cyclic dependencies. There it is mentioned that cycles have a detrimental effect on solution time, as expected. As for ordering the automata in the case of non-cyclic dependencies, we think it should not be very difficult. It is an implementation issue. However, we have purposefully concentrated on orderings that do not require the introduction of

Table 5.1: Storage Requirements and Generation Times for All Problems

Prob. 1				Desc.	Sparse	Sparse
N	P		n	nz	nz	$gtime$
12	1		4,096	48	28,684	1
12	6		4,096	48	40,960	1
12	10		4,096	48	53,236	1
16	1		65,536	64	589,840	26
16	8		65,536	64	851,968	26
16	15		65,536	64	1,114,096	27
20	1		1,048,576	80	11,534,356	870
20	10		1,048,576	80	16,777,216	889
20	19		1,048,576	80	22,020,076	882
Prob. 2				Desc.	Sparse	Sparse
C_1	C_2	C_3	n	nz	nz	$gtime$
5	5	10	2,500	105	11,875	0
10	10	10	10,000	145	50,960	1
10	10	20	40,000	225	205,000	6
15	15	20	90,000	265	471,605	13
15	15	30	202,500	345	1,063,125	30
15	15	50	562,500	505	2,957,025	84
20	20	50	1,000,000	545	5,315,100	147
Prob. 3				Desc.	Sparse	Sparse
C	N_i		n	nz	nz	$gtime$
26	6		6,480	95	39,960	1
51	11		73,205	200	479,160	14
76	16		327,680	330	2,191,360	86
101	21		972,405	485	6,575,310	331

a permutation vector. Searching for optimal orderings and relaxation parameters when testing newly devised algorithms is a problem in its own right and we have not attempted experimenting with all $N!$ orderings of N automata.

The main advantage of using SANs is memory efficiency as opposed to time efficiency. We implemented power, Jacobi, GS, SOR methods and their block versions for sparse matrices in the Harwell–Boeing format so that a comparison can be made. The sparse matrices are generated using the descriptors, which are also stored in sparse format. In Table 5.1, we present the sizes of the problems, the number of nonzero elements stored in sparse matrices and descriptors, and the generation times of the sparse matrices. The generation times of the sparse matrices should be added to the solution times of the sparse methods. In the table, *nz* denotes the number of nonzeros either in the descriptor approach (Desc.) or the sparse matrix approach (Sparse), and *gtime* denotes the global matrix generation time in sparse format. We should remark that identity matrices arising in synchronizing events or local transitions are kept in a special data structure and do not contribute to the space complexity of the descriptor approach. The generation time of the descriptor in each problem is negligible and hence not reported. Since one is limited with a certain amount of core memory on a target architecture, we report results with sparse methods only in problems for which we could generate and store the global transition rate matrix. That we could solve larger problems using the sparse matrix approach if we had used a larger core is immaterial. In this work, we aim at investigating the "relative" worth of the SAN approach compared to the sparse matrix approach for the solution methods at hand on a target architecture. Research along other viable alternatives for handling large numbers of nonzeros in sparse matrices is also of interest to researchers (see [8], for instance). In the following w_* refers to the optimal relaxation parameter, *it* and *time* denote respectively the number of iterations and the CPU time (in seconds) to converge to the prespecified tolerance. The bold figures in Tables 5.2–5.9 indicate the best run times for the particular problem. In the following sections, we describe the problems and present the results of the experiments with the descriptor methods and the sparse matrix methods.

5.2 The Resource Sharing Problem

The resource sharing problem has four parameters. The number of processes N, the number of processes P that can simultaneously access the critical resource, the rate $\lambda^{(i)}$ at which each process wakes up and tries to acquire the resource, and the rate $\mu^{(i)}$ at which each resource using the process releases the resource for $i = 1, 2, \ldots, N$. All automata have two states implying $n = 2^N$. In our experiments we used $\lambda^{(i)} = 0.04$ and $\mu^{(i)} = 0.4$ for $i = 1, 2, \ldots, N$. This model does not have any synchronizing events; it has functional transition rates but

Table 5.2: Results of Desc. Experiments with the Resource Sharing Problem

Prob. 1		Power		GS		SOR			Block GS		Block SOR		
N	*P*	*it*	*time*	*it*	*time*	w_*	*it*	*time*	*it*	*time*	w_*	*it*	*time*
12	1	142	83	2	**2**	1.0	2	**2**	2	**2**	1.0	2	**2**
12	6	222	131	26	23	1.3	18	16	26	22	1.3	18	**15**
12	11	222	123	28	25	1.3	18	16	26	22	1.3	18	**15**
16	1	188	2,299	2	39	1.0	2	40	2	**38**	1.0	2	**38**
16	8	294	3,793	32	613	1.3	22	420	32	592	1.3	22	**402**
16	15	294	3,562	34	650	1.4	22	420	32	589	1.3	22	**402**
20	1	236	63,265	2	825	1.0	2	826	2	**740**	1.0	2	**740**
20	10	362	94,157	38	15,039	1.5	26	9,777	38	13,764	1.4	24	**8,734**
20	19	364	89,126	40	15,554	1.5	24	8,891	40	14,311	1.5	24	**8,673**

Table 5.3: Results of Sparse Experiments with the Resource Sharing Problem

Prob. 1		Power		GS		SOR			Block GS		Block SOR		
N	*P*	*it*	*time*	*it*	*time*	w_*	*it*	*time*	*it*	*time*	w_*	*it*	*time*
12	1	142	4	2	**0**	1.0	2	**0**	2	**0**	1.0	2	**0**
12	6	222	9	26	1	1.3	18	1	26	2	1.3	18	**1**
12	11	222	12	28	2	1.3	18	**1**	26	2	1.3	18	2
16	1	188	118	2	**2**	1.0	**2**	2	2	3	1.0	2	3
16	8	294	255	32	30	1.3	22	**22**	32	52	1.3	22	37
16	15	294	326	34	39	1.4	22	**29**	32	63	1.3	22	44

no cyclic dependencies. Since all matrices are identical for the given $\lambda^{(i)}$ and $\mu^{(i)}$, reordering the automata is futile. The resource sharing problem does not converge for Jacobi and block Jacobi methods. As for backward block GS and SOR methods, they are expected to give (slightly) smaller iteration counts than their point versions when P is closer to N than to 0. This follows from Remark 4.1 and is particularly substantiated for the GS iteration. When P is small compared to N, many of the upper diagonal elements of the 2×2 matrices evaluate to zero and there is no advantage of using block methods. On the other hand, when P is larger, many functional rates evaluate to nonzero values and the block methods start to make a difference, however very little due to the extremely small block size. The results of the experiments are summarized in Table 5.2 and in Table 5.3. Observe that block SOR takes approximately 1/10th of the time power method takes for the case $N = 20$, $P = 19$, in the descriptor approach. We were not able to solve the largest three instances of the this problem with a sparse solver.

Table 5.4: Results of Descriptor Experiments with the Three Queues Problem

Prob. 2			Power		Jacobi		GS		SOR		
C_1	C_2	C_3	*it*	*time*	*it*	*time*	*it*	*time*	w_*	*it*	*time*
5	5	10	696	82	450	66	164	27	1.6	102	**17**
10	10	10	912	411	590	336	226	154	1.6	142	**98**
10	10	20	1,084	1,954	726	1,658	270	722	1.6	168	**455**
15	15	20	1,548	6,215	1,064	5,390	404	2,485	1.6	256	**1,577**
15	15	30	1,664	15,052	1,154	13,103	436	6,288	1.6	274	**3,838**
15	15	50	1,874	47,240	1,318	41,535	492	21,726	1.6	310	**11,962**
20	20	50	2,306	101,680	1,642	91,187	618	44,002	1.6	390	**27,123**

Prob. 2			Block Jacobi	
C_1	C_2	C_3	*it*	*time*
5	5	10	412	110
10	10	10	540	557
10	10	20	668	2,868
15	15	20	998	9,235
15	15	30	1,074	24,246
15	15	50	1,234	82,215
20	20	50	1,540	186,781

5.3 The Three Queues Problem

The three queues problem is an open queueing network of three finite capacity queues respectively with capacities $C_1 - 1$, $C_2 - 1$, and $C_3 - 1$ in which customers from queues 1 and 2 (try to) join queue 3. The customers that come through queues 1 and 2 are referred to as type 1 and type 2 customers. The arrival and service rates of queue i are respectively given by λ_i and μ_i for $i = 1, 2$. Queue 3 has a service rate of μ_{3_1} for type 1 and a service rate of μ_{3_2} for type 2 customers. The network is modeled using 4 automata $\mathcal{A}^{(1)}$, $\mathcal{A}^{(2)}$, $\mathcal{A}^{(3_1)}$, $\mathcal{A}^{(3_2)}$ with respectively C_1, C_2, C_3, and C_3 states. The state space size is given by $n = C_1 C_2 C_3^2$. Other details of this queueing network may be found in [5]. The parameters used in the experiments are $\lambda_1 = 0.4$, $\lambda_2 = 0.3$, $\mu_1 = 0.6$, $\mu_2 = 0.5$, $\mu_{3_1} = 0.7$, and $\mu_{3_2} = 0.2$. This model has both synchronizing events and functional rates; it does not have any cyclic dependencies.

For the three queues problem, the automata are ordered as $\mathcal{A}^{(1)}$, $\mathcal{A}^{(2)}$, $\mathcal{A}^{(3_1)}$,

Table 5.5: Results of Sparse Experiments with the Three Queues Problem

Prob. 2			Power		Jacobi		GS		SOR		
C_1	C_2	C_3	*it*	*time*	*it*	*time*	*it*	*time*	w_*	*it*	*time*
5	5	10	696	9	450	6	164	2	1.6	102	**1**
10	10	10	912	49	590	32	226	13	1.6	142	**9**
10	10	20	1,084	238	726	158	270	65	1.6	168	**44**
15	15	20	1,548	778	1,064	539	404	220	1.6	256	**153**
15	15	30	1,664	1,916	1,154	1319	436	541	1.6	274	**373**

Prob. 2			Block Jacobi	
C_1	C_2	C_3	*it*	*time*
5	5	10	412	17
10	10	10	540	91
10	10	20	668	654
15	15	20	998	2,186
15	15	30	1,074	6,922

$\mathcal{A}^{(3_2)}$. Backward SOR gives the best results. However, block versions of Gauss-Seidel and SOR do not make any difference since the matrices that correspond to the last automaton are all lower triangular. Block Jacobi gives smaller iteration counts than point Jacobi in this case as expected, yet the difference is negligible. The results of the experiments with descriptor methods are presented in Table 5.4. Note that point SOR takes a quarter of the time the power method takes for the largest problem that has 1,000,000 states, in the descriptor approach. The results of the experiments with the sparse solvers are presented in Table 5.5. in this problem, sparse methods could not be applied to the largest two instances.

5.4 The Mass Storage Problem

Fortunately, we were able to try all iterative methods in the mass storage problem (see [3]). The model is used to investigate the effects of interactive retrieval (get) and storage (put) requests, migration workload, and purging workload on a robotic tape library (RTL). The first (i.e, online storage) layer

Table 5.6: Parameters for the Mass Storage Problem.

λ_g:	arrival rate of get requests to the system
λ_p:	arrival rate of put requests to the disk cache
h_g:	hit ratio of get requests at the disk cache
h_p:	hit ratio of put requests at the disk cache
μ:	service rate of tape drives (includes robot tape mount and file seek times)
T:	total number of available tape drives in the tape server
t_i:	number of tape drives dedicated to interactive get requests
t_m:	number of tape drives dedicated to the migration queue ($T = t_i + t_m$)
n_1:	number of requests in the interactive tape queue (including any request(s) currently being served) ($0 \leq n_1 \leq N_1 - 1$)
T_1	threshold of requests at the interactive tape queue above which one tape drive from the migration tape queue is borrowed
n_2:	number of requests in the migration tape queue (including any request(s) currently being served) ($0 \leq n_2 \leq N_2 - 1$)
n_3:	number of put requests written to the disk cache which have not been migrated to the tape library yet ($0 \leq n_3 \leq N_3 - 1$)
$C - 1$:	maximum capacity of the disk cache.
H:	high water–mark for the disk cache used to activate the purging workload
L:	low water–mark used to terminate the purging workload
$\hat{C}$:	current occupancy level of the disk cache ($\lceil L(C-1) \rceil \leq \hat{C} \leq \lceil H(C-1) \rceil$)
M:	inter–migration time
R:	number of stages in the Erlangian approximation of the periodic migration workload ($R \geq 5$)
γ:	rate of the Erlangian approximation of the periodic migration workload ($\gamma = 1/m$)

usually consists of magnetic disks which provide fast access time but at a relatively high cost per byte. The second (i.e., nearline storage) layer utilizes robotic tape libraries (RTL), and the third (i.e., offline storage) layer consists of free–standing tape drives with human operators performing the mounting and unmounting of media from the drives. Since the interest is mainly in the performance of RTLs, it is assumed that the system to be modeled only consists of an online and a nearline layer. The parameters in this problem are quite a few, and we present them in Table 5.6. The unit of time for the given parameters is minutes. The system is modeled using five automata $\mathcal{A}^{(\tilde{C})}$, $\mathcal{A}^{(n_1)}$, $\mathcal{A}^{(n_2)}$, $\mathcal{A}^{(n_3)}$, and $\mathcal{A}^{(erl)}$ of order respectively $\lceil (H-L)(C-1) \rceil + 1$, N_1, N_2, N_3, and R giving

$$n = (\lceil (H-L)(C-1) \rceil + 1) N_1 N_2 N_3 R.$$

Table 5.7: Results of Descriptor Experiments with the Mass Storage Problem

.

Prob. 3		Power		Jacobi		GS		SOR		
C	N_i	*it*	*time*	*it*	*time*	*it*	*time*	w_*	*it*	*time*
26	6	178	**78**	1,522	1,064	254	217	1.7	168	144
51	11	612	3,062	2,084	17,765	334	3,485	1.6	228	2,354
76	16	1,146	29,432	2,130	92,364	428	21,207	1.5	306	14,910
101	21	1,860	145,162	2,842	394,517	668	104,229	1.5	454	70,774

Prob. 3		Block Jacobi		Block GS		Block SOR		
C	N_i	*it*	*time*	*it*	*time*	w_*	*it*	*time*
26	6	$> 2,700$	10^{-5}	158	140	1.7	98	88
51	11	1,598	18,676	156	1,673	1.6	106	**1,125**
76	16	$> 3,000$	10^{-5}	286	14,759	1.7	170	**8,876**
101	21	1,935	370,166	470	79,665	1.7	282	**46,708**

The mass storage model has both synchronizing events and functional rates; it does not have any cyclic dependencies.

We used $\lambda_g = \lambda_p = 1.5$, $\mu = 0.61$, $h_g = h_p = 0.3$, $t_i = t_m = 2$, $L = 0.75$, $H = 0.95$, $M = 40$, $R = 5$ (see [3, p. 5] for details). The automata are ordered as $\mathcal{A}^{(n_4)}, \mathcal{A}^{(n_1)}, \mathcal{A}^{(erl)}, \mathcal{A}^{(\tilde{C})}, \mathcal{A}^{(n_2)}$. In Table 5.7, we provide results for both block and point methods of the descriptor approach. Forward SOR gives the best results; its block version decreases both the iteration counts and the solution times. The information in Table 5.7 regarding block Jacobi should be interpreted differently. The two entries with $>$ signs in the iteration column and 10^{-5} in the time column indicate that the methods are executed until the 2–norm of the residual vector is on the order of 10^{-5}, and the iteration counts reach the numbers in the iteration column. The methods are not executed until the 2–norm of the residual vector is on the order of 10^{-10} since these runs take quite some time.

Interestingly, an alternative ordering, namely $\mathcal{A}^{(n_4)}, \mathcal{A}^{(\tilde{C})}, \mathcal{A}^{(erl)}, \mathcal{A}^{(n_1)}, \mathcal{A}^{(n_2)}$ gives better results for both block GS and block SOR as shown in Table 5.9. Note that it is possible to solve the largest system in less than two hours.

A final remark is that, for a given problem, the optimal parameter of (block) SOR and therefore the number of iterations taken to convergence

Table 5.8: Results of Sparse Experiments with the Mass Storage Problem

Prob. 3		Power		Jacobi		GS		SOR		
C	N_i	it	$time$	it	$time$	it	$time$	w_*	it	$time$
26	6	178	**7**	1,522	62	254	**11**	1.07	240	**11**
51	11	612	302	2,084	1018	334	177	1.06	318	**182**
76	16	1,146	2,763	2,130	5,014	428	1,056	1.06	406	1,112

Prob. 3		Block Jacobi		Block GS		Block SOR		
C	N_i	it	$time$	it	$time$	w_*	it	$time$
26	6	$> 2,700$	10^{-5}	158	16	1.09	146	16
51	11	1,598	2,388	156	230	1.10	144	219
76	16	$> 3,000$	10^{-5}	286	2,289	1.07	268	2,196

Table 5.9: Results of Other Experiments with the Mass Storage Problem

Prob. 3		Descriptor Block GS		Descriptor Block SOR			Sparse Block GS		Sparse Block SOR		
C	N_i	it	$time$	w_*	it	$time$	it	$time$	w_*	it	$time$
26	6	44	**41**	1.0	44	**41**	44	**5**	1.0	44	**5**
51	11	34	**370**	1.0	34	**370**	34	**50**	1.0	34	**50**
76	16	32	**1,715**	1.0	32	**1,715**	32	**255**	1.0	32	**255**
101	21	40	6,797	1.1	36	**6,115**	-	-	-	-	-

in the descriptor approach may be (significantly) different than those of the global generator in sparse format. This is something we observed in the mass storage problem for the ordering $\mathcal{A}^{(n_4)}, \mathcal{A}^{(n_1)}, \mathcal{A}^{(erl)}, \mathcal{A}^{(\tilde{C})}, \mathcal{A}^{(n_2)}$. For instance, $w_* = 1.1, it = 144, time = 219$ for block SOR in sparse format for the given ordering when $C = 51, N_i = 11$. The cause seems to be rounding errors incurred in generating and storing the global matrix.

Chapter 6

Conclusion

In this work, we presented iterative methods based on splitting a SAN descriptor. Block versions of the same methods follow directly from considering blocks of order n_N, the order of the last automaton, in the given ordering. Larger blocks may be considered by grouping several automata at the end of the given ordering and terminating recursive calls of the lower triangular back solve algorithm when the first automata in the group is encountered.

By deriving an upper bound on the Gauss-Seidel and SOR algorithms for the number of multiplications that is in the same order with the vector–descriptor multiplication, we show that the stationary methods are as efficient as non-stationary methods that do only vector–descriptor multiplications. Hence, we show that the solution times of a specific problem with different approaches depend only on the behavior of the algorithm for the given problem.

An important and frequently overlooked drawback of Markov chain solvers (including SAN solvers) that attempt at computing each and every stationary probability is the memory consumed by double precision temporary storage allocated to the current approximation, possibly the preceding one, and other work arrays. A vector of one million elements requires 8 MB of memory. Although not as large as the memory taken up by double precision nonzeros in the sparse matrix approach, these vectors may end up taking substantial space in iterative methods.

On a desktop workstation with 32 MB of RAM, one can compute the stationary distribution of a SAN descriptor with one million states in core on the order of hours using block SOR. On the other hand, the largest system that can be solved by the sparse matrix approach may be limited to less than one tenth of that could be solved using SANs if the generator is reasonably dense (as in the resource sharing problem: it takes roughly 176 MB to store the generator matrix in sparse format for the most difficult case). We believe the SAN modeling methodology has its merits and drawbacks. One may easily observe that sparse methods must be used whenever possible. The SAN formalism is likely to gain popularity as a viable modeling and analysis tool as faster solvers become available.

Appendix A

Incorporating a New Model To Peps

A.1 Preliminaries

This section explains how to incorporate a performance model, developed as a SAN to the software package Peps [12]. It is assumed that the model to incorporate must be available; that is, local matrices and synchronizing event matrices with their diagonal correctors must be available. A mathematical formulation of functional rates should also be available, i.e., one must know how to evaluate the function given the state of the automata. In the remaining, we use 'Peps', to mean the latest revision of the package as it is implemented at Bilkent University. We use 'original Peps' to mean the version that is supplied to Bilkent University as in [12]. There are two steps to complete the task of adding a new model. First, one should generate a text file describing the model; second, the necessary code for evaluating the functional rates should be incorporated to the package. The two sections that follow explain these two steps. The final section presents the text file for an example model.

Table A.1: Matrix Types

value	type	meaning
0	sparse	sparse matrix in HBF format
1	binary	NOT USED
2	element	NOT USED
3	diagonal	NOT USED
4	identity	identity matrix
5	zero	zero matrix

A.2 Generating the Text File

The text file, which is given a `.dsc` extension stores the descriptor of a SAN in a predefined format. For instance, the name of a text file of the mass storage model in [3] can be given the name `mass.dsc` or `m-6-6-6-6.dsc`.

The largest portion of the text file is used for storing the descriptor matrices of the SAN. A few lines of other information is given in the text file. Before going into the format of the file, we discuss how matrices are represented.

A.2.1 Format of a Single Matrix

All matrices, i.e., local generator matrices, synchronizing event matrices and diagonal corrector matrices are represented in the same way. A matrix consists of several consecutive lines of text in the file.

In the first line in the portion of the text file describing a matrix, there should be a single number representing the type of the matrix. The possible values are 0,1,2,3,4,5 as shown in Table A.1.

The first five matrix types are defined in original Peps; however, binary, element, and diagonal types are not used in Peps and original Peps. The zero matrix type is introduced in Peps.

In the next line there should be two integer values separated with a space. The first integer denotes the number of nonzero elements in the matrix. Since

Table A.2: Types of Nonzero Values

value	type	meaning
0	rate	a rate for continuous time MCs
1	probability	NOT USED
2	function	a functional entry
3	parameter	NOT USED

all matrices are square, the second number specifies the order of the matrix.

For zero and identity type matrices, these two lines completely specify a matrix. For a sparse matrix, which is neither a zero matrix nor an identity matrix, the lines following the first two describe the nonzero elements of the matrix. We follow the compact column format representation of sparse matrices. The number of lines that specify the nonzero elements of the matrix should be equal to the first integer given in the second line. In other words, a separate line is reserved for each nonzero element. The nonzero elements of the matrix are stored in a double precision array one column after the other starting from column zero.

A nonzero element is described using three numbers separated by spaces. The first number, an integer, defines the type of the element. There are four types of elements (see Table A.2). All of these types are defined in original Peps, however the probability and parameter types so far have no use in Peps and original Peps. The second number is the real value of the corresponding entry in the matrix for rate type elements. This entry is not currently used for function type elements. The third number is for the ID number of the function for functional elements in a C++ implementation file. This entry is not used for rate type elements.

In the line following the one that contains the last nonzero element, the row indices of all nonzero elements are written. The number of integers in this line should be equal to the number of nonzero elements of the particular sparse matrix. Note that row numbers start from zero in all matrices due to the C++ implementation.

In the next line, there should be $d + 1$ integers, where d is the order of the matrix. Each integer specifies the location of the first nonzero element, in the array of nonzero elements, of a particular column. That is, the kth integer will be the index of the first nonzero element of the kth column, in the array of nonzero elements. The final integer should be equal to the number of nonzero elements in the matrix. We illustrate the storage scheme on various examples in the next section

A.2.2 Example Matrices

Zero and identity matrices are easy to specify. A zero matrix of order 6 is written in two lines as:

5
0 6

An identity matrix of order 5 is written in two lines as:

4
5 5

The below sparse matrix containing only real entries

$$\begin{bmatrix} -0.4 & 0.4 & 0 \\ 0 & -0.4 & 0.4 \\ 0 & 0 & 0 \end{bmatrix}$$

is written as:

0				// type of matrix
4	3			// number of non-zeros and size of matrix
0	−0.4	0		// first nonzero element
0	0.4	0		// second nonzero element
0	−0.4	0		// third nonzero element
0	0.4	0		// fourth nonzero element
0	0	1	1	// rows of non-zeroes
0	1	3	4	// column indices for each column

The matrix below contains both functional and real entries :

$$\begin{bmatrix} f_0 & f_1 & 0 \\ 0 & -0.5 & 0.5 \\ 0 & 0 & 0 \end{bmatrix}.$$

It is represented as:

0				
4	3			
2	0	100		// functional element
2	0	101		// functional element
0	−0.5	0		
0	0.5	0		
0	0	1	1	
0	1	3	4	

The comments given beside some of the lines should not appear in the actual text files. Here, they are given to elaborate certain concepts.

A.2.3 The Text File and Its Parts

The text file is organized as a set of lines. In the first line, there should be three integers separated by spaces. The first one stands for the type of the model. The model can be a discrete-time model or a continuous-time model. Currently both original Peps and Peps work with continuous-time models. The possible values are 0 and 1 (see Table A.3).

Table A.3: Model Types

value	type	meaning
0	discrete	NOT USED
1	continuous	continuous time model

The second integer is for the number of automata and the third is for the number of synchronizing events. Sizes of automata are written in the second line. For instance the first two lines of a text file describing a continuous–time model with 4 automata and 2 synchronizing events is given below:

```
1 4 2        // type of SAN, number of automata, number of sync. events
3 4 5 5      // sizes of the four automata
```

In this example, the first matrix is 3×3, the second is 4×4, the third and the fourth are 5×5.

After these two lines, the local automata matrices should be written one after the other as explained in section A.2.1. Note that the number of local matrices should be equal to the second number in the first line of the text file. Following the local automata, the synchronizing event matrices should be written. First, all matrices of the first synchronizing event (including the correctors) should be listed, then the second synchronizing event's matrices, and so on. For each synchronizing event, the synchronizing event matrix of the first automaton should be written followed by its corrector, then the synchronizing event matrix of the second automaton should be written followed by its corrector, and so on.

After all synchronizing event matrices are written, the orderings of the rate matrices for the synchronizing events should be written in a single line. The rate matrix is the matrix that contains rate values for the synchronizing event, i.e., master of the synchronizing event. Actually these values are not used in the stationary vector calculations, but they are used in the thruput calculations. These calculations are made in the `Calculate` function of the `tensor.C` module. For example, in the three queues problem this line should contain the integers 1 and 2. Given that the automata are ordered as $\mathcal{A}^{(1)}$,$\mathcal{A}^{(2)}$,$\mathcal{A}^{(3_1)}$, $\mathcal{A}^{(3_2)}$, this means automaton $\mathcal{A}^{(2)}$ is the master automaton for the first synchronizing event and automaton $\mathcal{A}^{(3_1)}$ is the master automaton for the second synchronizing event. For the mass storage example, there should be three integers in the same line. Since the generation program is capable of generating any ordering of automata, these values will vary with the ordering. However, the three integers should be the orderings of the Erlangian server, queue three and queue one respectively: $\mathcal{A}^{(erl)}$, $\mathcal{A}^{(n_3)}$, $\mathcal{A}^{(n_1)}$. The `Calculate` function takes these values, retrieves a certain element in the matrices specified by these values, and uses the element in thruput calculations. This approach is useful in the sense that if some parameters of the example are altered, there is no need to make changes in the `Calculate` function. Currently the only use of this line is for the mass storage example in Peps. This line exists in original Peps, but is not used.

The next line in the text file is the last one and is intended to be used as the ordering of the automata. The number of integers on this line should be equal to the number of automata. In original Peps, this line does not exist. In Peps, this line is used only for the mass storage example. For the mass storage example, the values in this line are used only in function evaluations. Each number tells where a specific automaton is located in the given ordering, i.e., the first number explains where the Erlangian server is, the second where the cache is, the third, the fourth and the fifth numbers explain where $\mathcal{A}^{(n_3)}$, $\mathcal{A}^{(n_1)}$ and $\mathcal{A}^{(n_2)}$ are, respectively. For example, 2 0 1 3 4 corresponds to the ordering $\mathcal{A}^{(C)}$, $\mathcal{A}^{(n_3)}$, $\mathcal{A}^{(erl)}$, $\mathcal{A}^{(n_1)}$, $\mathcal{A}^{(n_2)}$.

A.3 Evaluating Functional Entries in Peps

In order to implement a functional entry of a matrix in Peps, there are two things to do. First an ID number should be determined for the functional entry and must be written into the matrix as described in section A.2.1. Second the actual code to implement the function must be incorporated to Peps. The code must be added to the module `function.C` and into the C++ function `Evaluate_Function`. This function's signature is:

```
rp Evaluate_Function(const function_id id,
                 const state_id * params, const int size)
```

This function is called for each functional entry. The first parameter `id` is the ID of the functional entry for which the function is called. The second parameter `params` is an array of `state_ids` (basicly integers), whose elements correspond to the state of automata. The third parameter `size` is the size of the array of states. The value returned by this function determines the value of the functional entry at the point of call.

ID numbers 0 to 99 are used in the three queues and resource sharing problems. The mass storage example uses ID numbers from 10000 to 10000+(size of the third queue), 100, and 101. The remaining ID numbers can be used freely in `Evaluate_Function`. There are some declarations in the first few lines. After those, one can add code for evaluating functions in a different model. The code should be an if-block. The check for the ID range, should be made in the if condition, and if the condition evaluates to true, the necessary calculations should be performed. The result of the calculation should be returned. That is, the if-block should end with a `return` statement. If the condition does not evaluate to true, control should be left to the remaining part of the code. An important point is that one must guarantee to return the result with a return statement when the ID is in the appropriate range.

In addition to the input parameters of the function, extra information that might be needed in function evaluations is available. An array of integers, `automata_sizes`, contains the order of automata. Another integer array, `ordering`, which is intended to be used as the ordering of automata, is also available. Since the latter array is supplied in the text file, it might be used for other purposes as well. Note that both arrays have as many elements as the number of automata.

Below is a sample C++ code snippet for evaluating a function as it would appear in `function.C`:

```
rp Evaluate_Function(const function_id id,
               const state_id * params, const int size)
{
 rp  result;
 int automata_acc = 0;
 automaton_id a;

// Example code starts here
// You should add your code here
if ((id >=200) && (id<300)) {
    switch (id) {
        case 200 : if (params[0] > (automata_sizes[0]/2))
                       result = 1.0;
```

```
                   else
                       result = 0.0;
            break;
        case 201 :  if (params[2]==0)
                       result = 1.0;
                   else
                       result = 0.0;
    break;
    }
    return result;
}
// End of sample code

...
...
...
```

A.4 An Example Text File

Below is an example file generated for the three queues example. Note that the comments (i.e., part of lines after the // characters) are added to this document for explanation purposes and should not appear in the actual text file.

```
1 4 2             // cont. time, 4 automata, 2 sync. events
3 3 5 5           // C1 = 3, C2 = 3, C3 = 5

0                 // local automaton of queue 1, sparse type
4 3
0 -0.4 0
0  0.4 0
0 -0.4 0
```

```
0  0.4 0
0 0 1 1
0 1 3 4
0                  // local automaton of queue 2, sparse type
4 3
0 -0.3 0
0  0.3 0
0 -0.3 0
0  0.3 0
0 0 1 1
0 1 3 4
0                  // local automaton of queue 3_1, sparse type
8 5
0  0.7 0
0 -0.7 0
0  0.7 0
0 -0.7 0
0  0.7 0
0 -0.7 0
0  0.7 0
0 -0.7 0
1 1 2 2 3 3 4 4
0 1 3 5 7 8
0                  // local automaton of queue 3_2, sparse type
8 5
2  0.0 20
2  0.0 21
2  0.0 20
2  0.0 21
2  0.0 20
2  0.0 21
2  0.0 20
2  0.0 21
1 1 2 2 3 3 4 4
0 1 3 5 7 8
```

```
0                  // sync. event #1, matrix of queue 1
2 3                         // sparse type
0  0.6 0
0  0.6 0
1 2
0 1 2 2
0                  // sync. event #1, corrector matrix of queue 1
2 3                         // sparse type
0 -0.6 0
0 -0.6 0
1 2
0 0 1 2
4                  // sync. event #1, matrix of queue 2
3 3                         // identity type
4                  // sync. event #1, corrector matrix of queue 2
3 3                         // identity type
0                  // sync. event #1, matrix of queue 3_1
4 5                         // sparse type
2  0.0 22
2  0.0 22
2  0.0 22
2  0.0 22
0 1 2 3
0 0 1 2 3 4
0                  // sync. event #1, corrector matrix of
4 5                         // queue 3_1, sparse type
2  0.0 22
2  0.0 22
2  0.0 22
2  0.0 22
0 1 2 3
0 1 2 3 4 4
4                  // sync. event #1, matrix of queue 3_2
5 5                         // identity type
```

```
4                  // sync. event #1, corrector matrix of
5 5                         // queue 3_2, identity type

4                  // sync. event #2, matrix of queue 1
3 3                         // identity type
4                  // sync. event #2, corrector matrix of queue 1
3 3                         // identity type
0                  // sync. event #2, matrix of queue 2
2 3                         // sparse type
0  0.5 0
0  0.5 0
1 2
0 1 2 2
0                  // sync. event #2, corrector matrix of queue 2
2 3                         // sparse type
0 -0.5 0
0 -0.5 0
1 2
0 0 1 2
4                  // sync. event #2, matrix of queue 3_1
5 5                         // identity type
4                  // sync. event #2, corrector matrix of
5 5                         // queue 3_1, identity type
0                  // sync. event #2, matrix of queue 3_2
9 5                         // sparse type
2  0.0 23          // a functional rate with ID 23
2  0.0 22          // a functional rate with ID 22
2  0.0 23
2  0.0 22
2  0.0 23
2  0.0 22
2  0.0 23
2  0.0 22
0  1.0 0           // a rate with value 1.0
0 0 1 1 2 2 3 3 4 // row numbers of 9 nonzero elements
```

```
0 1 3 5 7 9        // column 2 starts with nonzero #3, 2 0 22
4                  // sync. event #2, corrector matrix of
5 5                         //  queue 3_2, identity type

1 2                // rate matrices for synchronizing events
0 1 2 3            // ordering of automata
                            // not used in this example
```

Bibliography

[1] Benzi, M., and Dayar, T., "The arithmetic mean method for finding the stationary vector of Markov chains," *Parallel Algorithms and Applications*, 6 (1995) 25–37.

[2] Davio, M., "Kronecker products and shuffle algebra," *IEEE Transactions on Computers*, C–30/2 (1981), 116–125.

[3] Dayar, T., Pentakalos, O. I., and Stephens, A. B., "Analytical modeling of robotic tape libraries using stochastic automata," *Technical Report TR-97-198*, CESDIS, NASA/GSFC (1997).

[4] Dianne, O. P., "Iterative methods for finding the stationary vector for Markov chains," in: *Linear Algebra, Markov Chains, and Queueing Models*, Springer–Verlag, Meyer, C., D., Plemmons, R., J., eds., New York, NY, (1993) 125–136.

[5] Fernandes, P., Plateau, B., and Stewart, W. J., "Efficient descriptor–vector multiplications in stochastic automata networks," *INRIA Report # 2935*. Anonymous ftp ftp.inria.fr/INRIA/Publication/RR.

[6] Fernandes, P., Plateau, B., and Stewart, W. J., "Numerical issues for stochastic automata networks," in: *PAPM 96, Fourth Process Algebras and Performance Modelling Workshop*, Torino, Italy, July 1996.

[7] Hageman, L., and Young, D., *Applied Iterative Methods*, Academic Press, New York, NY, 1981.

[8] Malhis, L. M., and Sanders, W. H., "An efficient two–stage iterative method for the steady–state analysis of Markov regenerative stochastic Petri net models," *Performance Evaluation*, 27&28 (1996), 583–601.

[9] Plateau, B., "On the stochastic structure of parallelism and synchronization models for distributed algorithms," in: *Proceedings of the SIGMETRICS Conference on Measurement and Modelling of Computer Systems*, Austin, TX, August 1985, 147–154.

[10] Plateau, B., and Atif, K., "Stochastic automata network for modeling parallel systems," *IEEE Transactions on Software Engineering*, 17/10 (1991) 1093–1108.

[11] Plateau, B., and Fourneau, J. M., "A methodology for solving Markov models of parallel systems", *Journal of Parallel and Distributed Computing*, 12 (1991) 370–387

[12] Plateau, B., Fourneau, J. M., and K.-H. Lee, "PEPS: A package for solving complex Markov models of parallel systems," in: *Modeling Techniques and Tools for Computer Performance Evaluation*, R. Puigjaner and D. Potier, eds., Spain, September 1988, 291–305.

[13] Saad, Y., and Schultz, M., H., "Conjugate gradient like algorithms for solving nonsymmetric linear systems,"*Mathematics of Computation*, 170 (1985) 417–424

[14] Semal, P., "Refinable Bounds for large Markov chains," *IEEE Transactions On Computers*, 10 (1995) 1216–1222.

[15] Stewart, W. J., MARCA: Markov Chain Analyzer, *IEEE Computer Repository No. R76 232*, 1976.

[16] Stewart, W. J., *Introduction to the Numerical Solutions of Markov Chains*, Princeton University Press, Princeton, NJ, 1994.

[17] Stewart, W. J., Atif, K., and Plateau, B., "The numerical solution of stochastic automata networks," *European Journal of Operational Research*, 86 (1995) 503–525