DATA STRUCTURES FOR THE ANALYSIS OF LARGE STRUCTURED MARKOV MODELS

A Dissertation

Presented to

The Faculty of the Department of Computer Science

The College of William & Mary in Virginia

In Partial Fulfillment

Of the Requirements for the Degree of

Doctor of Philosophy

---

by

Andrew Stephen Miner

2000

APPROVAL SHEET

This dissertation is submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

Andrew S. Miner

Approved, June 2000

Gianfranco Ciardo
Thesis Advisor

Steve Park

Evgenia Smirni

Andreas Stathopoulos

Virginia Torczon

Alex Pothen
Old Dominion University

To my mother, who bought my first computer

Table of Contents

ACKNOWLEDGMENTS

This work would have been impossible without the help of many people. First and foremost, I would like to thank my advisor, Gianfranco Ciardo, who tolerated my unusual sleeping schedules. His office door was always open and we had countless fruitful discussions about SMART, research ideas, and papers. We also attended some excellent conferences and shared an extremely dangerous daily commute via rickety bicycles in Torino.

I would also like to thank Steve Park, who somehow managed to find time in his busy schedule as Department Chair (and now as Dean of Arts and Sciences) to help me with grant proposals, cover letters, and other delicate matters of diplomacy and word-smithing. I must also profusely thank Evgenia Smirni. Although I could not possibly enumerate the entire list of items for which I owe her thanks (which grows by the day), I will at least say that I would almost certainly be unemployed without her help. Many thanks also to Andreas Stathopoulos, Virginia Torczon, and Alex Pothen (and to the rest of the committee members already named) for reading this rather large body of work and providing many valuable constructive comments. Thanks also to Susanna Donatelli for arranging my productive research visit to the Universitá di Torino, Italy. I hope it was the first of many such visits. I would also like to thank the Virginia Space Grant Consortium, the NASA Graduate Student Researchers Program, and the Department of Computer Science at William and Mary for financial support during my graduate studies.

Finally, on a personal note, I thank my family and friends for their support, especially Shannon. I will always remember Wednesday pool, "celebrity" Cheese Shop lunches, disc golf, frisbee at all hours, Mystery Science Theater 3000, and 1am trips to Dennys.

This document was prepared using the LaTeX document preparation system [62]. The figures were drawn with Tgif.

List of Tables

List of Figures

List of Algorithms

List of Symbols

ABSTRACT

High-level modeling formalisms are increasingly popular tools for studying complex systems. Given a high-level model, we can automatically verify certain system properties or compute performance measures about the system. In the general case, measures must be computed using discrete-event simulations. In certain cases, exact numerical analysis is possible by constructing and analyzing the underlying stochastic process of the system, which is a continuous-time Markov chain (CTMC) in our case. Unfortunately, the number of states in the underlying CTMC can be extremely large, even if the high-level model is "small". In this thesis, we develop data structures and techniques that can tolerate these large numbers of states.

First, we present a multi-level data structure for storing the set of reachable states of a model. We then introduce the concept of event "locality", which considers the components of the model that an event may affect. We show how a state generation algorithm using our multi-level structure can exploit event locality to reduce CPU requirements.

Then, we present a symbolic generation technique based on our multi-level structure and our concept of event locality, in which operations are applied to sets of states. The extremely compact data structure and efficient manipulation routines we present allow for the examination of much larger systems than was previously possible.

The transition rate matrix of the underlying CTMC can be represented with Kronecker algebra under certain conditions. However, the use of Kronecker algebra introduces several sources of CPU overhead during numerical solution. We present data structures, including our new data structure called *matrix diagrams*, that can reduce this CPU overhead. Using our techniques, we can compute measures for large systems in a fraction of the time required by current state-of-the-art techniques.

Finally, we present a technique for approximating stationary measures using aggregations of the underlying CTMC. Our technique utilizes exact knowledge of the underlying CTMC using our compact data structure for the reachable states and a Kronecker representation for the transition rates. We prove that the approximation is exact for models possessing a product-form solution.

DATA STRUCTURES FOR THE ANALYSIS OF LARGE STRUCTURED MARKOV MODELS

Chapter 1

Introduction

Advancements in technology demand increasingly complex systems. The widespread growth of the internet and wireless communications, for instance, have fueled research in techniques for analyzing large systems. The design of such a system almost certainly requires the use of computer models and simulations to assist engineers in making important design decisions. As a result, high-level modeling formalisms, such as stochastic Petri nets, are gaining acceptance as tools to study such systems. These high-level models allow for automatic verification and performance evaluation of systems whose analysis would otherwise be impossible.

Generally, a model to be analyzed has certain properties that need to be verified, such as "the system never reaches a deadlocked state". In some cases, there are also performance or reliability measures of interest to be determined, such as "what is the probability that the system is down". The former type of analysis can only be performed in general by a systematic examination of the states of the high-level model [19, 27, 49, 57]. For certain types of models, efficient *symbolic* techniques can be used, which do not require explicit examination of every state [17, 32, 68, 78, 79, 80, 81]. These techniques are quite promising, as the states described by a high-level model can easily number in the millions or billions.

Performance evaluation of a high-level model can be performed either by discrete-event simulation, by exact analysis, or by approximation. While discrete-event simulation is applicable to an extremely general class of problems, accurate solutions may require long simulation runs, especially if the analysis involves the study of rare events. Exact analysis, on the other hand, is applicable to certain types of stochastic models only. In our work, we consider a fairly general class of formalisms in which the underlying stochastic process is a Markov chain. In this case, analysis of the model requires generation and analysis of the underlying Markov chain. As mentioned above, it is not uncommon for these Markov chains to contain millions or billions of states. This leads to difficulties, as exact analysis requires us to represent the reachable states of the model, the transition rate matrix of the Markov chain, and a solution vector corresponding to the computed probability for each state. These three structures pose obvious storage difficulties when the number of states of the Markov chain becomes large. Much attention has been given to the transition rate matrix, as it is the largest of the three structures. Techniques based on Kronecker algebra have received much attention [12, 14, 16, 18, 28, 29, 41, 42, 44, 56, 84, 85, 86, 91], although some alternatives have also been investigated [38, 39, 52].

Approximation techniques often involve decomposing the model into submodels, which are then analyzed in isolation. The results obtained for the submodels are then combined. Fixed-point iterations can be used to resolve the dependencies between submodels. The overall storage and CPU requirements for the analyses of the submodels are usually a small fraction of those for an exact analysis of the entire model. As a result, model-based decompositions have been successfully used [20, 30, 50, 51, 53, 69, 93, 98, 100, 105, 107] to accurately approximate performance measures when an exact solution is infeasible.

1.1 Contributions

In this work, we address each of the three major structures required for exact stationary analysis: the set of reachable states $\mathcal{S}$, the transition rate matrix $\mathbf{R}$ of the Markov chain, and the stationary probability vector $\boldsymbol{\pi}$. We consider only a very small subset of model verification problems; namely, that of generating and examining the set of reachable states $\mathcal{S}$.

First, we develop a multi-level data structure for storing $\mathcal{S}$ that can be used with structured models. We then show how, when an event occurs, we can update a portion of the data structure only, by exploiting structural properties of the model. This new concept of the "locality" of an event can substantially reduce generation times, and is used throughout the work.

Second, we develop a technique for symbolically generating $\mathcal{S}$ that can be applied to a general class of structured models. We present an encoding scheme that combines ideas from our multi-level structure and from decision diagrams. We then develop specialized manipulation routines for our encoding which allow us to generate $\mathcal{S}$ extremely efficiently.

It has been shown that a Kronecker representation for the transition rate matrix $\mathbf{R}$ can reduce the storage requirements for $\mathbf{R}$ by orders of magnitude. However, Kronecker techniques suffer from significant sources of CPU overhead. Our third contribution consists of new data structures and techniques that eliminate or reduce these CPU overheads.

With efficient representations for $\mathcal{S}$ and $\mathbf{R}$, the only remaining bottleneck for exact analysis is the solution vector $\boldsymbol{\pi}$. Our fourth major contribution is a technique for approximating $\boldsymbol{\pi}$. Unlike other approximations, ours makes use of exact knowledge of $\mathcal{S}$ and $\mathbf{R}$ by using our previous contributions. This enables our technique to correctly assign a zero

probability to unreachable states.

Finally, the software tool SMART [26], which is discussed in Appendix A, represents a considerable contribution to the academic and modeling community.

1.2 Organization

The remainder of the thesis is organized as follows. The next three chapters are background chapters. Chapter 2 introduces our notation and gives some important background information about storing matrices and solving linear systems. Chapter 3 presents an overview of random variables and stochastic processes, with particular emphasis on Markov chains. Chapter 4 describes how a high-level formalism can be used to generate a Markov chain. Various classes of structured models are defined.

Our main contributions are presented in four chapters. Chapter 5 describes our multi-level data structure for explicit storage of the states of the model. We compare our data structure with several other explicit storage schemes. Chapter 6 presents our symbolic technique for generating and storing states of a structured model with certain properties. Our new approach is compared with existing symbolic approaches. Chapter 7 discusses approaches in which the transition rate matrix of the underlying Markov chain for a structured model is represented algebraically using Kronecker products and sums. We present data structures and techniques for reducing or eliminating overheads inherent with Kronecker approaches. Chapter 8 describes a novel technique for approximating stationary measures of a structured model, based on exact knowledge of the underlying Markov chain.

Example applications of our techniques are presented in Chapter 9. Concluding remarks

and directions for future work are given in Chapter 10. There are four appendices. Appendix A discusses SMART, a software package that incorporates the techniques described in this thesis. Appendix B describes the models we use as benchmarks throughout the work. Appendix C presents a detailed analysis of one of the algorithms described in Chapter 6. Finally, Appendix D derives an expression for the number of reachable states for one of our benchmark models.

Chapter 2

Background

This chapter covers basic concepts that are used throughout our work. Section 2.1 introduces our notation and presents a few basic definitions used in the remainder of this thesis. Section 2.2 gives an overview of data structures used to store sparse matrices. Finally, Section 2.3 briefly describes the iterative techniques we use to solve the linear systems that arise in our work. For in-depth treatment of these topics, we refer the reader to [58, 83, 96] on the subject of sparse matrix storage, and to [6, 47, 96, 103] on the subject of solving linear systems.

2.1 Notation and basic definitions

Sets are denoted in upper-case calligraphic letters, such as $\mathcal{S}$. The fundamental sets are exceptions: the set of naturals is denoted $\mathbb{N}$, the set of reals is denoted $\mathbb{R}$, and the sets of positive and non-negative reals are denoted $\mathbb{R}_+$ and $\mathbb{R}_*$, respectively. The minimal and maximal elements of a set $\mathcal{S}$ of reals are denoted by $\min(\mathcal{S})$ and $\max(\mathcal{S})$, respectively.

Matrices are written in upper-case bold letters, such as $\mathbf{M}$. We say a real matrix $\mathbf{M} \in \mathbb{R}^{m \times n}$ has m rows and n columns. The identity matrix of size $n \times n$ is denoted as

$\mathbf{I}_n$, although if the size is clear from the context it will be written as simply $\mathbf{I}$. The matrix $\mathbf{1}^{m\times n}$ ($\mathbf{0}^{m\times n}$) is the matrix of all ones (zeroes) with m rows and n columns, although it will be written as simply $\mathbf{1}$ ($\mathbf{0}$) if the size is clear from the context. The matrix element at row i and column j is denoted as $\mathbf{M}[i,j]$, for $i \in \{0,\ldots,m-1\}$ and $j \in \{0,\ldots,n-1\}$. A set is used to indicate more than one row or column. For example, $\mathbf{M}[\mathcal{I}, \mathcal{J}]$ refers to the submatrix of $\mathbf{M}$ with rows $\mathcal{I}$ and columns $\mathcal{J}$. Row i (column j) of matrix $\mathbf{M}$ is denoted $\mathbf{M}[i,\bullet]$ ($\mathbf{M}[\bullet,j]$). The number of non-zero elements in matrix $\mathbf{M}$ is denoted $\eta(\mathbf{M})$. The transpose of matrix $\mathbf{M}$ is denoted $\mathbf{M}^{\mathrm{T}}$. The inverse of a square matrix $\mathbf{M}$ is denoted $\mathbf{M}^{-1}$.

A matrix with a single column (row) is called a column (row) vector. Vectors will be denoted with lower-case bold letters, such as $\mathbf{x}$. Elements of a vector are denoted as $\mathbf{x}[i]$. If $\mathbf{x}$ is a column vector, then $\mathbf{x}[i] \equiv \mathbf{x}[i,0]$ otherwise $\mathbf{x}[i] \equiv \mathbf{x}[0,i]$. The same notation is used for both row and column vectors, as usually it is clear from the context if a vector is a row or column vector. A probability vector is a vector whose elements are non-negative and sum to one:

$$\mathbf{x} \in I\!R^n_* \quad \wedge \quad \sum_{i=0}^{n-1} \mathbf{x}[i] = 1.$$

The dot product of two vectors $\mathbf{x}, \mathbf{y} \in I\!R^n$ is defined as

$$\mathbf{x} \cdot \mathbf{y} = \sum_{i=0}^{n-1} \mathbf{x}[i]\mathbf{y}[i].$$

RowSum$(\mathbf{M})$ is the vector whose elements are the row sums of matrix $\mathbf{M}$. That is, if $\mathbf{M} \in I\!R^{m\times n}$ then *RowSum*$(\mathbf{M}) = \mathbf{M} \cdot \mathbf{1}^{n\times 1} \in I\!R^m$. *Diag* $(\mathbf{x})$ is the square matrix with $\mathbf{x}$ along the diagonal and zeroes elsewhere:

$$\mathit{Diag}\,(\mathbf{x})\,[i,j] \equiv \begin{cases} \mathbf{x}[i] & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases} .$$

2.2 Sparse matrix storage

A matrix $\mathbf{M} \in \mathbb{R}^{m \times n}$ can be stored using *full storage*, in which every element of $\mathbf{M}$ is explicitly stored. This is typically done using a two-dimensional array of size $m \times n$, or a one-dimensional array of size mn. In either case, full storage requires exactly $m \cdot n \cdot b_f$ bits of memory, where b_f is the number of bits to store a floating-point number of desired precision. A matrix is called *sparse* if it contains relatively few non-zero entries: $\eta(\mathbf{M}) \ll m \cdot n$. Memory can be conserved by storing only the non-zero elements of sparse matrices [83, 96]. To do so, we must also store some indexing information. Thus, sparse-storage structures may be inefficient when applied to dense matrices due to the overhead of the indexing information. The amount of "sparseness" required for a sparse-storage structure to be more memory-efficient than full storage depends on the structure, the number of bits required for floating-point representation, and other factors.

One way to represent a sparse matrix is to use a linked list for each row, which stores only the non-zero elements of that row. Each node in the list stores a column index and the associated value for that column. We say such a matrix is stored in sparse *row-wise* format. While it is relatively easy to access a row of a matrix stored in this format, it is not so easy to access a column of the matrix. If we require column access, we can use sparse *column-wise* format. This is essentially the same structure, except each list stores the non-zero elements of a column. If we require both row and column access, we can store linked lists for both the rows and the columns [58]. Alternatively, we can convert from row-wise format to column-wise format in $O(\eta(\mathbf{M}))$ operations.

If we have prior knowledge of $\eta(\mathbf{M})$, the rows or columns can be represented using

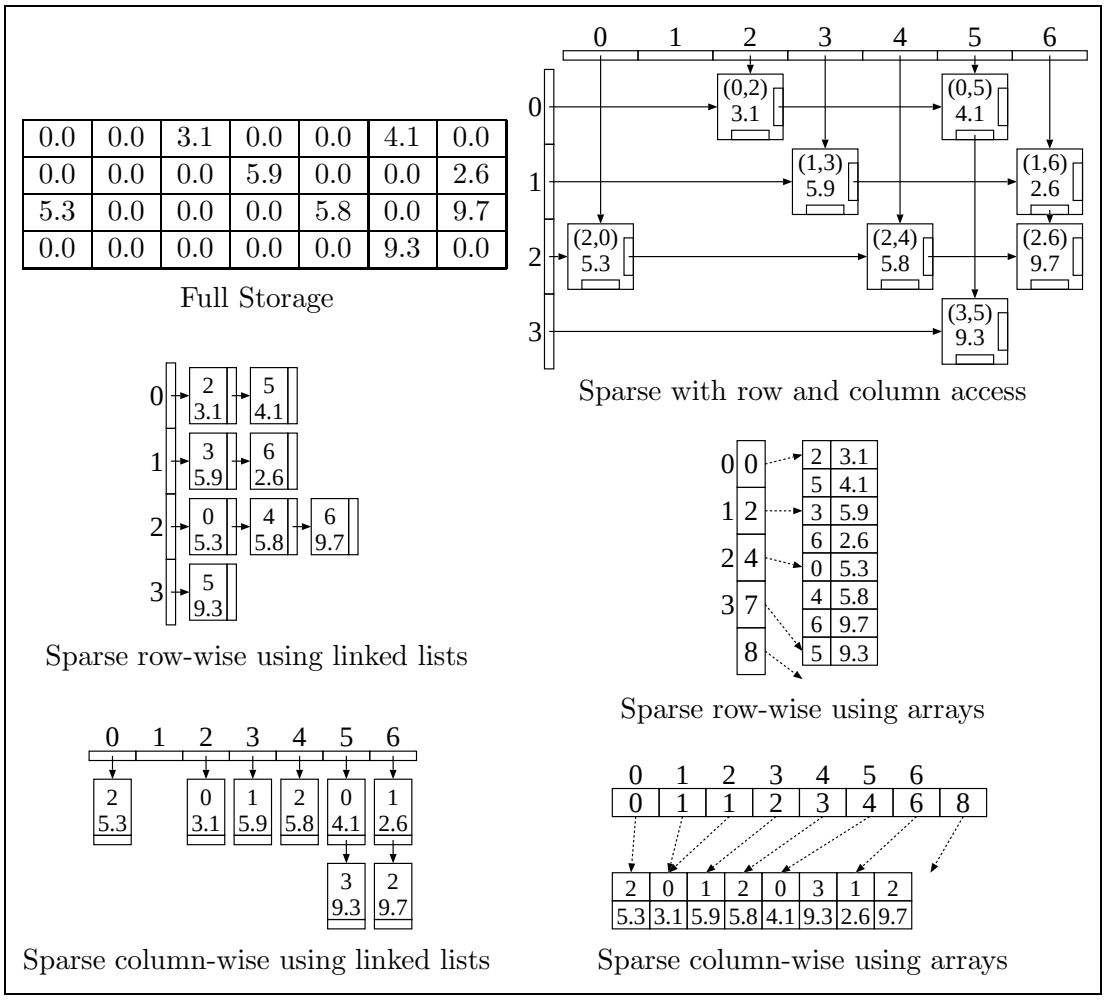


Figure 2.1: Storage of a sparse matrix

arrays instead of linked lists. For efficiency, instead of using a separate array for each row or column, we use a single array. In place of the pointers to linked lists, we maintain array indices to the first non-zero element of each row or column. To mark the last non-zero element of the matrix, an extra index is added.

An example illustrating the data structures used for sparse storage of a matrix is given in Figure 2.1. Each structure represents the same 4×7 matrix. For clarity, null pointers are not drawn. The storage requirement for each structure is shown in Table 2.1. Note

Storage of matrix $\mathbf{M} \in I\!R^{m \times n}$, where:
b_f = # bits for a floating-point number of desired precision
b_p = # bits for a pointer
b_i = # bits for an integer of appropriate size

Technique	**Memory**
Full storage	mnb_f
By rows with linked lists	$mb_p + \eta(\mathbf{M})(b_i + b_f + b_p)$
By columns with linked lists	$nb_p + \eta(\mathbf{M})(b_i + b_f + b_p)$
By rows and columns with linked lists	$mb_p + nb_p + \eta(\mathbf{M})(2b_i + b_f + 2b_p)$
By rows with arrays	$(m+1)b_i + \eta(\mathbf{M})(b_i + b_f)$
By columns with arrays	$(n+1)b_i + \eta(\mathbf{M})(b_i + b_f)$

Table 2.1: Memory required to store a matrix

that integers of size one or two bytes can be used if the number of rows, columns, and the number of non-zero elements is sufficiently small. Further memory savings can be achieved by using the minimal number of bits to store each integer. For instance, if sparse, column-wise storage is used with arrays, the row indices can be stored in $\lceil \log_2(m) \rceil$ bits, and the "pointers" to the elements can be stored in $\lceil \log_2(\eta(\mathbf{M}) + 1) \rceil$ bits. Also, note that the sparse structures described do not work well for "ultra-sparse" matrices, in which many rows and columns are empty. Row-wise, sparse-storage structures can be modified to store only the non-empty rows, and these modified structures will conserve memory if most of the rows are empty. Similar modifications can be made for column-wise storage.

Another important benefit to using sparse-storage structures is the savings in computational complexity. A frequently used matrix operation is that of vector-matrix multiplication. Given a matrix $\mathbf{M} \in I\!R^{m \times n}$ and vectors $\mathbf{x} \in I\!R^m, \mathbf{y} \in I\!R^n$, all stored using full storage, the cost of computing $\mathbf{xM}$ or $\mathbf{My}$ is mn floating-point multiplications. However, multiplication algorithms for sparse matrices require only $\eta(\mathbf{M})$ floating-point multiplica-

tions, assuming the matrix is stored using a sparse structure. For large, sparse matrices this difference is substantial.

2.3 Solving linear systems

Many computations of interest will require us to solve a linear system of equations of the form $\mathbf{xM} = \mathbf{y}$ for an unknown vector $\mathbf{x}$. This form can be rearranged by

$$\begin{aligned} \mathbf{xM} &= \mathbf{y} \\ (\mathbf{xM})^{\mathrm{T}} &= \mathbf{y}^{\mathrm{T}} \\ \mathbf{M}^{\mathrm{T}}\mathbf{x}^{\mathrm{T}} &= \mathbf{y}^{\mathrm{T}} \end{aligned}$$

to obtain the preferred form $\mathbf{Ax} = \mathbf{b}$. It is important to note that the techniques we discuss apply to $\mathbf{Ax} = \mathbf{b}$; thus if our solution technique requires row access of $\mathbf{A}$, then this translates to column access of $\mathbf{M}$.

Solution of the linear system $\mathbf{Ax} = \mathbf{b}$ is a thoroughly discussed problem [6, 47, 96, 103] and several techniques are available. For our applications, $\mathbf{A}$ is typically a very large, extremely sparse, square matrix. Usually, we do not use techniques that compute the inverse of $\mathbf{A}$, for two reasons.

1. Time requirements: Computing the inverse of an $n \times n$ matrix requires $O(n^3)$ floating-point operations.

2. Memory requirements: Since $\mathbf{A}$ is sparse, it can be stored using $O(\eta(\mathbf{A}))$ memory. However, the inverse of a sparse matrix is not necessarily sparse (and usually is not sparse), so storage of $\mathbf{A}^{-1}$ will require $O(n^2)$ memory.

Instead, we prefer "indirect" techniques that perform a series of matrix-vector multiplications, each requiring $O(\eta(\mathbf{A}))$ floating-point operations. Since the matrix $\mathbf{A}$ is never modified, relatively low-precision floating-point representation can be used for $\mathbf{A}$. This combined with a sparse-storage structure results in significant memory savings.

We consider iterative techniques that compute a sequence $\mathbf{x}_n$ of approximations to $\mathbf{x}$. Given an initial guess $\mathbf{x}_0$, the remainder of the sequence is computed from an equation of the form

$$\mathbf{x}_{n+1} = \mathbf{B}\mathbf{x}_n + \mathbf{k} \tag{2.1}$$

where the matrix $\mathbf{B}$ and the vector $\mathbf{k}$ are specified by our iterative technique. The sequence is guaranteed to converge for any initial guess $\mathbf{x}_0$, provided

$$\lim_{n\to\infty} \mathbf{B}^n = \mathbf{0}.$$

This occurs when $\rho(\mathbf{B})$, the largest eigenvalue of $\mathbf{B}$, is strictly less than one. The asymptotic rate of convergence depends on $\rho(\mathbf{B})$: the smaller the value of $\rho(\mathbf{B})$, the faster the sequence computed by Equation 2.1 is likely to converge.

Of course, we cannot compute $\mathbf{x}_\infty$; instead we must compute $\mathbf{x}_N$ for some large value of N, and hope that $\mathbf{x}_N$ is an accurate approximation to $\mathbf{x}$. The number of iterations required to satisfy a tolerance ϵ can be obtained by the approximate relationship [96]

$$\rho(\mathbf{B})^N = \epsilon,$$

which is not used in practice, since $\rho(\mathbf{B})$ is usually not known. Instead, the technique used most often in practice is to somehow compare successive vectors. One technique

frequently used is to compare some norm of the difference of successive vectors with a desired tolerance ϵ. The iterations then continue until an *absolute* precision has been achieved:

$$\|\mathbf{x}_N - \mathbf{x}_{N-1}\| < \epsilon.$$

Alternatively, we can use a *relative* measure. The technique we use is to continue iterations until the maximum relative difference between elements of $\mathbf{x}_N$ and $\mathbf{x}_{N-1}$ is within the desired precision:

$$\max_i \left| \frac{\mathbf{x}_N[i] - \mathbf{x}_{N-1}[i]}{\mathbf{x}_N[i]} \right| < \epsilon.$$

Relative precision is safer to use when entries in the vector $\mathbf{x}$ differ in size by orders of magnitude. This is not uncommon, especially in computing probability vectors for Markov chains.

Another frequently used technique is that of *residual* testing. Since we are solving the system $\mathbf{Ax} = \mathbf{b}$, the idea is that $\mathbf{Ax}_N$ will be "close" to $\mathbf{b}$ if $\mathbf{x}_N$ is "close" to $\mathbf{x}$. Of course, if $\mathbf{A}$ is a large matrix, the cost of computing the residual can be high. Residual testing may not work well for ill-conditioned systems.

2.3.1 Jacobi and Gauss-Seidel

Conceptually, we split the matrix $\mathbf{A}$ into matrices $\mathbf{L}$, $\mathbf{D}$, and $\mathbf{U}$ such that $\mathbf{A} = \mathbf{D} - \mathbf{L} - \mathbf{U}$, where $\mathbf{L}$ and $\mathbf{U}$ are strictly lower- and upper-triangular matrices, respectively, and $\mathbf{D}$ is a diagonal matrix. Thus we have the system $\mathbf{Dx} - \mathbf{Lx} - \mathbf{Ux} = \mathbf{b}$.

For the Jacobi technique, we use the following iteration:

$$\mathbf{Dx}_{n+1} - \mathbf{Lx}_n - \mathbf{Ux}_n = \mathbf{b}.$$

RowJacobi($\mathbf{x}_{old}, \mathbf{x}_{new}, \mathbf{A}', \mathbf{d}, \mathbf{b}$)

- Inputs: vector $\mathbf{x}_{old}$, the current probability vector $\mathbf{x}_n$; matrix $\mathbf{A}' = \mathbf{A} - \mathbf{D}$; vector $\mathbf{d}$, where $\mathbf{D} = Diag\,(\mathbf{d})$; and vector $\mathbf{b}$.
- Output: vector $\mathbf{x}_{new}$, the next probability vector $\mathbf{x}_{n+1}$.

1: **for each** row r **do**
2: $\quad \mathbf{x}_{new}[r] \leftarrow \frac{1}{\mathbf{d}[r]} \left(\mathbf{b}[r] - \mathbf{A}'[r, \bullet] \cdot \mathbf{x}_{old}\right)$ • Dot product of $\mathbf{A}'[r, \bullet]$ and $\mathbf{x}_{old}$
3: **end for**

Algorithm 2.1: Computing a Jacobi iteration by rows

In this case we can compute $\mathbf{x}_{n+1}$ using

$$\mathbf{x}_{n+1} = \mathbf{D}^{-1}(\mathbf{L} + \mathbf{U})\mathbf{x}_n + \mathbf{D}^{-1}\mathbf{b}$$

where $\mathbf{D}^{-1}$ is trivial to compute since $\mathbf{D}$ is a diagonal matrix. A single Jacobi iteration can be computed either using Algorithm 2.1, which accesses elements of $\mathbf{A}$ by rows, or using Algorithm 2.2, which accesses elements of $\mathbf{A}$ by columns. In the algorithms, we store the diagonal elements of $\mathbf{A}$ separately in a vector $\mathbf{d}$. Thus, matrix $\mathbf{A}$ is represented by the two structures $\mathbf{A}'$ and $\mathbf{d}$, where $\mathbf{A}' = \mathbf{A} - Diag\,(\mathbf{d})$. Another common practice is to store $\mathbf{d}^{-1}$, where $\mathbf{D}^{-1} = Diag\,(\mathbf{d}^{-1})$; in that case, the divisions by $\mathbf{d}[i]$ in Algorithm 2.1 and Algorithm 2.2 are replaced with multiplications by $\mathbf{d}^{-1}[i]$.

Jacobi does not use the newest approximation of $\mathbf{x}$ during the computation. That is, once we have determined $\mathbf{x}_{n+1}[i]$, we do not use it until we compute $\mathbf{x}_{n+2}$. Thus Jacobi is insensitive to the ordering of the rows and columns of $\mathbf{A}$. However, it makes sense to use $\mathbf{x}_{n+1}[i]$ if it is known when computing the remaining entries of $\mathbf{x}_{n+1}$. This is the idea

ColJacobi($\mathbf{x}_{old}, \mathbf{x}_{new}, \mathbf{A}', \mathbf{d}, \mathbf{b}$)

- Inputs: vector $\mathbf{x}_{old}$, the current probability vector $\mathbf{x}_n$; matrix $\mathbf{A}' = \mathbf{A} - \mathbf{D}$; vector $\mathbf{d}$, where $\mathbf{D} = Diag\,(\mathbf{d})$; and vector $\mathbf{b}$.
- Output: vector $\mathbf{x}_{new}$, the next probability vector $\mathbf{x}_{n+1}$.

1: $\mathbf{x}_{new} \leftarrow \mathbf{b}$
2: **for each** column c **do**
3: $\quad \mathbf{x}_{new} \leftarrow \mathbf{x}_{new} - \mathbf{x}_{old}[c]\mathbf{A}'[\bullet, c]$ • Vector equation
4: **end for**
5: **for each** column c **do**
6: $\quad \mathbf{x}_{new}[c] \leftarrow \dfrac{\mathbf{x}_{new}[c]}{\mathbf{d}[c]}$
7: **end for**

Algorithm 2.2: Computing a Jacobi iteration by columns

behind the Gauss-Seidel iteration. Formally, we have

$$\mathbf{D}\mathbf{x}_{n+1} - \mathbf{L}\mathbf{x}_{n+1} - \mathbf{U}\mathbf{x}_n = \mathbf{b}$$

which can also be written

$$\mathbf{x}_{n+1} = (\mathbf{D} - \mathbf{L})^{-1}\mathbf{U}\mathbf{x}_n + (\mathbf{D} - \mathbf{L})^{-1}\mathbf{b}$$

although in practice we do not compute the inverse of $\mathbf{D} - \mathbf{L}$. Since $\mathbf{x}_{n+1}[i+1]$ must be computed after we have computed $\mathbf{x}_{n+1}[i]$, Gauss-Seidel is usually implemented using row access of $\mathbf{A}$, as in Algorithm 2.3. One benefit of row Gauss-Seidel is that we only need to store a single vector $\mathbf{x}$, in which $\mathbf{x}_{n+1}[i]$ overwrites $\mathbf{x}_n[i]$. This is possible because $\mathbf{x}_n[i]$ is no longer used once $\mathbf{x}_{n+1}[i]$ has been computed. An algorithm for Gauss-Seidel that uses column access of $\mathbf{A}$ was recently developed in [39]; this algorithm requires an auxiliary vector $\mathbf{w}$ in addition to the single vector $\mathbf{x}$.

RowGaussSeidel($\mathbf{x}, \mathbf{A}', \mathbf{d}, \mathbf{b}$)

- Inputs: vector $\mathbf{x}$, the current probability vector $\mathbf{x}_n$; matrix $\mathbf{A}' = \mathbf{A} - \mathbf{D}$; vector $\mathbf{d}$, where $\mathbf{D} = Diag\,(\mathbf{d})$; and vector $\mathbf{b}$.
- Output: vector $\mathbf{x}$ (overwritten), the next probability vector $\mathbf{x}_{n+1}$.

1: **for each** row r **do**
2: $\quad \mathbf{x}[r] \leftarrow \frac{1}{\mathbf{d}[r]} \left(\mathbf{b}[r] - \mathbf{A}'[r, \bullet] \cdot \mathbf{x}\right)$ • Dot product of $\mathbf{A}'[r, \bullet]$ and $\mathbf{x}$
3: **end for**

Algorithm 2.3: Computing a Gauss-Seidel iteration by rows

Both the Jacobi and Gauss-Seidel iterations fall under the type of Equation 2.1; for Jacobi we have $\mathbf{B}_{Jac} = \mathbf{D}^{-1}(\mathbf{L}+\mathbf{U})$, and for Gauss-Seidel we have $\mathbf{B}_{GS} = (\mathbf{D} - \mathbf{L})^{-1}\mathbf{U}$. The Stein-Rosenberg theorem [103] states that for non-negative Jacobi matrices $\mathbf{B}_{Jac}$, exactly one of the following statements holds.

1. $\rho(\mathbf{B}_{Jac}) = \rho(\mathbf{B}_{GS}) = 0$.
2. $0 < \rho(\mathbf{B}_{GS}) < \rho(\mathbf{B}_{Jac}) < 1$.
3. $\rho(\mathbf{B}_{Jac}) = \rho(\mathbf{B}_{GS}) = 1$.
4. $1 < \rho(\mathbf{B}_{Jac}) < \rho(\mathbf{B}_{GS})$.

Thus if the Jacobi matrix $\mathbf{D}^{-1}(\mathbf{L} + \mathbf{U})$ is non-negative, then Jacobi and Gauss-Seidel will either both converge or both diverge. Furthermore, if both techniques converge, then Gauss-Seidel has a faster asymptotic rate of convergence (meaning that Gauss-Seidel is expected to converge faster). Since updated values are used immediately with Gauss-Seidel, the variable ordering may affect the rate of convergence [96]. In contrast, the convergence rate of Jacobi is independent of the variable ordering.

Both the Jacobi and Gauss-Seidel techniques can make use of *relaxation* [96, 103]. The idea is that for each iteration, we are changing our approximation of $\mathbf{x}$ by

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \boldsymbol{\delta}(\mathbf{x}_n)$$

where $\boldsymbol{\delta}(\mathbf{x}_n)$ is determined by our iterative technique. We can alter the speed of convergence by instead computing

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \omega\boldsymbol{\delta}(\mathbf{x}_n)$$

where ω is called the relaxation parameter. If we use $1 < \omega < 2$ ($0 < \omega < 1$), then the technique is called over-relaxation (under-relaxation). Determining an optimal value for ω is not a trivial task.

Another popular group of techniques to solve a linear system $\mathbf{Ax} = \mathbf{b}$ are *projection* techniques, in which an exact solution is approximated from a sequence of approximations taken from an m-dimension subspace [96]. While these techniques are quite sophisticated, they require storage of m vectors in addition to the solution vector $\mathbf{x}$. As we will see in later chapters, the size of the solution vector can become quite large; thus the storage of m additional vectors is often not possible due to excessive memory requirements.

Chapter 3

Markov Chains

This chapter presents some mathematical background on the concept of "randomness", which is fundamental to much of our work. Section 3.1 discusses random variables and describes the important distribution functions for our work. Section 3.2 continues the discussion with an overview of stochastic processes (families of random variables) and Markov chains (a special type of stochastic process). For more detail on this material, we refer the reader to [31, 89]. Section 3.3 and Section 3.4 give thorough discussions on discrete-time and continuous-time Markov chains, which are critical to our work. For more on Markov chains, the reader is referred to [55], an excellent treatment on discrete-time Markov chains, and to [89, 96]. Finally, Section 3.5 gives a brief overview of phase-type random variables, which use Markov chains in their definition. This topic is covered particularly well in [75].

3.1 Random variables and important distributions

Suppose we conduct some experiment. The set of all possible outcomes of the experiment is called the sample space. Given some sample space $\mathcal{W}$, a random variable [31, 89] is a function $X : \mathcal{W} \to \mathcal{S}$. If the set $\mathcal{S}$ is countable, X is a discrete random variable, otherwise

X is a continuous random variable. Typically, if X is discrete then $\mathcal{S} \subseteq \mathbb{N}$, and if X is continuous then $\mathcal{S} \subseteq \mathbb{R}$. Random variables are written in upper-case.

A discrete random variable can be completely described by its probability distribution or *probability mass function* [89], which specifies the probability of each possible value of X. An important point is that two random variables X and Y may have the same probability distribution, but this does not imply that X and Y are equal. Arguably the simplest distribution is when the random variable is not random at all: random variable X is said to be a constant x, written $X \sim \text{Const}(x)$, if

$$\Pr\{X = n\} = \begin{cases} 1 & \text{if } n = x \\ 0 & \text{otherwise} \end{cases} .$$

A constant random variable is then just what its name implies: a random variable that is only allowed to take on a single value. A more interesting distribution is the Bernoulli distribution: random variable X is said to be a Bernoulli random variable with success parameter p, written $X \sim \text{Bernoulli}(p)$, if

$$\Pr\{X = n\} = \begin{cases} 1-p & \text{if } n = 0 \\ p & \text{if } n = 1 \\ 0 & \text{otherwise} \end{cases}$$

where $0 \leq p \leq 1$. Thus, a Bernoulli(p) random variable can take on values 0 and 1, except for the limiting cases Bernoulli(0) = Const(0) and Bernoulli(1) = Const(1).

Consider an infinite sequence of independent Bernoulli random variables $X_1, X_2, \ldots$ all with success parameter p. Let J be the position of the first occurrence of the value 1. That is, $X_J = 1$ and for all $0 \leq i < J, X_i = 0$. Then J is said to be a Geometric random variable

with success parameter p, written $J \sim \text{Geom}(p)$, and

$$\Pr\{J = n\} = \begin{cases} (1-p)^{n-1}p & \text{if } n > 0 \\ 0 & \text{otherwise} \end{cases}.$$

Let K be the number of 0's before the first 1. Note that K is one less than J. Then K is said to be a Modified Geometric random variable with success parameter p, written $K \sim \text{ModGeom}(p)$, and

$$\Pr\{K = n\} = \begin{cases} (1-p)^{n}p & \text{if } n \geq 0 \\ 0 & \text{otherwise} \end{cases}.$$

Let Y be the sum of the first n variables, $Y = \sum_{i=1}^{n} X_i$. Then Y is said to be a Binomial random variable with parameters n and p, written $Y \sim \text{Binomial}(n, p)$, and

$$\Pr\{Y = i\} = \begin{cases} \left(\begin{array}{c} n \\ i \end{array} \right) p^i (1-p)^{n-i} & \text{if } 0 \leq i \leq n \\ 0 & \text{otherwise} \end{cases}.$$

Consider the limiting case of a Binomial random variable where $n \to \infty$ and $p \to 0$ such that the product np remains a constant λ. This is the Poisson distribution with parameter λ, written Poisson(λ). If $Z \sim \text{Poisson}(\lambda)$, then we have

$$\begin{aligned} \Pr\{Z = i\} &= \lim_{n\to\infty, p\to 0, np=\lambda} \left(\begin{array}{c} n \\ i \end{array} \right) p^i (1-p)^{n-i} \\ &= \lim_{n\to\infty, p\to 0, np=\lambda} \frac{1}{i!} \frac{n!}{(n-i)!} p^i (1-p)^{-i} (1-p)^n \\ &= \lim_{n\to\infty, p\to 0, np=\lambda} \frac{1}{i!} np\,(n-1)p \cdots (n-i+1)p\,(1-p)^{-i} \left(1 - \frac{\lambda}{n}\right)^n \\ &= \frac{1}{i!} \lambda^i\,(1)^{-i}\,e^{-\lambda} \\ &= \frac{\lambda^i}{i!} e^{-\lambda}. \end{aligned}$$

With continuous random variables, the distribution cannot be specified by the probabilities for each possible value of X. Since X can take on an uncountably infinite number of

values, the probability of X taking on a single value is always zero. Instead, a continuous random variable is described by either its *probability density function* or by its cumulative distribution function (CDF) [89], which specifies $\Pr\{X \leq x\}$ for every value of $x \in \mathbb{R}$. A continuous random variable X is said to be an Exponential random variable with rate λ, written $X \sim \text{Expo}(\lambda)$, if

$$\Pr\{X \leq x\} = \begin{cases} 1 - e^{-\lambda x} & \text{if } x \geq 0 \\ 0 & \text{if } x \leq 0 \end{cases}.$$

In our work we are primarily interested in the Exponential distribution. The Exponential distribution is often used to build other distributions. For instance, a continuous random variable Y is said to be an Erlang random variable with n stages and parameter μ, written $Y \sim \text{Erlang}(n, \mu)$, if

$$Y = X_1 + \cdots + X_n,$$

where $X_1, \ldots, X_n$ are independent and identically distributed random variables with distribution $\text{Expo}(\mu)$.

We are especially interested in distributions that satisfy the *memoryless* property

$$\Pr\{X > s + t | X > t\} = \Pr\{X > s\} \tag{3.1}$$

for any non-negative values of s and t. If $X \sim \text{Geom}(p)$, then we have

$$\begin{aligned} \Pr\{X > n\} &= \Pr\{\text{The first } n \text{ Bernoulli random variables are zero}\} \\ &= (1-p)^n, \end{aligned}$$

which gives us

$$\begin{aligned}
\Pr\{X > s+t | X > t\} &= \frac{\Pr\{X > s+t \wedge X > t\}}{\Pr\{X > t\}} \\
&= \frac{\Pr\{X > s+t\}}{\Pr\{X > t\}} \\
&= \frac{(1-p)^{s+t}}{(1-p)^t} \\
&= (1-p)^s \\
&= \Pr\{X > s\},
\end{aligned}$$

thus satisfying Equation 3.1. If $X \sim \text{Expo}(\lambda)$, then we have

$$\begin{aligned}
\Pr\{X > s+t | X > t\} &= \frac{\Pr\{X > s+t\}}{\Pr\{X > t\}} \\
&= \frac{e^{-\lambda(s+t)}}{e^{-\lambda t}} \\
&= e^{-\lambda s} \\
&= \Pr\{X > s\}
\end{aligned}$$

and Equation 3.1 is satisfied. Thus the Geometric and Exponential distributions are memoryless, and it can be shown that they are the only memoryless distributions.

The Exponential distribution is closely related to both the Geometric and Poisson distributions. Suppose we have a sequence of independent random variables $X_1, X_2, \ldots$ that are all exponentially distributed with rate λ. Let N be a Geometric random variable with success parameter p. Then the random variable Y given by

$$Y = \sum_{i=1}^{N} X_i$$

is an Exponential random variable with rate λp. If J is the integer such that

$$\sum_{i=1}^{J} X_i \leq 1 < \sum_{i=1}^{J+1} X_i \tag{3.2}$$

then J is a Poisson random variable with parameter λ.

3.2 Stochastic processes and Markov chains

A *stochastic process* [31, 89] is a collection of random variables $\{X(t) : t \in \mathcal{T}\}$. The set $\mathcal{S}$ of possible values for $X(t)$ is called the *state space*. The parameter t is often considered to be time. If $\mathcal{T}$ is countable, the stochastic process is a discrete-time process; otherwise it is a continuous-time process. We say the process is in state $s \in \mathcal{S}$ at time $t \in \mathcal{T}$ if $X(t) = s$.

Markov processes are special cases of stochastic processes that obey the memoryless or Markovian property: only the current state of the process determines the probability or rate of switching to another state [96]. This is expressed formally as

$\forall n \in \mathbb{N}, \quad \forall i_0, \ldots, i_{n+1} \in \mathcal{S}, \forall t_0, \ldots, t_{n+1} \in \mathcal{T}, t_0 < \cdots < t_{n+1},$

$$\Pr\{X(t_{n+1}) = i_{n+1} | X(t_n) = i_n \wedge \cdots \wedge X(t_0) = i_0\} = \Pr\{X(t_{n+1}) = i_{n+1} | X(t_n) = i_n\}.$$

A Markov process with a discrete state space $\mathcal{S}$ is called a *Markov chain*. A Markov chain whose transition probabilities do not depend on time is called *homogeneous*. We will limit our discussion to homogeneous Markov chains with finite state spaces.

The states of a Markov chain can be classified based on their ability to reach other states [55]. In a Markov chain, we say state j is *reachable* from state i if

$$\Pr\{X(t) = j | X(0) = i\} > 0$$

for some time $t > 0$. A state i is called *transient* if there exists a state j such that j is reachable from i but i is not reachable from j. Conversely, a state i is called *recurrent* if for every state j reachable from i, i is reachable from j. A recurrent state is called *absorbing* if no other state is reachable from it. Two states i and j are said to be *mutually reachable* if i is reachable from j and j is reachable from i. The equivalence relation "mutually reachable" creates equivalence classes over the set of states, where all states in a given class are either transient or recurrent. A set of states $\mathcal{X} \subseteq \mathcal{S}$ is called a *recurrent class* if all pairs of states in $\mathcal{X}$ are mutually reachable and no state outside of $\mathcal{X}$ is reachable from a state in $\mathcal{X}$. A Markov chain is called *irreducible* if $\mathcal{S}$ is a recurrent class. Note that an absorbing state is in its own recurrent class.

3.3 Discrete-time Markov chains

A discrete-time Markov chain (DTMC) is a Markov chain with a discrete set of time indices [96]. That is, the set $\mathcal{T}$ is countable and so without loss of generality we assume $\mathcal{T} = \mathbb{N}$, and we write $\{X(n) : n \in \mathbb{N}\}$. A DTMC is completely described by its probability matrices $\mathbf{P}_n$, where

$$\mathbf{P}_n[i,j] = \Pr\{X(n+1) = j | X(n) = i\}.$$

Note that each row of $\mathbf{P}_n$ sums to one due to the law of total probability. When the Markov chain is homogeneous, we have

$$\forall n \in \mathbb{N}, \quad \mathbf{P}_n = \mathbf{P}.$$

In a discrete-time Markov chain, the *period* [89] of a state i is given by the greatest common divisor of all the integers k such that $\Pr\{X(k) = i | X(0) = i\} > 0$. A state with

period 1 is called *aperiodic*. We say a DTMC is aperiodic if all its states are aperiodic; otherwise, it is periodic. Each recurrent class of a periodic DTMC has its own period, and each state in that class will have that period. A DTMC that is both aperiodic and irreducible is said to be *ergodic*.

A homogeneous DTMC can be depicted as a labeled digraph with adjacency matrix $\mathbf{P}$. Some example DTMCs are shown in Figure 3.1. The DTMC shown in Figure 3.1(a) has state space $\{A, B, C, D, E\}$. State A is transient, state B is absorbing, and states $\{C, D, E\}$ form a recurrent class with period 3. The DTMC shown in Figure 3.1(b) is an ergodic DTMC: the state space $\{W, X, Y, Z\}$ is a recurrent class with period 1. The DTMC shown in Figure 3.1(c) has absorbing states $\{A, B, C\}$ and has transient states $\{X, Y, Z\}$.

3.3.1 Transient analysis

Let $\mathbf{p}_n$ be a sequence of probability (row) vectors such that

$$\mathbf{p}_n[i] = \Pr\{X(n) = i\}.$$

Then subsequent probability vectors are related by [55]

$$\forall n \geq 0, \quad \mathbf{p}_{n+1} = \mathbf{p}_n \mathbf{P}. \tag{3.3}$$

Repeated application of Equation 3.3 gives us

$$\mathbf{p}_n = \mathbf{p}_0 \mathbf{P}^n \tag{3.4}$$

where $\mathbf{p}_0$ is the initial probability vector. In practice, however, we use Equation 3.3, since Equation 3.4 requires the computation of $\mathbf{P}^n$.

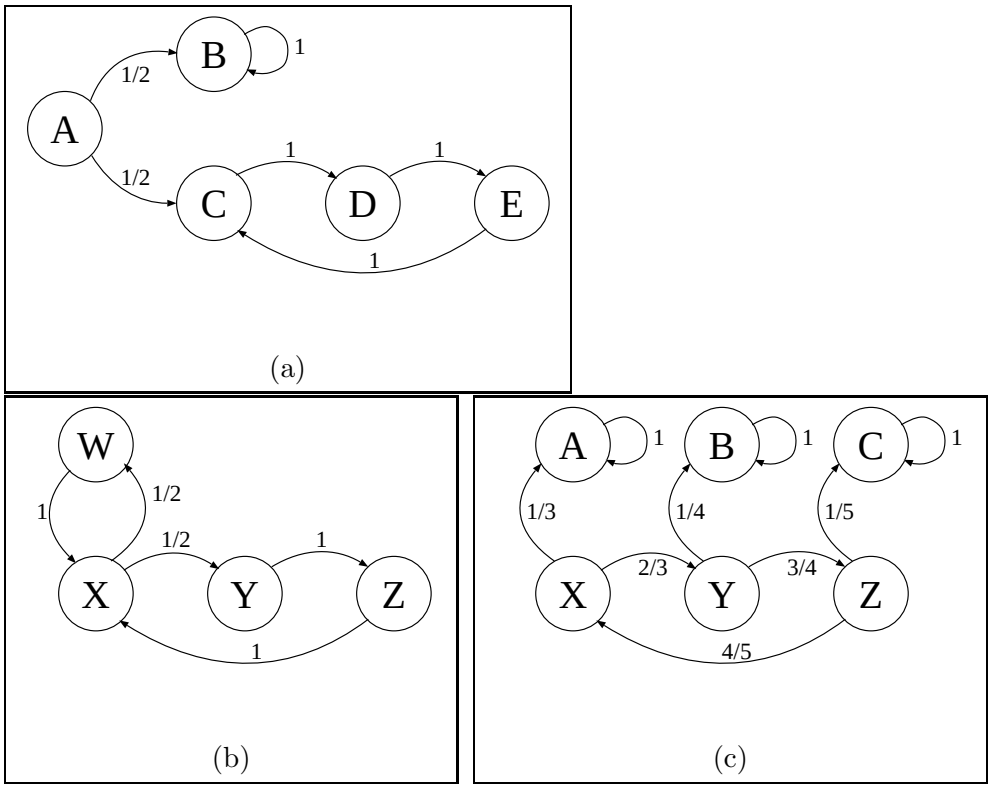


Figure 3.1: Example DTMCs

For example, suppose we want to compute the probability distribution of each state in the DTMC in Figure 3.1(a) at time 6, where

$$\mathbf{p}_0 = \begin{bmatrix} \frac{1}{3} & \frac{1}{3} & 0 & \frac{1}{3} & 0 \end{bmatrix}$$

is the initial probability vector. Using Equation 3.4, we compute

$$\mathbf{P} = \begin{bmatrix} 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix} \quad \mathbf{P}^2 = \begin{bmatrix} 0 & \frac{1}{2} & 0 & \frac{1}{2} & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \quad \mathbf{P}^3 = \begin{bmatrix} 0 & \frac{1}{2} & 0 & 0 & \frac{1}{2} \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{P}^4 = \mathbf{P} \qquad \mathbf{P}^5 = \mathbf{P}^2 \qquad \mathbf{P}^6 = \mathbf{P}^3$$

$$\mathbf{p}_6 = \mathbf{p}_0\mathbf{P}^6 = \begin{bmatrix} 0 & \frac{1}{2} & 0 & \frac{1}{3} & \frac{1}{6} \end{bmatrix}$$

requiring intermediate storage for matrix $\mathbf{P}^6$ (and storage for vector $\mathbf{p}_6$). Note that due to the periodic nature of the DTMC, $\mathbf{P}^{n+1} = \mathbf{P}^{n \bmod 3+1}$ for any natural number n. Alternatively, we can use Equation 3.3 to compute the sequence

$$\begin{aligned}
\mathbf{p}_1 &= \mathbf{p}_0\mathbf{P} = \begin{bmatrix} 0 & \frac{1}{2} & \frac{1}{6} & 0 & \frac{1}{3} \end{bmatrix} \\
\mathbf{p}_2 &= \mathbf{p}_1\mathbf{P} = \begin{bmatrix} 0 & \frac{1}{2} & \frac{1}{3} & \frac{1}{6} & 0 \end{bmatrix} \\
\mathbf{p}_3 &= \mathbf{p}_2\mathbf{P} = \begin{bmatrix} 0 & \frac{1}{2} & 0 & \frac{1}{3} & \frac{1}{6} \end{bmatrix} \\
\mathbf{p}_4 &= \mathbf{p}_3\mathbf{P} = \begin{bmatrix} 0 & \frac{1}{2} & \frac{1}{6} & 0 & \frac{1}{3} \end{bmatrix} = \mathbf{p}_1 \\
\mathbf{p}_5 &= \mathbf{p}_4\mathbf{P} = \begin{bmatrix} 0 & \frac{1}{2} & \frac{1}{3} & \frac{1}{6} & 0 \end{bmatrix} = \mathbf{p}_2 \\
\mathbf{p}_6 &= \mathbf{p}_5\mathbf{P} = \begin{bmatrix} 0 & \frac{1}{2} & 0 & \frac{1}{3} & \frac{1}{6} \end{bmatrix} = \mathbf{p}_3
\end{aligned}$$

which requires storage for two vectors (the previous vector and the one being computed), since we can overwrite vector $\mathbf{p}_n$ with vector $\mathbf{p}_{n+1}$ once it has been computed. Of course, in this example we have

$$\mathbf{p}_{n+1} = \mathbf{p}_0\mathbf{P}^{n+1} = \mathbf{p}_0\mathbf{P}^{n \bmod 3+1} = \mathbf{p}_{n \bmod 3+1} \qquad \forall n \in \mathbb{N}$$

due to the periodic nature of the DTMC. Note that the DTMC will never be in state A after time 0. In general, the probability of being in a transient state tends to zero as time progresses, regardless of the initial probabilities.

3.3.2 Stationary analysis

In an ergodic DTMC,

$$\mathbf{P}^\infty \equiv \lim_{n\to\infty} \mathbf{P}^n = \begin{bmatrix} \mathbf{p} \\ \vdots \\ \mathbf{p} \end{bmatrix} \tag{3.5}$$

where $\mathbf{p}$ is the stationary probability vector [55]. The vector $\mathbf{p}$ can also be defined as

$$\mathbf{p} = \mathbf{p}_0\mathbf{P}^\infty \tag{3.6}$$

$$\mathbf{p} = \lim_{n\to\infty} \mathbf{p}_n \tag{3.7}$$

by substitution into Equation 3.4. Looking at Equation 3.5, we see that $\mathbf{p}$ does not depend on the initial probability vector $\mathbf{p}_0$. Equation 3.6 holds since all rows of $\mathbf{P}^\infty$ are $\mathbf{p}$; thus any probability vector $\mathbf{p}_0$ multiplied by $\mathbf{P}^\infty$ will give $\mathbf{p}$. In particular, Equation 3.6 holds if $\mathbf{p}_0 = \mathbf{p}$, thus we have $\mathbf{p} = \mathbf{pP}$. Note that $\mathbf{p}$ is the only probability vector for which this holds: a probability vector $\mathbf{p}'$ satisfying $\mathbf{p}' = \mathbf{p}'\mathbf{P}$ implies $\mathbf{p}' = \mathbf{p}'\mathbf{P}^n$, thus $\mathbf{p}' = \mathbf{p}'\mathbf{P}^\infty = \mathbf{p}$ from Equation 3.6. Thus $\mathbf{p}$ is the unique probability vector satisfying

$$\mathbf{p}(\mathbf{P} - \mathbf{I}) = \mathbf{0}. \tag{3.8}$$

For example, looking at the ergodic DTMC in Figure 3.1(b), we can compute

$$\mathbf{P} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{bmatrix} \quad \mathbf{P}^{10} = \begin{bmatrix} \frac{7}{32} & \frac{12}{32} & \frac{7}{32} & \frac{6}{32} \\ \frac{6}{32} & \frac{13}{32} & \frac{6}{32} & \frac{7}{32} \\ \frac{6}{32} & \frac{14}{32} & \frac{6}{32} & \frac{6}{32} \\ \frac{7}{32} & \frac{12}{32} & \frac{7}{32} & \frac{6}{32} \end{bmatrix} \quad \mathbf{P}^{20} = \begin{bmatrix} \frac{205}{1024} & \frac{410}{1024} & \frac{205}{1024} & \frac{204}{1024} \\ \frac{205}{1024} & \frac{409}{1024} & \frac{205}{1024} & \frac{205}{1024} \\ \frac{204}{1024} & \frac{410}{1024} & \frac{204}{1024} & \frac{206}{1024} \\ \frac{205}{1024} & \frac{410}{1024} & \frac{205}{1024} & \frac{204}{1024} \end{bmatrix}$$

which shows how quickly the sequence $\mathbf{P}^n$ approaches the limit

$$\mathbf{P}^\infty = \begin{bmatrix} \frac{1}{5} & \frac{2}{5} & \frac{1}{5} & \frac{1}{5} \\ \frac{1}{5} & \frac{2}{5} & \frac{1}{5} & \frac{1}{5} \\ \frac{1}{5} & \frac{2}{5} & \frac{1}{5} & \frac{1}{5} \\ \frac{1}{5} & \frac{2}{5} & \frac{1}{5} & \frac{1}{5} \end{bmatrix}$$

where the stationary probability vector is

$$\mathbf{p} = \begin{bmatrix} \frac{1}{5} & \frac{2}{5} & \frac{1}{5} & \frac{1}{5} \end{bmatrix}.$$

We can confirm the correctness of $\mathbf{P}^\infty$ using the equations

$$\mathbf{P} \cdot \mathbf{P}^\infty = \mathbf{P}^\infty \qquad \text{and} \qquad \mathbf{P}^\infty \cdot \mathbf{P} = \mathbf{P}^\infty.$$

Of course, in practice we do not compute matrix $\mathbf{P}^\infty$ directly; instead we compute $\mathbf{p}$. This can be done using the *power method* [96], which uses Equation 3.3 to compute $\mathbf{p}_n$ for a large value of n as an approximation for $\mathbf{p}$. Another technique is to solve the linear system in Equation 3.8 for $\mathbf{p}$.

Looking at the DTMC in Figure 3.1(a) gives some intuition as to why stationary analysis is only defined for ergodic DTMCs. As illustrated in the previous section, $\mathbf{P}^{n+1} = \mathbf{P}^{n \bmod 3+1}$ and $\mathbf{p}_{n+1} = \mathbf{p}_{n \bmod 3+1}$ for any natural number n. Of course, this implies that the limit in Equation 3.5 does not exist; nor does the limit in Equation 3.7. Note also that the long-term probability of being in a given state depends on the initial probability vector. For instance, if the initial state is state B with probability one, then the DTMC will remain in state B with probability one. If the initial probability vector instead has zero entries except for one or more of the states C, D and E, then the DTMC will remain in those states.

3.3.3 Mean time to absorption

Suppose we have an absorbing DTMC. That is, we have a DTMC where all the states are either transient or absorbing. Let $\mathcal{A}$ be the set of absorbing states, and let $\mathcal{Z} = \mathcal{S} - \mathcal{A}$ be the set of transient states. The matrix

$$\mathbf{N} = \sum_{n=0}^{\infty} \mathbf{P}^n[\mathcal{Z}, \mathcal{Z}] = (\mathbf{I} - \mathbf{P}[\mathcal{Z}, \mathcal{Z}])^{-1} \tag{3.9}$$

is called the fundamental matrix of the DTMC [55]. Since an entry $[i, j]$ of matrix $\mathbf{P}^n$ is the probability of being in state j at time n, given that the DTMC is in state i at time 0, an entry $[i, j]$ of the infinite sum in Equation 3.9 is then the probability accumulated

over all time of being in transient state j given that the DTMC is initially in transient state i. Of course, once the DTMC reaches an absorbing state, it will never again reach transient state j. Thus $\mathbf{N}[i, j]$ is the expected number of times the process is in state j until absorption, given that it starts in state i.

Note that the fundamental matrix refers only to the transient portion of matrix $\mathbf{P}$, since the expected number of visits to an absorbing state will either be zero or infinitely many, depending on the initial state. The expected number of visits to a transient state i is instead finite, because by definition there must be some state j reachable from i where i is not reachable from j. Thus every time state i is reached, there is a positive probability that state j will be reached at a later time, and once the DTMC reaches state j, state i will never again be reached.

The total expected number of visits to a given transient state (before absorption) is given by

$$\mathbf{n} = \mathbf{p}_0[\mathcal{Z}] \cdot \mathbf{N} \tag{3.10}$$

where $\mathbf{n} \in I\!R^{|\mathcal{Z}|}$. In practice, $\mathbf{n}$ is computed by solving the system

$$\mathbf{n}(\mathbf{I} - \mathbf{P}[\mathcal{Z}, \mathcal{Z}]) = \mathbf{p}_0[\mathcal{Z}]$$

which is obtained from Equation 3.10 by multiplying both sides by $\mathbf{N}^{-1}$. The mean time to absorption (MTTA) is then simply the sum of the elements of $\mathbf{n}$.

We can also compute the probability of being absorbed into each of the absorbing states. First, consider the meaning of element $[i, j]$ of the product $\mathbf{P}^n[\mathcal{Z}, \mathcal{Z}] \cdot \mathbf{P}[\mathcal{Z}, \mathcal{A}]$ where the matrix $\mathbf{P}[\mathcal{Z}, \mathcal{A}]$ describes the probabilities of going from transient states to absorbing states.

Element $[i, j]$ is given by the dot product

$$\mathbf{P}^n[i, \mathcal{Z}] \cdot \mathbf{P}[\mathcal{Z}, j] = \sum_{\forall k \in \mathcal{Z}} \mathbf{P}^n[i, k]\mathbf{P}[k, j]$$

where $\mathbf{P}^n[i, k]$ is the probability of being in transient state k at time n if the initial state is i and $\mathbf{P}[k, j]$ is the probability of going to absorbing state j from transient state k in one time step. Thus element $[i, j]$ of $\mathbf{P}^n[\mathcal{Z}, \mathcal{Z}] \cdot \mathbf{P}[\mathcal{Z}, \mathcal{A}]$ is the probability of being absorbed by state j at time $n+1$ if the initial state is i. Summing over all naturals n gives us $\mathbf{N} \cdot \mathbf{P}[\mathcal{Z}, \mathcal{A}]$, where an element $[i, j]$ is the probability of being absorbed by state j if the initial state is i. We can then compute the probability of being absorbed into each absorbing state by

$$\mathbf{a} = \mathbf{p}_0 \cdot \mathbf{N} \cdot \mathbf{P}[\mathcal{Z}, \mathcal{A}] = \mathbf{n} \cdot \mathbf{P}[\mathcal{Z}, \mathcal{A}] \tag{3.11}$$

where $\mathbf{a}$ is a probability vector of size $|\mathcal{A}|$.

For example, looking at the absorbing DTMC in Figure 3.1(c) we can compute the mean time spent in each transient state assuming the initial state is X by first computing the fundamental matrix

$$\mathbf{N} = (\mathbf{I} - \mathbf{P}[\mathcal{Z}, \mathcal{Z}])^{-1} = \left(\mathbf{I} - \begin{bmatrix} 0 & \frac{2}{3} & 0 \\ 0 & 0 & \frac{3}{4} \\ \frac{4}{5} & 0 & 0 \end{bmatrix} \right)^{-1} = \begin{bmatrix} \frac{5}{3} & \frac{10}{9} & \frac{5}{6} \\ 1 & \frac{5}{3} & \frac{5}{4} \\ \frac{4}{3} & \frac{8}{9} & \frac{5}{3} \end{bmatrix}$$

and then computing the vector $\mathbf{n}$

$$\mathbf{n} = \mathbf{p}_0 \cdot \mathbf{N} = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} \cdot \mathbf{N} = \begin{bmatrix} \frac{5}{3} & \frac{10}{9} & \frac{5}{6} \end{bmatrix}.$$

Alternatively, we could solve Equation 3.10 for $\mathbf{n}$, which is the preferred technique for large DTMCs, since it does not require us to compute and store the dense matrix $\mathbf{N}$. In either

case, we can then compute the probability of being absorbed into each absorbing state

$$\mathbf{a} = \mathbf{n} \cdot \mathbf{P}[\mathcal{Z}, \mathcal{A}] = \begin{bmatrix} \frac{30}{18} & \frac{20}{18} & \frac{15}{18} \end{bmatrix} \cdot \begin{bmatrix} \frac{1}{3} & 0 & 0 \\ 0 & \frac{1}{4} & 0 \\ 0 & 0 & \frac{1}{5} \end{bmatrix} = \begin{bmatrix} \frac{10}{18} & \frac{5}{18} & \frac{3}{18} \end{bmatrix}$$

which requires a single vector-matrix multiplication operation once we know $\mathbf{n}$.

3.4 Continuous-time Markov chains

A continuous-time Markov chain (CTMC) is a Markov chain with a continuous set of time indices [96]. Without loss of generality, we assume $\mathcal{T} = \mathbb{R}_*$ and so we write $\{X(t) : t \in \mathbb{R}_*\}$. A CTMC can be described by its transition rate matrices $\mathbf{R}_t$, where

$$\mathbf{R}_t[i,j] = \begin{cases} \lim_{h \to 0} \dfrac{\Pr\{X(t+h) = j | X(t) = i\}}{h} & \text{if } i \neq j \\ 0 & \text{if } i = j \end{cases}$$

When the Markov chain is homogeneous, we have

$$\forall t \in \mathbb{R}_*, \quad \mathbf{R}_t = \mathbf{R}.$$

Alternatively, we can use the infinitesimal generator matrix $\mathbf{Q}$, where

$$\mathbf{Q} \equiv \mathbf{R} - \mathit{Diag}\left(\mathit{RowSum}(\mathbf{R})\right).$$

Note that each row of $\mathbf{Q}$ sums to zero by definition. Also, note that the concept of periodicity does not apply to CTMCs.

As with DTMCs, a homogeneous CTMC can be depicted as a labeled digraph with adjacency matrix $\mathbf{R}$. Some example CTMCs are shown in Figure 3.2. The CTMC shown

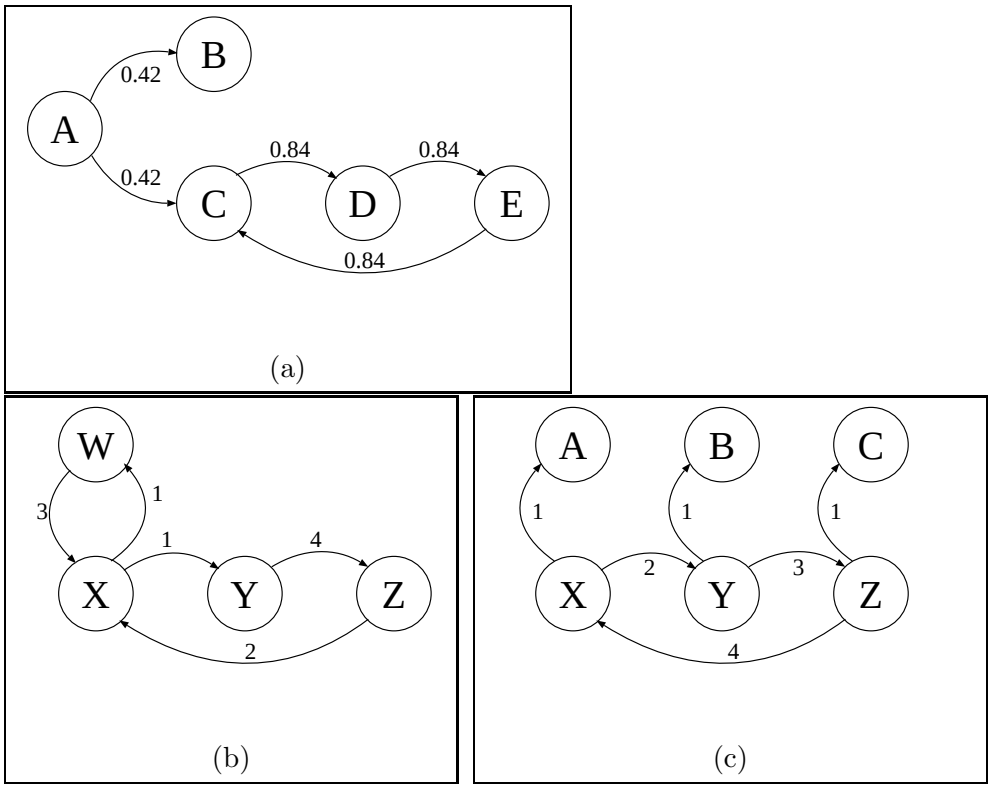


Figure 3.2: Example CTMCs

in Figure 3.2(a) has state space $\{A, B, C, D, E\}$. State A is transient, state B is absorbing, and states $\{C, D, E\}$ form a recurrent class. The CTMC shown in Figure 3.2(b) is an irreducible CTMC: the state space $\{W, X, Y, Z\}$ is a recurrent class. The CTMC shown in Figure 3.2(c) has absorbing states $\{A, B, C\}$ and has transient states $\{X, Y, Z\}$.

3.4.1 Transient analysis

Let $\boldsymbol{\pi}(t)$ be a probability vector such that

$$\boldsymbol{\pi}(t)[i] \equiv \Pr\{X(t) = i\}.$$

Then $\boldsymbol{\pi}(t)$ is the solution of the system of differential equations [96]

$$\frac{d\boldsymbol{\pi}(t)}{dt} = \boldsymbol{\pi}(t)\mathbf{Q}$$

with initial condition $\boldsymbol{\pi}(0)$, which can be expressed as

$$\boldsymbol{\pi}(t) = \boldsymbol{\pi}(0) \cdot e^{\mathbf{Q}t} = \boldsymbol{\pi}(0) \cdot \sum_{n=0}^{\infty} \frac{t^n}{n!}\mathbf{Q}^n. \tag{3.12}$$

Using a process called *uniformization* [89, 96], we can convert a CTMC to a DTMC by defining

$$\mathbf{P} = \frac{\mathbf{Q}}{q} + \mathbf{I} \tag{3.13}$$

where q is chosen such that $q \geq \max(\{-\mathbf{Q}[i,i]\})$. For example, if we apply uniformization to the CTMC in Figure 3.2(a) using $q = 0.84$, we obtain the DTMC in Figure 3.1(a).

From Equations (3.12) and (3.13) we obtain

$$\begin{aligned}
\boldsymbol{\pi}(t) &= \boldsymbol{\pi}(0) \cdot e^{(\mathbf{P}-\mathbf{I})qt} \\
&= \boldsymbol{\pi}(0) \cdot \sum_{n=0}^{\infty} \frac{(qt)^n}{n!}\mathbf{P}^n \cdot \sum_{n=0}^{\infty} \frac{(-qt)^n}{n!}\mathbf{I}^n \\
&= \boldsymbol{\pi}(0) \cdot \sum_{n=0}^{\infty} \frac{(qt)^n}{n!}\mathbf{P}^n \cdot \mathbf{I} \cdot e^{-qt} \\
&= \boldsymbol{\pi}(0) \cdot \sum_{n=0}^{\infty} \frac{(qt)^n}{n!}e^{-qt}\mathbf{P}^n.
\end{aligned}$$

Combining this result with Equation 3.4 gives us

$$\boldsymbol{\pi}(t) = \sum_{n=0}^{\infty} \frac{(qt)^n}{n!}e^{-qt}\mathbf{p}_n \tag{3.14}$$

where $\mathbf{p}_0 = \boldsymbol{\pi}(0)$ and the sequence $\mathbf{p}_n$ can be computed using Equation (3.3).

The probabilistic interpretation of Equation 3.14 is surprisingly intuitive. Of course, $\mathbf{p}_n$ describes the probability of being in each state of the DTMC at time n. This is multiplied by $\frac{(qt)^n}{n!} \cdot e^{-qt}$, which is the probability that a Poisson random variable with parameter qt has value n. Looking at Equation 3.2 we conclude that each term of the infinite sum is the probability distribution of the CTMC given that it has changed state exactly n times multiplied by the probability that the CTMC has changed state exactly n times. The infinite sum of Equation 3.14 is then just the law of total probability to compute the probability distribution of the CTMC.

From a practical standpoint, to compute $\boldsymbol{\pi}(t)$ we use the approximation

$$\boldsymbol{\pi}(t) \approx e^{-qt} \sum_{n=0}^{N} \frac{(qt)^n}{n!} \mathbf{p}_n \tag{3.15}$$

for some value of N [96]. Since

$$e^{-qt} \sum_{n=0}^{\infty} \frac{(qt)^n}{n!} = 1$$

we can choose ϵ and determine the value of N such that

$$e^{-qt} \sum_{n=0}^{N-1} \frac{(qt)^n}{n!} < 1 - \epsilon \leq e^{-qt} \sum_{n=0}^{N} \frac{(qt)^n}{n!}.$$

This truncation guarantees that our approximation of the probability vector has elements that sum to at least $1 - \epsilon$.

For example, looking at the CTMC in Figure 3.2(a), we can compute the probability of being in each state at time 6.0. Various values of N and the resulting computed value of $\boldsymbol{\pi}(6.0)$ from Equation 3.15 are shown in Table 3.1. In the table, ϵ is the value that must be added to elements of the computed vector $\boldsymbol{\pi}(6.0)$ to make it a probability vector. Note

N	$\boldsymbol{\pi}(6.0)$					ϵ
0	[0.002158	0.002158	0.000000	0.002158	0.000000]	9.935×10^{-1}
3	[0.002158	0.128649	0.032845	0.061906	0.033898]	7.405×10^{-1}
6	[0.002158	0.377077	0.120333	0.140269	0.116475]	2.437×10^{-1}
9	[0.002158	0.482270	0.160299	0.163888	0.158083]	3.330×10^{-2}
12	[0.002158	0.497841	0.166325	0.166539	0.164977]	2.160×10^{-3}
15	[0.002158	0.498883	0.166729	0.166680	0.165474]	7.556×10^{-5}
18	[0.002158	0.498920	0.166743	0.166684	0.165493]	1.571×10^{-6}
21	[0.002158	0.498921	0.166743	0.166684	0.165493]	2.089×10^{-8}

Table 3.1: Computing $\boldsymbol{\pi}(6.0)$ using uniformization

that it is possible for the CTMC to be in state A at time 6.0, even though the CTMC can only be in state A as the initial state. This is because it is possible for the CTMC to still be in the initial state at time 6.0. Also note that, unlike the similarly structured DTMC, the periodic structure of states $\{C, D, E\}$ hardly affects the CTMC: at time 6.0 the CTMC can be in any of those states with almost equal probability. This is because a CTMC can change state any number of times before time 6.0.

3.4.2 Stationary analysis

Another method of converting a CTMC to a DTMC is through an embedding [96]. Using this technique, we consider which state the CTMC switches to, given that it is changing state. We define a vector of expected holding times $\mathbf{h}$, where

$$\mathbf{h}[i] = \frac{-1}{\mathbf{Q}[i,i]}.$$

Then the probability matrix of the embedded DTMC is given by

$$\mathbf{P} = \mathbf{I} + Diag\,(\mathbf{h})\,\mathbf{Q} = Diag\,(\mathbf{h})\,\mathbf{R} \tag{3.16}$$

and the initial probability vector of the embedded DTMC is $\mathbf{p}_0 = \boldsymbol{\pi}(0)$.

From Equations (3.8) and (3.16) we can derive

$$\begin{aligned} \mathbf{p}(\mathbf{P} - \mathbf{I}) &= \mathbf{0} \\ \mathbf{p} \cdot (\mathbf{I} + Diag\,(\mathbf{h})\,\mathbf{Q} - \mathbf{I}) &= \mathbf{0} \\ (\mathbf{p}\,Diag\,(\mathbf{h}))\mathbf{Q} &= \mathbf{0} \end{aligned}$$

If the embedded DTMC is ergodic, then the stationary probability vector $\boldsymbol{\pi}$ of the original CTMC is the unique probability vector satisfying

$$\boldsymbol{\pi}\mathbf{Q} = \mathbf{0} \tag{3.17}$$

and $\boldsymbol{\pi}$ is related to the stationary probability vector $\mathbf{p}$ of the embedded DTMC by

$$\boldsymbol{\pi} = \frac{\mathbf{p}\,Diag\,(\mathbf{h})}{\mathbf{p} \cdot \mathbf{h}} \tag{3.18}$$

where the division by the dot product $\mathbf{p} \cdot \mathbf{h}$ normalizes $\boldsymbol{\pi}$ so that its elements sum to one.

For example, the CTMC in Figure 3.2(b) has generator matrix

$$\mathbf{Q} = \begin{bmatrix} -3 & 3 & 0 & 0 \\ 1 & -2 & 1 & 0 \\ 0 & 0 & -4 & 4 \\ 0 & 2 & 0 & -2 \end{bmatrix}$$

and so the holding times are

$$\mathbf{h} = \begin{bmatrix} \frac{1}{3} & \frac{1}{2} & \frac{1}{4} & \frac{1}{2} \end{bmatrix}$$

yielding the DTMC in Figure 3.1(b) as the embedded DTMC. Thus we can compute

$$\boldsymbol{\pi} = \frac{\begin{bmatrix} \frac{1}{5} & \frac{2}{5} & \frac{1}{5} & \frac{1}{5} \end{bmatrix} \begin{bmatrix} \frac{1}{3} & 0 & 0 & 0 \\ 0 & \frac{1}{2} & 0 & 0 \\ 0 & 0 & \frac{1}{4} & 0 \\ 0 & 0 & 0 & \frac{1}{2} \end{bmatrix}}{\begin{bmatrix} \frac{1}{5} & \frac{2}{5} & \frac{1}{5} & \frac{1}{5} \end{bmatrix} \cdot \begin{bmatrix} \frac{1}{3} & \frac{1}{2} & \frac{1}{4} & \frac{1}{2} \end{bmatrix}} = \frac{\begin{bmatrix} \frac{4}{60} & \frac{12}{60} & \frac{3}{60} & \frac{6}{60} \end{bmatrix}}{\frac{25}{60}} = \begin{bmatrix} \frac{4}{25} & \frac{12}{25} & \frac{3}{25} & \frac{6}{25} \end{bmatrix}$$

using Equation 3.18 and the stationary probability vector $\mathbf{p}$ computed previously. Alternatively, we could solve Equation 3.17 for $\boldsymbol{\pi}$ which would yield the same result.

3.4.3 Mean time to absorption

Given an absorbing CTMC, we can obtain an absorbing DTMC using an embedding. We can then compute the expected number of visits $\mathbf{n}$ to each transient state in the DTMC by solving the system

$$\mathbf{n}(\mathbf{I} - \mathbf{P}[\mathcal{Z}, \mathcal{Z}]) = \mathbf{p}_0[\mathcal{Z}] \tag{3.19}$$

where $\mathcal{Z}$ is the set of transient states. From Equations (3.19) and (3.16) we can derive

$$\begin{aligned} \mathbf{n}(\mathbf{I} - \mathbf{P}[\mathcal{Z}, \mathcal{Z}]) &= \mathbf{p}_0[\mathcal{Z}] \\ \mathbf{n} \cdot (\mathbf{I} - (\mathbf{I} + \mathit{Diag}\,(\mathbf{h}[\mathcal{Z}])\,\mathbf{Q}[\mathcal{Z}, \mathcal{Z}])) &= \mathbf{p}_0[\mathcal{Z}] \\ (\mathbf{n}\mathit{Diag}\,(\mathbf{h}[\mathcal{Z}]))\mathbf{Q}[\mathcal{Z}, \mathcal{Z}] &= -\boldsymbol{\pi}(0)[\mathcal{Z}] \end{aligned}$$

Then we can solve the system

$$\boldsymbol{\sigma}\mathbf{Q}[\mathcal{Z}, \mathcal{Z}] = -\boldsymbol{\pi}(0)[\mathcal{Z}]. \tag{3.20}$$

for $\boldsymbol{\sigma}$, where

$$\boldsymbol{\sigma} = \mathbf{n} Diag\left(\mathbf{h}[\mathcal{Z}]\right). \tag{3.21}$$

For each transient state i, $\boldsymbol{\sigma}[i]$ is the product of the expected number of visits to state i before absorption and the average time spent in state i per visit. Thus the elements of $\boldsymbol{\sigma}$ specify the expected amount of time spent in each transient state before absorption. The mean time to absorption is then the sum of the elements of $\boldsymbol{\sigma}$. The probability of being absorbed into each absorbing state is the same in the CTMC as in the embedded DTMC.

For example, the CTMC in Figure 3.2(c) has generator matrix

$$\mathbf{Q} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & -3 & 2 & 0 \\ 0 & 1 & 0 & 0 & -4 & 3 \\ 0 & 0 & 1 & 4 & 0 & -5 \end{bmatrix}$$

and so the holding times are

$$\mathbf{h} = \begin{bmatrix} \infty & \infty & \infty & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \end{bmatrix}.$$

Note that the holding times are infinite for the absorbing states: the average duration of a visit to an absorbing state is forever. The embedded DTMC is the one in Figure 3.1(c); thus we can compute

$$\boldsymbol{\sigma} = \begin{bmatrix} \frac{30}{18} & \frac{20}{18} & \frac{15}{18} \end{bmatrix} \begin{bmatrix} \frac{1}{3} & 0 & 0 \\ 0 & \frac{1}{4} & 0 \\ 0 & 0 & \frac{1}{5} \end{bmatrix} = \begin{bmatrix} \frac{10}{18} & \frac{5}{18} & \frac{3}{18} \end{bmatrix}$$

using Equation 3.21 and the vector $\mathbf{n}$ computed previously. Alternatively, we could solve Equation 3.20 for $\boldsymbol{\sigma}$ which would yield the same result. The probability of being absorbed

in each absorbing state is

$$\mathbf{a} = \begin{bmatrix} \frac{10}{18} & \frac{5}{18} & \frac{3}{18} \end{bmatrix}$$

which is the same as for the embedded DTMC and was computed earlier.

3.5 Phase-type distributions

With an absorbing Markov chain, we can look at the time required for the Markov chain to enter an absorbing state. This may depend on the initial state of the Markov chain. Given an absorbing Markov chain and an initial probability distribution, the amount of time required to reach an absorbing state is a random variable. Such a random variable is said to have a *phase-type* distribution [75]. A phase-type random variable based on a DTMC (CTMC) is discrete (continuous), and is called a discrete (continuous) phase-type.

3.5.1 Discrete phase-types

We say X is a discrete phase-type random variable if there exists a DTMC with absorbing final state f and an initial probability distribution for the DTMC such that the random variable describing the time to absorption of the DTMC into state f has the same distribution as X. Some example absorbing DTMCs and their associated discrete phase-type random variables are shown in Figure 3.3. The DTMCs shown have final state represented by state F, and initial probabilities as indicated below the state.

Let $\mathcal{D}$ be the set of all discrete phase-type random variables. It can be shown that the set $\mathcal{D}$ is closed under the following.

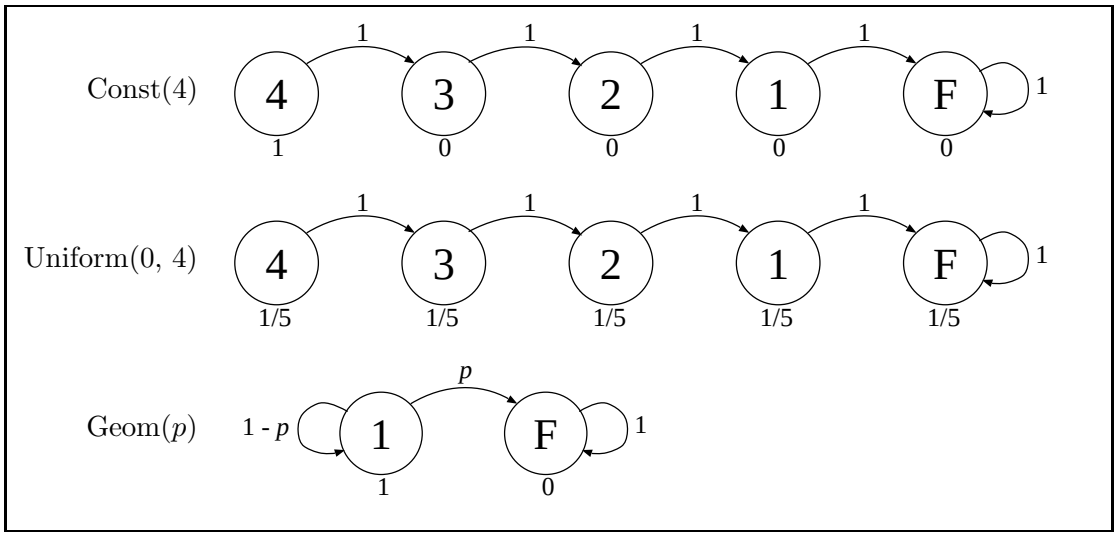


Figure 3.3: Discrete phase distributions

Convolution:

$$X_1, \ldots, X_n \in \mathcal{D} \;\Rightarrow\; \sum_{i=1}^{n} X_i \;\in \mathcal{D}$$

Probabilistic Choice:

$$X_1, \ldots, X_n \in \mathcal{D} \;\Rightarrow\; Z = \begin{cases} X_1 & \text{with probability } \boldsymbol{\alpha}[1] \\ \vdots & \\ X_n & \text{with probability } \boldsymbol{\alpha}[n] \end{cases} \in \mathcal{D}$$

where $\boldsymbol{\alpha}$ is a probability vector.

Random convolution over a discrete phase-type:

$$X, Y \in \mathcal{D} \;\Rightarrow\; \sum_{i=1}^{X} Y_i \in \mathcal{D}$$

where $Y_1, \ldots, Y_X$ are independent random variables with the same distribution as Y. Note that if Y is a constant, the sum reduces to XY; thus the product of a natural number and a discrete phase-type random variable is also a discrete phase-type random variable.

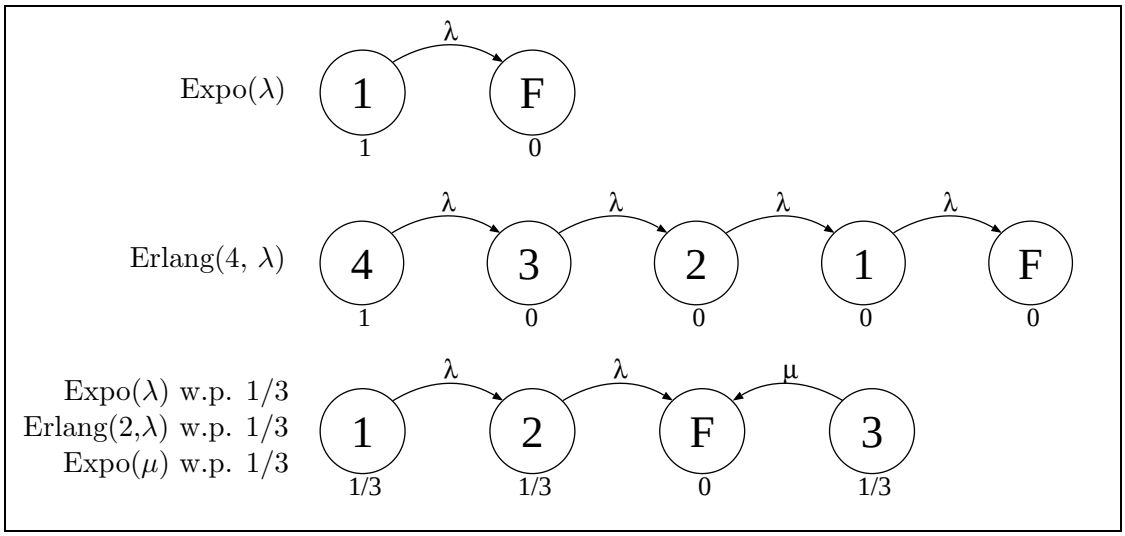


Figure 3.4: Continuous phase distributions

Finite order statistics:

$$\mathcal{X} \subset \mathcal{D}, |\mathcal{X}| < \infty, k \in \{0, \ldots, |\mathcal{X}| - 1\} \quad \Rightarrow \quad \text{ord}_k(\mathcal{X}) \in \mathcal{D},$$

where $\text{ord}_k(\mathcal{X})$ is the k^{th} largest element in set $\mathcal{X}$.

3.5.2 Continuous phase-types

We say X is a continuous phase-type random variable if there exists a CTMC with absorbing final state f and an initial probability distribution for the CTMC such that the random variable describing the time to absorption of the CTMC into state f has the same distribution as X. Some example absorbing CTMCs and their associated continuous phase-type random variables are shown in Figure 3.3. The CTMCs shown have final state represented by state F, and initial probabilities as indicated below the state.

Let $\mathcal{C}$ be the set of all continuous phase-type random variables. It can be shown that

the set $\mathcal{C}$ is closed under the following.

Convolution:

$$X_1, \ldots, X_n \in \mathcal{C} \quad \Rightarrow \quad \sum_{i=1}^{n} X_i \ \in \mathcal{C}$$

Probabilistic Choice:

$$X_1, \ldots, X_n \in \mathcal{C} \quad \Rightarrow \quad Z = \begin{cases} X_1 & \text{with probability } \boldsymbol{\alpha}[1] \\ \quad \vdots & \\ X_n & \text{with probability } \boldsymbol{\alpha}[n] \end{cases} \in \mathcal{C}$$

where $\boldsymbol{\alpha}$ is a probability vector.

Random convolution over a discrete phase-type:

$$X \in \mathcal{D}, Y \in \mathcal{C} \quad \Rightarrow \quad \sum_{i=1}^{X} Y_i \in \mathcal{C}$$

where $Y_1, \ldots, Y_X$ are independent random variables with the same distribution as Y.

Scaling:

$$a \in I\!R_*, \ X \in \mathcal{C} \quad \Rightarrow \quad aX \in \mathcal{C}$$

Finite order statistics:

$$\mathcal{X} \subset \mathcal{C}, |\mathcal{X}| < \infty, k \in \{0, \ldots, |\mathcal{X} - 1\} \quad \Rightarrow \quad \mathrm{ord}_k(\mathcal{X}) \in \mathcal{C}$$

where $\mathrm{ord}_k(\mathcal{X})$ is the k^{th} largest value in set $\mathcal{X}$.

3.5.3 Phase-types and general distributions

Note that any discrete distribution with finite, non-negative integer support has a discrete phase-type representation: if $\mathcal{N}$ is the set of integers which occur with positive probability,

and p is the probability mass function, then the distribution is equivalent to the probabilistic choice

$$\forall n \in \mathcal{N}, \quad \text{Const}(n) \text{ with probability } p(n)$$

which is clearly a discrete phase-type random variable. We can also approximate any infinite, non-negative, discrete distribution within an error of ϵ by truncation. That is, we use $\mathcal{N} = \{n : p(n) \geq \epsilon\}$, which guarantees that

$$\forall n \in \mathbb{N}, |p(n) - p'(n)| < \epsilon$$

where $p'(n)$ is the probability mass function for the discrete phase-type distribution. This may be difficult in practice, since a small value of ϵ may produce a very large DTMC for the phase-type distribution.

The set of continuous phase-type distributions can be shown to be dense on the set of all probability distributions on the set of positive reals $\mathbb{R}_+$ [75]. This implies that, given an arbitrary CDF F, we can find a continuous phase-type distribution with CDF G that approximates F within ϵ [35]:

$$\forall x \in \mathbb{R}_+, |F(x) - G(x)| < \epsilon.$$

The above property, while interesting, is of little practical use. The number of CTMC states required to approximate an arbitrary distribution accurately could be astronomical. For instance, it has been shown [5] that the best continuous phase-type approximation to Const(δ), where the CTMC contains exactly n states has distribution Erlang(n, $\frac{\delta}{n}$), which has variance $\frac{\delta^2}{n}$. Thus, a continuous phase-type approximation to a constant distribution with variance less than ϵ requires a CTMC with $O(\frac{1}{\epsilon})$ states.

Chapter 4

High-level formalisms

This chapter shows how high-level models of systems can be analyzed. Our work applies to any model described using any type of high-level formalism that meets the requirements discussed in Section 4.1. In particular, we consider models with finite state spaces only. Section 4.2 gives an overview of Petri nets, a formalism that meets our requirements. For more information, we refer the reader to [2, 22, 74, 82] for theoretical aspects of Petri nets, to [71, 72] for stochastic Petri nets, and to [4, 8, 21] for generalized stochastic Petri nets. Section 4.3 gives necessary definitions for studying the possible states of the model. Section 4.4 shows how the underlying CTMC can be generated for our model. Once we have obtained the CTMC, we can analyze the CTMC to obtain performance measures of interest about the system. Finally, Section 4.5 describes classes of structured models, in which a model is a composition of several smaller models.

4.1 Model paradigm

While Markov chains are useful tools, specifying a large Markov chain by hand is impractical. Instead, we use high-level formalisms to describe a model compactly. From a high-level

formalism we can then automatically generate and analyze the underlying Markov chain. Once the stationary or transient probability vector (or the time spent in each state) is obtained, we can compute some measure(s) of interest about the model.

Instead of focusing on one particular formalism, we assume a general model "interface" containing necessary abstract entities. In this way, our results apply to a wide class of formalisms. We assume that our model either directly specifies or allows for trivial computation of the following:

- A finite set of events $\mathcal{E}$.
- A finite set of *potential* or possible states $\hat{\mathcal{S}}$.
- An initial state $s^\circ \in \hat{\mathcal{S}}$.
- An indexing function for potential states, given by the bijection $\hat{\Psi} : \hat{\mathcal{S}} \to \{0, \ldots, |\hat{\mathcal{S}}|\}$. Usually, the set $\hat{\mathcal{S}}$ has a very regular structure, and so computing $\hat{\Psi}$ and $\hat{\Psi}^{-1}$ are trivial operations.
- For each event $e \in \mathcal{E}$, a function $Next^e : \hat{\mathcal{S}} \to 2^{\hat{\mathcal{S}}}$, where

$$Next^e(s) = \{s' : s \xrightarrow{e} s'\}$$

  and the notation $s \xrightarrow{e} s'$ means that from state s we can reach state s' if event e occurs. Note that $Next^e(s)$ can be the empty set; this occurs when event e is *disabled* in state s. Also, note that we allow $Next^e(s)$ to contain more than one element; this implies that if an event occurs in a given state, there may be some uncertainty as to which new state is reached. Such a definition is useful in practice, but not necessary; we could represent the same behavior by introducing additional events.

- For each event $e \in \mathcal{E}$, a probability function $Prob^e : \hat{\mathcal{S}} \times \hat{\mathcal{S}} \rightarrow \mathbb{R}_*$, where $Prob^e(s, s')$ is the probability that event e causes the system to switch from state s to state s'. Note that $Prob^e(s, s')$ is zero if and only if $s' \notin Next^e(s)$. Of course, we assume that

$$\forall s \in \hat{\mathcal{S}} \text{ such that } Next^e(s) \neq \emptyset, \quad \sum_{s' \in Next^e(s)} Prob^e(s, s') \ = 1,$$

  which says that $Prob^e$ is a "proper" probability function.

- For each event $e \in \mathcal{E}$, a rate function $Rate^e : \hat{\mathcal{S}} \rightarrow \mathbb{R}$, where $Rate^e(s)$ is the rate of occurrence of event e when the system is in state s. In other words, we assume that the time for event e to occur in state s is a random variable with distribution Expo($Rate^e(s)$).

- Function $Reward : \hat{\mathcal{S}} \rightarrow \mathbb{R}$, which will be used to compute our measure of interest.

With this class of models, we assume that the time for every event to occur is exponentially distributed. As we shall see, this allows us to easily generate the underlying stochastic process of the model, which is a CTMC. In general, we could allow the time for events to occur to be continuous phase-type random variables and still obtain an underlying CTMC (with many more states). Also, discrete phase-type random variables could be used for the duration of events, which would produce an underlying DTMC. Such extensions, while conceptually straightforward, raise interesting issues (especially with respect to efficient implementation) that are beyond the scope of our work; see for instance [3, 23, 24, 36, 48, 91].

4.2 Petri nets

An example of a modeling formalism that fits our paradigm is the class of Petri net models [74, 82]. Our Petri net definition is quite general, allowing inhibitor arcs [2] and marking-dependent arc cardinalities [22]. A Petri net is a directed, bipartite graph consisting of the following.

- A finite set of places $\mathcal{P} = \{p_0, \ldots, p_{|\mathcal{P}|-1}\}$. Places are usually drawn as circles. Each place contains a non-negative number of tokens, depicted either as filled circles or as a written number. A marking $\mathbf{m} \in \mathbb{N}^{|\mathcal{P}|}$ completely describes the state of the Petri net, where $\mathbf{m}[p]$ is the number of tokens in place $p \in \mathcal{P}$.

- An initial marking $\mathbf{m}^\circ$.

- A finite set of transitions $\mathcal{T} = \{t_0, \ldots, t_{|\mathcal{T}|-1}\}$ which are usually depicted as bars or boxes. The sets $\mathcal{P}$ and $\mathcal{T}$ are disjoint.

- For each place-transition pair (p, t), an input-arc cardinality function $I_p^t : \mathbb{N}^{|\mathcal{P}|} \to \mathbb{N}$, an output-arc cardinality function $O_p^t : \mathbb{N}^{|\mathcal{P}|} \to \mathbb{N}$, and an inhibitor-arc cardinality function $H_p^t : \mathbb{N}^{|\mathcal{P}|} \to \mathbb{N} \cup \{\infty\}$. If I_p^t is not identically zero, we say p is an *input place* of transition t, and we write $p \to t$. Similarly, if O_p^t is not identically zero, we say p is an *output place* of transition t, and we write $t \to p$. If H_p^t is not identically infinity (i.e., for some marking $\mathbf{m}$, $H_p^t(m)$ is finite), we say p is an *inhibitor place* of transition t, and we write $p \multimap t$. Given a transition t, we denote its input places as $\bullet t$, its output places as $t\bullet$, and its inhibitor places as $\circ t$.

We say a transition t is *enabled* in marking $\mathbf{m}$ if

$$\forall p \in \mathcal{P}, \quad I_p^t(\mathbf{m}) \leq \mathbf{m}[p] < H_p^t(\mathbf{m}). \tag{4.1}$$

Such an enabled transition may *fire*, leading to a new marking $\mathbf{m'}$ given by

$$\mathbf{m'}[p] = \mathbf{m}[p] - I_p^t(\mathbf{m}) + O_p^t(\mathbf{m}). \tag{4.2}$$

Thus, the firing of a transition destroys tokens from the input places and creates tokens in the output places. Note that tokens do not necessarily need to be "conserved" in a Petri net: a firing transition may create fewer or more tokens than it destroys.

A place p is said to be B-bounded if the number of tokens in place p cannot exceed B, given the initial marking $\mathbf{m}^\circ$. A Petri net is said to be B-bounded if all its places are B-bounded. A 1-bounded net is said to be *safe*.

Petri nets are usually depicted graphically, as seen in the examples in Figure 4.1 and Figure 4.2. For each transition t, an input arc is drawn from each input place $p \in \bullet t$, an output arc is drawn to each output place $p \in t\bullet$, and an inhibitor arc is drawn from each inhibitor place $p \in \circ t$. Inhibitor arcs are distinguished from input arcs by a small circle at the end of the arc instead of an arrowhead. Arcs are labeled with their cardinalities, unless the cardinality is identically one. Zero-cardinality input and output arcs and infinite-cardinality inhibitor arcs are omitted. Note that inhibitor arcs with zero cardinality will permanently disable a transition; we assume that these are omitted since the desired behavior can be represented by eliminating the transition completely.

The Petri net in Figure 4.1 represents an open queueing network with 4 queues (places q_1, q_2, q_3, q_4). The Petri net is not bounded regardless of the initial marking, since transition

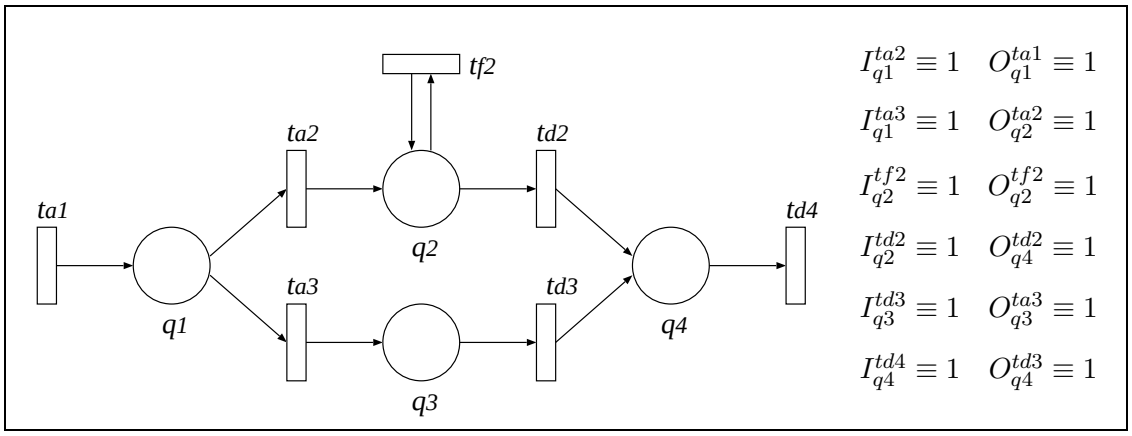


Figure 4.1: Petri net of an open queueing network

t_{a1} (representing an arrival to queue 1) may fire an arbitrary number of times before transition t_{a2} or t_{a3} fires. The Petri net in Figure 4.2 represents a simple producer-consumer system with buffer capacity $B > 0$. In this simple model, the producer continuously produces data that must be processed in a large batch; this might correspond to a block of data written to disk or a packet to be sent across a network. Note that this net is B-bounded: transition *producer* is disabled when place *buffer* contains B tokens.

Our above definition of Petri nets can almost be applied to our modeling paradigm:

- The set of events $\mathcal{E}$ is given by the set of transitions $\mathcal{T}$.
- A set of potential states $\hat{\mathcal{S}}$ of a bounded net can be given by

$$\hat{\mathcal{S}} = \{0, \ldots, B_{|\mathcal{P}|-1}\} \times \cdots \times \{0, \ldots, B_0\} \subseteq \{0, \ldots, B\}^{|\mathcal{P}|}$$

where B_i is the bound on the number of tokens in place p_i. We cannot consider unbounded nets, since they do not have a finite set of potential states.

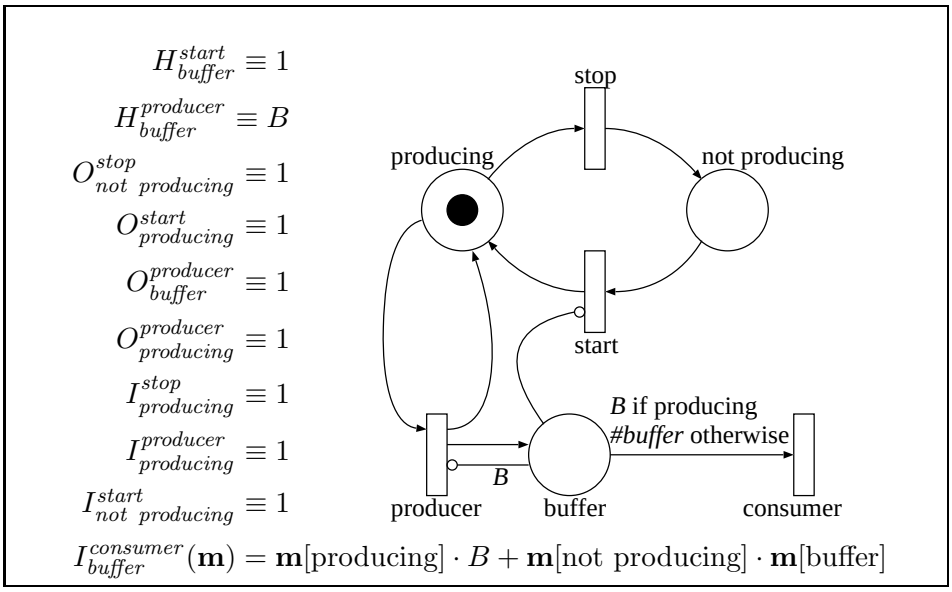


Figure 4.2: Producer / Consumer Petri net

- One possible indexing function $\hat{\Psi}$ is based on the notion of a *mixed-radix* number representation system [37, 59]. Given an integer-valued *basis* vector

$$\mathbf{b} = [b_K, \ldots, b_1], \qquad b_i \geq 2,$$

any integer $0 \leq a < \prod_{k=1}^{K} b_k$ has a unique representation $\mathbf{a} = [a_K, \ldots, a_1]$ satisfying:

1. $a = a_1 + \sum_{i=2}^{K} a_i \prod_{k=1}^{i-1} b_k$
2. $0 \leq a_i < b_i, \quad \forall i \in \{1, \ldots, K\}$.

We say $\mathbf{a}$ is the mixed-radix representation of a with respect to the basis $\mathbf{b}$. If we consider a marking $\mathbf{m}$ as a mixed-radix integer, then we can define the indexing function $\hat{\Psi}$ such that $\mathbf{m}$ is the mixed-radix representation of $\hat{\Psi}(\mathbf{m})$ with respect to the basis $\mathbf{b} = [B_{|\mathcal{P}|-1} + 1, \ldots, B_0 + 1]$.

- The initial state s° corresponds to the initial marking $\mathbf{m}^\circ$.
- Function $Next^t$ corresponds to Equation 4.2. Of course, $Next^t$ returns the empty set when transition t is disabled.
- Reward functions can be defined in terms of the number of tokens in certain places.

The only part of our modeling paradigm that cannot be captured by Petri nets are the functions *Rate* and *Prob*. Thus, we use an extension of Petri nets called *stochastic* Petri nets (SPNs) [71, 72], where the firing of a transition is associated with an exponentially-distributed random delay. We assume that each transition t requires a time to fire given by a random variable with distribution $Expo(\lambda_t)$, where λ_t may depend on the current marking. This corresponds exactly to the rate function, and so $Rate^t(\mathbf{m}) = \lambda_t(\mathbf{m})$ in our case. For SPNs, the probability function $Prob^t$ is one, since only one marking can be reached when a transition fires.

We can also handle *generalized* SPNs (GSPNs) [4], where some transitions are allowed to fire in zero time. The firing of a timed transition may lead to a *vanishing* marking, in which immediate transitions are enabled. If multiple immediate transitions are enabled, they are assigned relative probabilities of firing. Immediate transitions will continue to fire until a *tangible* marking is reached, in which only timed transitions are enabled. In this case, we eliminate the vanishing markings [8, 21] and consider only the tangible markings as states and the timed transitions as events; thus the firing of a timed transition in a given tangible marking may lead to more than one tangible marking. The *Prob* function is then the probability of reaching each of the tangible markings, and can be computed from the relative weights for the immediate transitions.

4.3 Logical analysis

Once we have formalized our model, we can perform various types of analysis. From now on, we describe model analysis in terms of the model paradigm, with the understanding that we can trivially transform our model from our formalism of choice to the paradigm.

We may be interested only in logical aspects of the model. In that case, we can ignore the *Rate* and *Prob* functions for each event and focus on which states can be reached. For instance, we may want to know if it is possible to reach a state that meets a set of criteria, or if it is possible to reach a state meeting condition B after reaching a state meeting condition A. For now, we will only consider a special case of logical analysis: namely, determining all states that can be reached from the initial state. Some aspects of other kinds of logical analysis will be briefly discussed later. A thorough discussion of various types of logical analysis is beyond the scope of our work; see for instance [17, 32, 61, 65].

Given a model described using our paradigm, we can compute the following:

- A set of *actual* or reachable states $\mathcal{S} \subseteq \hat{\mathcal{S}}$, defined as the smallest set containing s° such that

$$\forall s \in \mathcal{S}, \forall e \in \mathcal{E} : s' \in Next^e(s) \Rightarrow s' \in \mathcal{S}.$$

  We say state s' is *zero-step reachable* from state s, written $s \overset{0}{\rightsquigarrow} s'$, if $s = s'$. We say state s' is *n-step reachable* from state s, written $s \overset{n}{\rightsquigarrow} s'$, if

$$\exists t \in \hat{\mathcal{S}}, e \in \mathcal{E} \text{ such that } s \xrightarrow{e} t \wedge t \overset{n-1}{\rightsquigarrow} s'.$$

  Note that s' may be both n- and n'-step reachable from s. In particular, if $s \overset{n}{\rightsquigarrow} s'$ and $s' \overset{m}{\rightsquigarrow} s$, then $s \overset{2n+m}{\rightsquigarrow} s'$.

We say state s' is reachable from state s, written $s \rightsquigarrow s'$, if s' is n-step reachable from s for some $n \in \mathbb{N}$. In other words, s' is reachable from state s if there is a (possibly empty) sequence of events whose occurrence can change the state of the system from state s to state s'. The set $\mathcal{S}$ can then also be defined as the subset of $\hat{\mathcal{S}}$ such that

$$s \in \mathcal{S} \;\Leftrightarrow\; s^{\circ} \rightsquigarrow s.$$

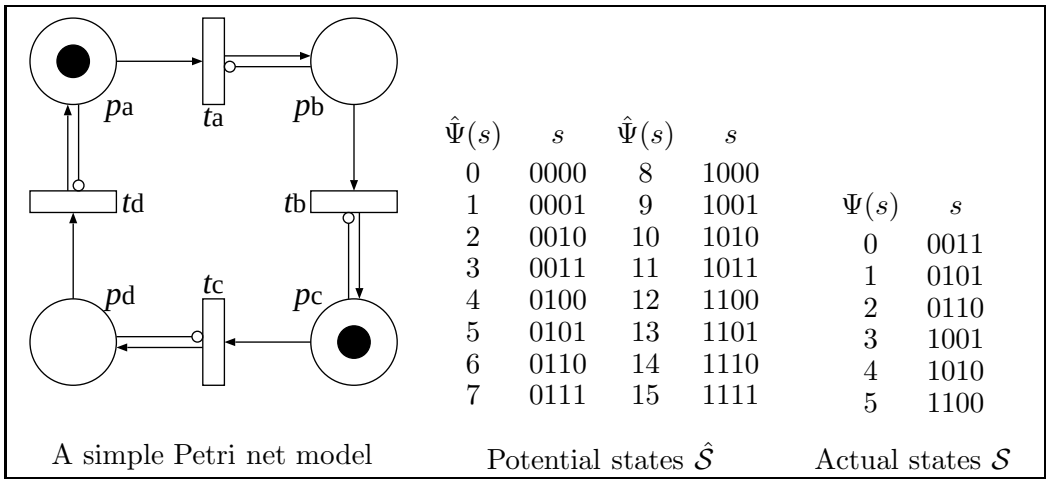
This definition is a bit more practical in the sense that the set $\mathcal{S}$ must typically be generated by examining all states reachable from the initial state. Thus, determining $\mathcal{S}$ is a non-trivial operation in practice.

- An indexing function for actual states, given by the bijection $\Psi : \mathcal{S} \rightarrow \{0, \ldots, |\mathcal{S}|\}$. For most sets $\mathcal{S}$, computation of Ψ and Ψ^{-1} are non-trivial operations that require storing $\mathcal{S}$.

An example showing a simple Petri net model and its potential and actual states ($\hat{\mathcal{S}}$ and $\mathcal{S}$) is provided in Figure 4.3. Since the Petri net is safe and contains exactly 4 places, we use $\hat{\mathcal{S}} = \{0, 1\}^4$ which gives exactly 16 potential states. Since each state is also a 4-bit binary number, we use that number (shown in decimal) for the potential index $\hat{\Psi}$. Starting from an initial state of $[1, 0, 1, 0]$ as shown in the figure, we discover that only 6 of those states are reachable.

4.4 Markov analysis

In cases where we are interested in performance or reliability, the events of our model occur with certain rates. Then our focus is usually more towards the probability of being in certain



A simple Petri net model

$\hat{\Psi}(s)$	s	$\hat{\Psi}(s)$	s
0	0000	8	1000
1	0001	9	1001
2	0010	10	1010
3	0011	11	1011
4	0100	12	1100
5	0101	13	1101
6	0110	14	1110
7	0111	15	1111

Potential states $\hat{\mathcal{S}}$

$\Psi(s)$	s
0	0011
1	0101
2	0110
3	1001
4	1010
5	1100

Actual states $\mathcal{S}$

Figure 4.3: A simple model and its state space

states, or the probability of reaching a state that meets condition A before reaching a state that meets condition B. To perform these types of analysis, we must study the underlying process of the model. For our model paradigm, the underlying process is a CTMC. Thus, we use our model to generate the underlying CTMC, and then analyze the underlying CTMC.

For any underlying CTMC, we can compute transient probabilities. If the underlying CTMC is irreducible, then we can also compute stationary probabilities. If the underlying CTMC is absorbing, then we can also compute the mean time to absorption or the probability of absorption in each absorbing state. Transient and stationary analysis will yield a probability for each state, and cumulative analysis will yield the expected time spent in each transient state or the probability of being absorbed in each absorbing state. The reward contribution from a state is then the product of the reward rate for a given state (given by the *Reward* function) and the probability or time for each state. Summing the reward contribution over all reachable states yields the desired result: the expected value of the reward function.

In any event, to compute our quantity of interest we must be able to specify the underlying CTMC by defining the transition rate matrix. When the Markov chain is generated from the model, there must be a mapping from states of the model to states of the Markov chain. One possibility is to use $\hat{\Psi}$ as the mapping. In this case, the state space of the Markov chain is equivalent to the set of potential states $\hat{\mathcal{S}}$ of the model. Another possibility is to use Ψ as the mapping, and so the state space of the Markov chain is equivalent to the set of actual states $\mathcal{S}$ of the model.

If we decide to use $\hat{\Psi}$ as the mapping, our CTMC will have $|\hat{\mathcal{S}}|$ states; i.e., each state in the CTMC will correspond to a state in $\hat{\mathcal{S}}$. The matrix $\hat{\mathbf{R}}$ given by

$$\forall s, s' \in \hat{\mathcal{S}}, \quad \hat{\mathbf{R}}[\hat{\Psi}(s), \hat{\Psi}(s')] = \sum_{e \in \mathcal{E}} \hat{\mathbf{R}}^e[\hat{\Psi}(s), \hat{\Psi}(s')] = \sum_{e \in \mathcal{E}} Prob^e(s, s') \cdot Rate^e(s) \tag{4.3}$$

is the transition rate matrix of the underlying CTMC, where $\hat{\mathbf{R}}^e$ is the contribution due to event e. Simply put, Equation 4.3 says that the rate of transition from state s to state s' in the CTMC is the sum over all events e of the rate of transition from state s to state s' via the occurrence of event e. By rearranging the rows and columns of $\hat{\mathbf{R}}$, we see that

$$\hat{\mathbf{R}} = \begin{bmatrix} \mathbf{R} & \mathbf{0} \\ \mathbf{X} & \mathbf{Y} \end{bmatrix} \tag{4.4}$$

where the block $\mathbf{R} = \hat{\mathbf{R}}[\mathcal{S}, \mathcal{S}]$ corresponds to the transitions from reachable states to reachable states. The $\mathbf{0}$ block corresponds to transitions from reachable states to unreachable states, which cannot occur by definition. The $\mathbf{X}$ block corresponds to transitions from unreachable states to reachable states, and it can be shown by example that this block is not guaranteed to be $\mathbf{0}$. This is because an unreachable state can lead to a reachable state,

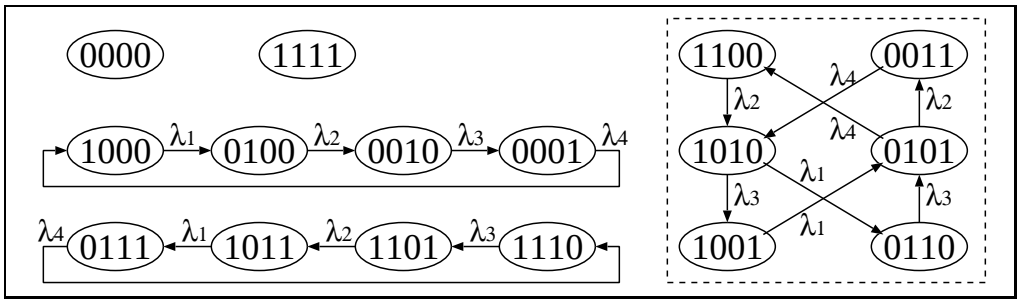


Figure 4.4: Underlying CTMC based on $\hat{\mathcal{S}}$

even though a reachable state can never lead to an unreachable state. Finally, the **Y** block corresponds to transitions from unreachable states to unreachable states.

If we instead use Ψ as the mapping, then our underlying CTMC will have $|\mathcal{S}|$ states, and each state in the CTMC will correspond to a state in $\mathcal{S}$. The transition rate matrix of the underlying CTMC is then exactly **R** from Equation 4.4. However, using Ψ as a mapping introduces some overheads. In general, the function Ψ cannot be computed without first generating (and storing) the set $\mathcal{S}$. This can require substantial storage space and computation. For most cases, $|\hat{\mathcal{S}}| \gg |\mathcal{S}|$, so the amount of memory and CPU required to generate $\mathcal{S}$ is less than the amount saved by using Ψ instead of $\hat{\Psi}$.

An example that shows the CTMC obtained from a simple model using mappings $\hat{\Psi}$ and Ψ is shown in Figure 4.4. The model used is the Petri net in Figure 4.3. It is assumed that each transition t_i in the Petri net fires with constant rate λ_i when enabled. The CTMC based on $\hat{\Psi}$ has $|\hat{\mathcal{S}}| = 16$ states, each labeled with the appropriate marking. The CTMC based on Ψ, enclosed by the dashed rectangle, has $|\mathcal{S}| = 6$ states. Note that in this example, there are no transitions from unreachable states to reachable states, so block **X** in Equation 4.4 is **0**.

State space	$\hat{\mathcal{S}}$	$\mathcal{S}$
number of states	16	6
storage requirement	0 bytes	6 bytes
Transition rate matrix	$\hat{\mathbf{R}}$	$\mathbf{R}$
number of entries	16	8
(sparse row-wise storage with arrays)		
Row pointers	17 bytes	7 bytes
Column indices	16 bytes	8 bytes
Single-precision entries	64 bytes	32 bytes
Probability vector	$\hat{\boldsymbol{\pi}}$	$\boldsymbol{\pi}$
Double-precision entries	128 bytes	48 bytes
Total	225 bytes	101 bytes

Table 4.1: Comparison of memory required for $\hat{\Psi}$ vs. Ψ

A comparison of the memory requirements for using an underlying CTMC based on $\hat{\mathcal{S}}$ versus $\mathcal{S}$ is shown in Table 4.1. We assume that the set of reachable states $\mathcal{S}$ is stored using an array of potential state indices; since there are fewer than 256 potential states we can use an array of one-byte integers. To further conserve memory we could store two indices per byte (since there are only 16 potential states). The transition rate matrices are stored in a sparse, row-wise structure using arrays. Again, one-byte integers are used for the row pointers and column indices. We assume that a four-byte, single-precision floating-point representation is used for the real numbers in the matrix. Finally, the probability vectors are stored in full using eight-byte, double-precision floating-point representation for each probability.

Continuing our analysis of the Petri net model in Figure 4.3, we can obtain the transition rate matrix and the infinitesimal generator matrix of the underlying CTMC based on the

set of actual states

$$\mathbf{R} = \begin{bmatrix} 0 & 0 & 0 & 0 & 4 & 0 \\ 2 & 0 & 0 & 0 & 0 & 4 \\ 0 & 3 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 \end{bmatrix} \quad \mathbf{Q} = \begin{bmatrix} -4 & 0 & 0 & 0 & 4 & 0 \\ 2 & -6 & 0 & 0 & 0 & 4 \\ 0 & 3 & -3 & 0 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 3 & -4 & 0 \\ 0 & 0 & 0 & 0 & 2 & -2 \end{bmatrix}$$

where we use firing rates of $\lambda_i = i$ and we order the states according to Figure 4.3. We can then compute the stationary probability vector

$$\boldsymbol{\pi} = \begin{bmatrix} \frac{1}{20} & \frac{2}{20} & \frac{1}{20} & \frac{9}{20} & \frac{3}{20} & \frac{4}{20} \end{bmatrix}$$

using an iterative linear solver such as Gauss-Seidel. Once we have a probability vector, we can compute measures of interest. For instance, to compute the expected number of tokens in each place p, we can take the sum over all reachable states of the product of the probability of that state and the number of tokens in place p in that state.

$$\begin{aligned} \text{E(tokens in } Pa) &= 0 \cdot \tfrac{1}{20} + 0 \cdot \tfrac{2}{20} + 0 \cdot \tfrac{1}{20} + 1 \cdot \tfrac{9}{20} + 1 \cdot \tfrac{3}{20} + 1 \cdot \tfrac{4}{20} = \tfrac{16}{20} \\ \text{E(tokens in } Pb) &= 0 \cdot \tfrac{1}{20} + 1 \cdot \tfrac{2}{20} + 1 \cdot \tfrac{1}{20} + 0 \cdot \tfrac{9}{20} + 0 \cdot \tfrac{3}{20} + 1 \cdot \tfrac{4}{20} = \tfrac{7}{20} \\ \text{E(tokens in } Pc) &= 1 \cdot \tfrac{1}{20} + 0 \cdot \tfrac{2}{20} + 1 \cdot \tfrac{1}{20} + 0 \cdot \tfrac{9}{20} + 1 \cdot \tfrac{3}{20} + 0 \cdot \tfrac{4}{20} = \tfrac{5}{20} \\ \text{E(tokens in } Pd) &= 1 \cdot \tfrac{1}{20} + 1 \cdot \tfrac{2}{20} + 0 \cdot \tfrac{1}{20} + 1 \cdot \tfrac{9}{20} + 0 \cdot \tfrac{3}{20} + 0 \cdot \tfrac{4}{20} = \tfrac{12}{20} \end{aligned}$$

Once the probability vector has been computed, determining each measure requires $O(|\mathcal{S}|)$ time, assuming the probability vector is stored in full.

The steps taken to analyze the model in the previous example are the same for complex models. Thus, for Markov analysis of a model we perform the following steps.

1. Generate the set of reachable states $\mathcal{S}$.

2. Build the transition rate matrix $\mathbf{R}$ of the underlying CTMC.

3. Compute the desired probability vector (or average time spent in each state).

4. Compute measures of interest.

Note that the first step, generation of $\mathcal{S}$, is also useful for logical analysis. Unfortunately, practical models can easily have an *extremely* large number of reachable states. In those cases, the straightforward approaches used for each of the steps in the previous example can lead to storage requirements well beyond the capabilities of modern workstations. As the number of reachable states grows, we must turn to sophisticated techniques for generating and storing $\mathcal{S}, \mathbf{R}$, and $\boldsymbol{\pi}$.

4.5 Structured models

Sometimes, it makes sense to compose a model from several components, where each component is also a model. We say each component is a *submodel*, even though the submodels may be defined as a complete model. This hierarchical approach is often a very natural way to model a system. As we shall see in later chapters, it is also possible to exploit properties of the submodels and their interactions for more efficient analysis of the model. Thus, often times a model is *decomposed* into submodels so that these efficient algorithms can be applied.

We say a model is *structured* if it is composed of $K > 1$ submodels. We denote the components of the k^{th} submodel by adding the subscript k; for instance, $\mathcal{E}_k$ denotes the events for submodel k, and $\hat{\mathcal{S}}_k$ denotes the set of potential states for submodel k. We assume that a structured model has the following properties.

- The set of events $\mathcal{E}$ of the model is given by

$$\mathcal{E} = \mathcal{E}_K \cup \cdots \cup \mathcal{E}_1$$

where $\mathcal{E}_k$ is the set of events in submodel k. We say an event $e \in \mathcal{E}$ *affects* submodel k if $e \in \mathcal{E}_k$. We define functions $\mathit{first} : \mathcal{E} \to \{1, \ldots, K\}$ and $\mathit{last} : \mathcal{E} \to \{1, \ldots, K\}$ as the first and last submodels that an event affects. More formally, we have

$$\begin{aligned} \mathit{first}(e) &= \max\left(\{k : e \text{ affects submodel } k\}\right) \\ \mathit{last}(e) &= \min\left(\{k : e \text{ affects submodel } k\}\right) \end{aligned}$$

which is potentially confusing because $\mathit{first}(e) \geq \mathit{last}(e)$.

An event is called *synchronizing* if it affects more than one submodel (i.e., $\mathit{first}(e)$ is strictly greater than $\mathit{last}(e)$). Note that the interaction between submodels occurs entirely through the synchronizing events. An event that affects only one submodel k is said to be a *local* event of submodel k.

- A state of the model can be expressed as $(s_K, \ldots, s_1)$, where s_k is a state of submodel k. Thus, the set of potential states is given by

$$\hat{\mathcal{S}} = \mathcal{S}_K \times \cdots \times \mathcal{S}_1$$

where $\mathcal{S}_k$, the set of actual states for submodel k, is called the set of *local* states for submodel k. Alternatively, we could use potential states based on the potential states for each submodel

$$\hat{\mathcal{S}} = \hat{\mathcal{S}}_K \times \cdots \times \hat{\mathcal{S}}_1$$

if the local states of each submodel are unknown.

- The initial state of the model is

$$s^\circ = (s_K^\circ, \ldots, s_1^\circ)$$

where s_k° is the initial state of submodel k.

Note that we do not distinguish between large models that are decomposed into submodels and the composition of small models into a structured model. In fact, we do not even assume that the submodels are described with the same formalism; although in the case of decomposition this is usually true.

An example of a structured model is shown in Figure 4.5. In the example, we have a structured model composed of four submodels, each equivalent to the Petri net of Figure 4.3. Of course, the structured model is also a Petri net. A state of submodel 1 is determined by the number of tokens in places Pa_1, Pb_1, Pc_1 and Pd_1. States for submodels $2, 3$ and 4 are similarly determined. A state of the model is given by (s_1, s_2, s_3, s_4). The initial state of the model is then

$$([1,0,1,0],[1,0,1,0],[1,0,1,0],[1,0,1,0])$$

as shown in the figure. The synchronizing events of the model are the shaded transitions. We see for instance that events Tb_1 and Td_2 have been merged into a single synchronizing event $Tsynch_{12}$. Non-shaded transitions represent local events. For instance, transition Ta_1 is an event local to submodel 1: it is not affected by, nor does it affect other submodels.

We say a structured model is in *logical product form* if the following property holds:

$$Next^e((s_K, \ldots, s_1)) = Next_K^e(s_K) \times \cdots \times Next_1^e(s_1) \tag{4.5}$$

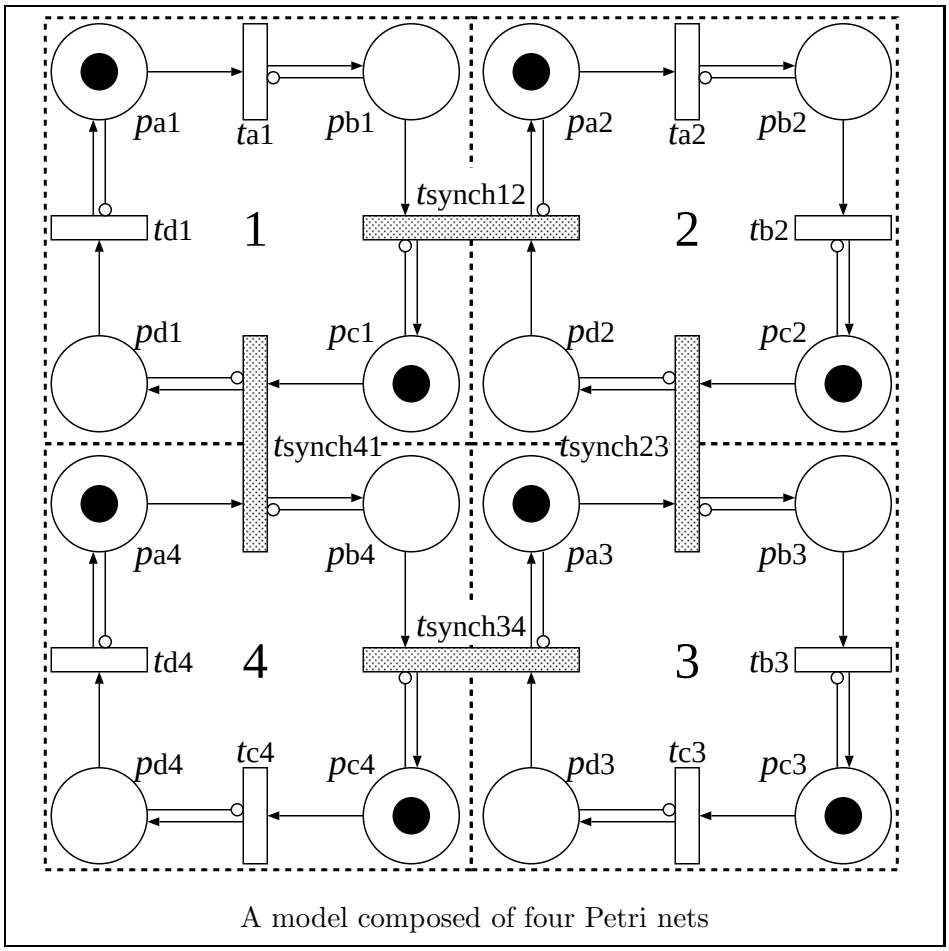


Figure 4.5: Example of a structured model

where $Next_k^e$ is the next state function due to event e for submodel k. Note that when $e \notin \mathcal{E}_k$ (i.e., event e does not affect submodel k), we require that $Next_k^e(s_k) = \{s_k\}$. This says that a submodel not affected by event e does not change its local state when event e occurs. Equation 4.5 says that the possible states reached by submodel k when event e occurs are independent of the local states of the other submodels. Note that if an event is disabled in one submodel, then it is disabled in the model; this is because a disabled event e in submodel k gives us $Next_k^e(s_k) = \emptyset$, which implies that $Next^e((s_K, \ldots, s_1)) = \emptyset$. Thus,

for each submodel, an event e must either be enabled in the submodel or must not affect the submodel if event e is to occur in the model.

Several example model structures are shown in Figure 4.6. Each of the models is composed of two Petri-net submodels, where submodel 1 contains place p_1 and submodel 2 contains place p_2, as indicated by the dashed rectangles. Model (a) is in logical product form, because we have

$$Next^t((\#p_2, \#p_1)) = Next^t_{p_2}(\#p_2) \times Next^t_{p_1}(\#p_1)$$

where the functions $Next^t_{p_2}$ and $Next^t_{p_1}$ are defined by

$$\begin{aligned} Next^t_{p_2}(\#p_2) &= \begin{cases} \emptyset & \text{if } \#p_2 = 0 \\ \{\#p_2 - 1\} & \text{if } \#p_2 > 0 \end{cases} \\ Next^t_{p_1}(\#p_1) &= \begin{cases} \emptyset & \text{if } \#p_1 = 0 \\ \{\#p_1 - 1\} & \text{if } \#p_1 > 0 \end{cases} . \end{aligned}$$

In model (b), we have a transition that is enabled when the number of tokens in p_2 is less than the number of tokens in p_1

$$Next^t((\#p_2, \#p_1)) = \begin{cases} \emptyset & \text{if } \#p_2 \geq \#p_1 \\ (\#p_2, \#p_1) & \text{if } \#p_2 < \#p_1 \end{cases}$$

which cannot be expressed as products of local functions. In model (c), we have a transition whose output into place p_2 depends upon the number of tokens in place p_1. Since p_1 and p_2 belong to different submodels, Equation 4.5 cannot be satisfied. Finally, model (d) has a transition whose enabling depends upon the number of tokens in both p_1 and p_2, and whose next state depends upon both p_1 and p_2. Thus, for dependencies of the type found in models (b), (c), and (d), places p_1 and p_2 must belong to the same submodel to obtain a *logical-product-form* model.

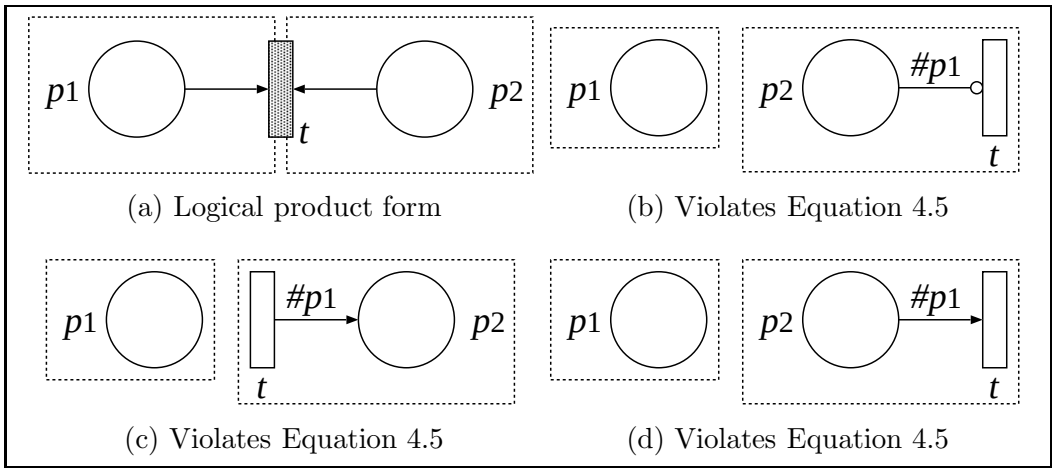


Figure 4.6: Structured models and logical product form

We say a logical-product-form model is in *Kronecker product form* if the rate and probability functions can be expressed as the product of local functions:

$$Rate^e((s_K, \ldots, s_1)) = \prod_{k=1}^{K} Rate_k^e(s_k) \tag{4.6}$$

$$Prob^e((s_K, \ldots, s_1), (s'_K, \ldots, s'_1)) = \prod_{k=1}^{K} Prob_k^e(s_k, s'_k). \tag{4.7}$$

Note that if $e \notin \mathcal{E}_k$, then we must have $Prob_k^e(s_k, s_k) = 1$ for all local states s_k. This is because submodel k does not change local state when an event occurs that does not affect the submodel. We also require that $Rate_k^e \equiv 1$ when $e \notin \mathcal{E}_k$, which means that submodel k cannot affect the rate of event e.

Equation 4.6 says that the rate of a synchronizing event is allowed to depend on the global state of the model, but only in a restricted way. Note in particular that Equation 4.6 says that the rate of an event local to submodel k can depend on the state of submodel k only. Thus a local event of a *logical-product-form* model with a rate dependency on the local state of another submodel becomes a synchronizing event when the model is considered in

Local Event	Rate	Synch Event	Rate
Ta_1	1.1	Tb_2	$1.5 + \#Pb_3$
Td_1	$2.1 + \#Pb_1$	$Tsynch_{12}$	1.2
Ta_2	1.7	$Tsynch_{23}$	$1.6 + \#Pb_2$
Tb_3	2.3	$Tsynch_{34}$	$(1 + 0.4 \cdot \#Pc_3)\cdot$
Tc_3	2.4		$(1.5 + 0.5 \cdot \#Pd_4)$
Tc_4	2.5	$Tsynch_{41}$	1.4
Td_4	2.6		

Table 4.2: Event rates for the structured model of Figure 4.5

Kronecker product form. For example, if the rates of the transitions of the Petri net in Figure 4.5 are assigned as shown in Table 4.2, then the model is in *Kronecker product form.* Event Tb_2 is a synchronizing event due to its rate dependency on submodel 3.

Chapter 5

Explicit State Space Generation

In this chapter, we investigate a variety of data structures for explicitly representing the set of reachable states $\mathcal{S}$. Section 5.1 presents and discusses a traditional, exhaustive-search algorithm for generating $\mathcal{S}$. Traditional data structures have been proposed for storing $\mathcal{S}$, including binary trees in [19] and hash tables in [49]; these are discussed in Section 5.2. Our contributions, some of which can be found in [27], are in Section 5.3 and Section 5.4. Section 5.3 describes a multi-level structure for storing $\mathcal{S}$ that can be used when the model is structured. An important contribution is the concept of the "locality" of an event: based on the structure of the model, we can determine which portion of the data structure must be modified due to a given event. This concept will be used again in later chapters. Section 5.4 describes efficient techniques that can be used when the components of a structured model can be studied in isolation. The techniques presented in the chapter are compared experimentally in Section 5.5. Finally, Section 5.6 presents conclusions.

```
Generate(Model M, State s°)
 1: U ← {s°}                                   • Unexplored states
 2: R ← {s°}                                   • Reachable states
 3: while U ≠ ∅ do
 4:    Choose an unexplored state s ∈ U
 5:    U ← U \ {s}
 6:    for each event e ∈ E do
 7:       for each state r ∈ Next^e(s) do
 8:          if r ∉ R then
 9:             R ← R ∪ {r}
10:             U ← U ∪ {r}
11:          end if
12:       end for
13:    end for
14: end while
15: return R
```

Algorithm 5.1: Traditional state space generation

5.1 Generation algorithm

One straightforward method for generating the set $\mathcal{S}$ is to use a systematic search of all states reachable from the initial state s°. This is the basis for Algorithm 5.1. Note that the algorithm will not terminate if the set $\mathcal{S}$ is infinite. In the algorithm, $\mathcal{R}$ stores the set of all states reachable from s° encountered so far, which is exactly $\mathcal{S}$ when the algorithm terminates. Another data structure, $\mathcal{U}$, is required for the set of states to explore. The order in which states are discovered depends strongly on the implementation of line 4. If we choose the state that has been unexplored for the longest amount of time (i.e., $\mathcal{U}$ is implemented as a queue), then we have a breadth-first search. If we choose the state that has been most recently added to the unexplored states (i.e., $\mathcal{U}$ is implemented as a stack), then we have a depth-first search.

The complexity of Algorithm 5.1 depends on the choice of data structures for $\mathcal{R}$ and $\mathcal{U}$.

The if statement of lines 8 through 11 will be executed for each transition from a reachable state to a reachable state. Since the number of transitions due to event e is exactly $\eta(\mathbf{R}^e)$, where $\mathbf{R}^e$ is the contribution to $\mathbf{R}$ due to event e, the if statement will be executed

$$\sum_{e \in \mathcal{E}} \eta(\mathbf{R}^e)$$

times. Note that this could be larger than $\eta(\mathbf{R})$, because one element of $\mathbf{R}$ could contain contributions from two or more events. The inner portion of the if statement will be executed exactly $|\mathcal{S}| - 1$ times. The complexity of the algorithm is then

$$O\left(C_1 \sum_{e \in \mathcal{E}} \eta(\mathbf{R}^e) + C_2 |\mathcal{S}|\right)$$

where C_1 is the complexity of determining whether a specific element belongs to $\mathcal{R}$ and C_2 is the complexity of adding an element to $\mathcal{R}$ and $\mathcal{U}$. Note that for many data structures, the operations "$r \notin \mathcal{R}$" and "$\mathcal{R} \leftarrow \mathcal{R} \cup \{r\}$" can be combined into a single operation. Thus, to efficiently generate and store $\mathcal{S}$, we must use a data structure that can represent $|\mathcal{S}|$ states efficiently, and allow for efficient insertions of new states and searches for states. If our interest is logical analysis only, then generation of $\mathcal{S}$ is our primary focus. If our interest is Markov analysis, then after $\mathcal{S}$ is generated, it will be used during the generation of the underlying CTMC. In particular, we will need to use our data structure for $\mathcal{S}$ to compute the index function Ψ. At this point, it may make sense to copy $\mathcal{S}$ into a less-dynamic structure in which insertions are no longer supported and computation of Ψ is efficient.

5.2 Traditional data structures

In this section we discuss several traditional structures, in which each state is explicitly stored along with some mechanism for inserting and searching for states. This introduces two orthogonal issues, namely how to store each state and how to store a set of states. State storage is discussed in Section 5.2.1, and set storage is discussed in Section 5.2.2.

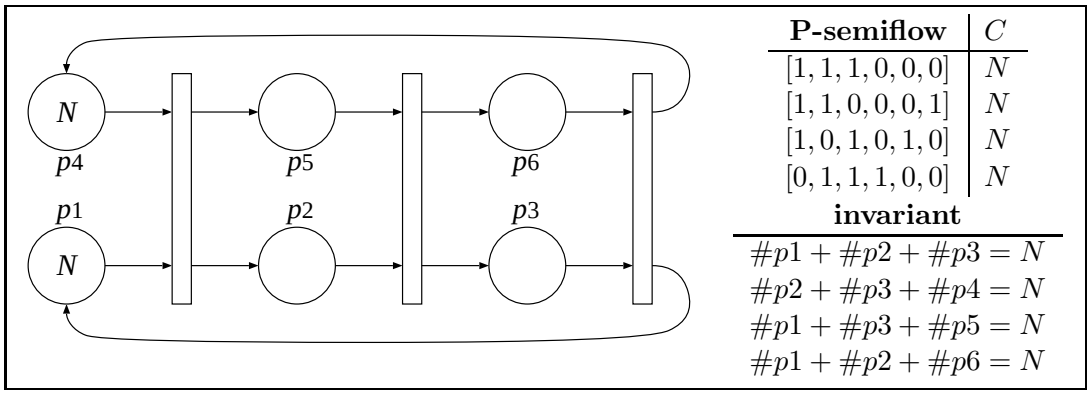
5.2.1 Storage of states

While we make no assumptions about the structure of a state, we do know that the set of potential states $\hat{\mathcal{S}}$ is finite. Thus, at the very least, each state s can be stored using its potential index $\hat{\Psi}(s)$. This requires $\left\lceil \log_2(|\hat{\mathcal{S}}|) \right\rceil$ bits per state. When the set $\hat{\mathcal{S}}$ is quite large (which can easily occur in practice), it may be more efficient to use some formalism-dependent information about states to store them efficiently.

For instance, a Petri-net marking can be stored by storing the number of tokens in each place, which is equivalent to storing the potential index. However, if the Petri net contains a large number of places, and in most markings only a few places contain tokens, we can store a marking using sparse storage and conserve memory. Also, the structure of the net can be used to reduce the number of places that must be stored [19]. For instance, a *P-semiflow* [74] is a vector of integers $\mathbf{y}$ of size $|\mathcal{P}|$ such that

$$\sum_{p \in \mathcal{P}} \mathbf{y}[p]\mathbf{m}[p] = C$$

where C is a constant, for all markings $\mathbf{m}$ reachable from the initial marking. Since the sum of two P-semiflows is also a P-semiflow, a Petri net has either zero or infinitely many



P-semiflow	C
$[1,1,1,0,0,0]$	N
$[1,1,0,0,0,1]$	N
$[1,0,1,0,1,0]$	N
$[0,1,1,1,0,0]$	N

invariant

$$\#p1 + \#p2 + \#p3 = N$$
$$\#p2 + \#p3 + \#p4 = N$$
$$\#p1 + \#p3 + \#p5 = N$$
$$\#p1 + \#p2 + \#p6 = N$$

Figure 5.1: Some P-semiflows and invariants of a Petri net

P-semiflows. The value of the constant C can be determined from the initial marking. A P-semiflow can be used to construct an *invariant* which holds for all markings reachable from the initial marking. The invariants for a Petri net can be determined [66] in a pre-processing step.

An example Petri net and some of its invariants are shown in Figure 5.1. Ignoring the invariants, we can store a marking of the net using 6 integers, corresponding to the number of tokens in each of the places $\{p_1, p_2, p_3, p_4, p_5, p_6\}$. Each integer requires $\lceil \log_2(N+1) \rceil$ bits since each place is N-bounded. The first invariant tells us that $\#p_1 + \#p_2 + \#p_3 = N$; thus, the number of tokens in place p_3 can be determined from the value of N and the number of tokens in places p_1 and p_2. Similarly, from the second, third, and fourth invariants we can determine $\#p_4$ from $\#p_2$ and $\#p_3$, $\#p_5$ from $\#p_1$ and $\#p_3$, and $\#p_6$ from $\#p_1$ and $\#p_2$. Thus, given a value of N, we only need to store the number of tokens in places p_1 and p_2, and from that information we can correctly determine the entire marking. Of course, this is a simple example, but it illustrates how invariants can be used to reduce the amount of storage required for a marking. From now on, we simply assume that a state can be

represented using some encoding that requires a certain number of bits, and we ignore the specifics of how each state is encoded.

5.2.2 Storing a set of states

Once we have a way to store each state, we then must choose an efficient data structure for collecting a set of states. The data structure we use must be able to efficiently insert new states and search for existing states. Due to the possibility of an extremely large number of states, we cannot use elementary data structures such as linked lists which require linear searches. Some of the traditional data structures that can be used are discussed in this section. However, once we have finished constructing the state space, we can create a sorted array of states (or an array of state indices), since insertions are no longer required. After the array of states has been created, we may destroy the structure used during state space generation. Then, to compute a state's index (the function Ψ), we simply perform a binary search on the array for the given state.

Hash tables

A hash table [33, 60] is based on a table with M entries to store objects. The location of each object in the table is determined by applying a *hashing* function h to the object. In our case, we have $h : \hat{\mathcal{S}} \rightarrow \{0, \ldots, M-1\}$, since we wish to store states. Depending on our choice of hashing function, we may have *collisions*, where two different states hash to the same table location (i.e., $h(s_1) = h(s_2),\ s_1 \neq s_2$). If collisions are possible, some arrangement must be made for states that are assigned to the same location. The technique we consider is *chaining*, where each entry in the table is a linked list of the states that hash to that

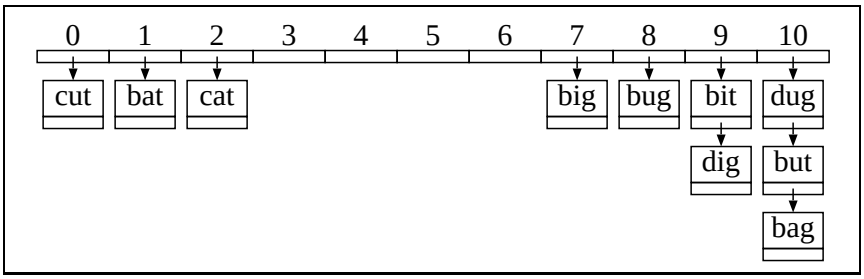


Figure 5.2: A hash table with chaining

location. Other techniques such as linear probing can also be used to handle collisions. An example of a hash table with chaining is shown in Figure 5.2.

There are two main difficulties with using hash tables to effectively store the set of actual states: choosing the hashing function h and choosing the table size M. Ideally, our hashing function will distribute the states evenly over the M possible table locations. This can be difficult due to the complex interactions between states. For instance, suppose we have the Petri-net model shown in Figure 5.1, and we choose

$$h = (\#p_1 + \#p_2 + \#p_5 + \#p_6) \bmod M$$

as our hashing function. Looking at the invariants, we find that our hashing function reduces to $h = (N + \#p_5) \bmod M$. This means that reachable states that have the same number of tokens in place p_5 are stored at the same location. Since the number of reachable states corresponding to place p_5 containing exactly $n \leq N$ tokens is $1 + N - n$, we know that the chain at table location $N + n \bmod M$ has exactly $1 + N - n$ entries. Thus, regardless of the size of M, only $N + 1$ locations of the table are used. An even worse hashing function is

$$h = (\#p_1 + \#p_2 + \#p_3 + \#p_4 + \#p_5 + \#p_6) \bmod M$$

where the entire state space is stored at table location $2N \bmod M$.

Assuming a uniform distribution of hash locations, searches and insertions into a table of size M with N elements require $O(1 + \frac{N}{M})$ operations on average. Regardless of the distribution of hash locations, a hash table of size M with N elements using chaining requires $O(N + M)$ memory. From a computation standpoint, we want to choose M larger than the number of states, so that searches and insertions take constant time on average. From a memory standpoint, we want M as small as possible. Thus, we want M to be about the same size as $\mathcal{S}$, so that the overall memory requirements are still $O(|\mathcal{S}|)$. Unfortunately, at the moment it not known how to determine tight bounds on the size of $\mathcal{S}$ in general without actually generating $\mathcal{S}$.

One possible solution to this problem is to expand the size of the hash table when it becomes too full. Then we can start with a relatively small value for M. Once the number of elements in the table states exceeds some threshold, such as $N > M$, we can increase the size of the table to M'. Then we must use a new hashing function $h_{M'}$ and re-hash the elements of the old table to new locations in the enlarged table. Obviously, we want to minimize the number of times this costly operation must be performed. By doubling the hash table size each time (i.e., using $M' \approx 2 \cdot M$) we can limit the number of table enlargements to $\lceil \log_2(\mathit{upper}) - \log_2(\mathit{lower}) \rceil$ where *upper* is the final size of the table and *lower* is the starting size of the table. Using an initial table size of about one thousand locations, only 20 enlargements are required to reach a table size of about one billion locations, which is beyond the storage capabilities of today's workstations.

Binary trees

Another traditional data structure for storing a set is a binary tree [60]. To use a binary tree, the elements to be stored must be ordered in some way. Usually, we use the potential index function $\hat{\Psi}$ to order the states. For a given node in the binary tree that represents state s, we have that any state s_l stored in a left descendant of the current node has a smaller index than s (i.e., $\hat{\Psi}(s_l) < \hat{\Psi}(s)$). Similarly, any state s_r stored in a right descendant of the current node has larger index than s (i.e., $\hat{\Psi}(s) < \hat{\Psi}(s_r)$). Thus, an *inorder* traversal [58] of the tree visits states in the order of their potential index.

The advantage of binary trees over hashing tables is that a tree representing N elements always uses $O(N)$ memory, and we are not burdened with the task of choosing a good hashing function. However, binary trees have the same worst-case behavior as hash tables unless the trees are kept *balanced*, where each node has roughly the same number of left descendants as right descendants. We consider two different strategies for tree balancing.

AVL trees [1, 60, 104] store balance information in each node that is used to keep trees "mostly" balanced. Formally, a binary tree satisfies the *AVL property* if for every node n in the tree,

$$|Height(Left(n)) - Height(Right(n))| \leq 1$$

where $Height(n)$ is the height of the tree with root node n and $Left(n)$ and $Right(n)$ are the left and right children of node n, respectively. To maintain the AVL property, each node must store the balance information $Height(Left(n)) - Height(Right(n))$ which requires 2 additional bits per node, since the possible values are $-1, 0$ and 1.

Insertions and deletions of nodes are accompanied by a series of rotations to maintain the AVL property. It has been shown that the height of any AVL tree with N nodes is $O(\log_2(N))$. Thus searches, insertions, and deletions in AVL trees cost $O(\log_2(N))$ in the average and worst case.

Splay trees [60, 94] are based on the *splay* operation which moves a specified element to the root of the tree via a series of rotations along the search path. Searches, insertions, and deletions are all implemented using splay operations. Thus, recently and frequently accessed items are near the root of the tree, while the leaves of the tree contain items that have not been accessed for some time. It has been shown [94] that a worst-case sequence of m splay operations on a tree with N nodes has a time complexity bound of $O(m \log_2 N)$.

Thus, the behavior of a balanced binary tree is worse than a hash table with a uniform hashing function, but much better than a hash table with a badly chosen hashing function.

5.2.3 Storage of unexplored states

During state space generation, we must be able to find some unexplored state quickly. Since we know that $\mathcal{U} \subseteq \mathcal{R}$, we only need to indicate which states in $\mathcal{R}$ belong to $\mathcal{U}$ in an efficient way, so that we can easily choose and remove an unexplored state. The techniques we discuss here will work for both hash tables and binary trees, since both data structures have a node associated with each state. In the discussion, we will refer to a binary tree. We consider three ways to maintain the unexplored states.

1. Each node stores an extra pointer to the next unexplored state, and we maintain a

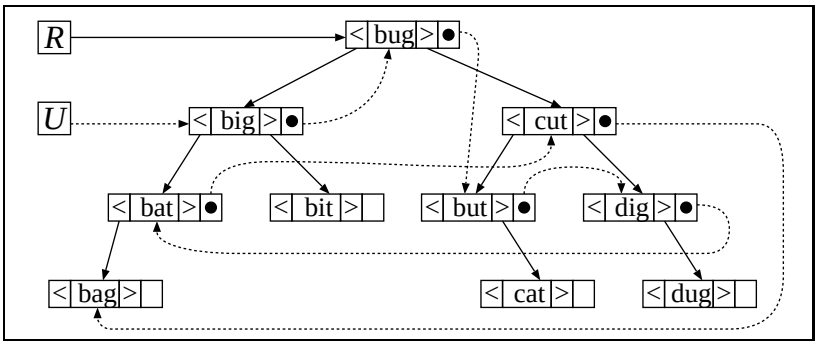


Figure 5.3: Storing unexplored states using an extra pointer per node

pointer to the front of the list. This structure requires an additional pointer for every node and some list maintenance. An example of this structure is shown in Figure 5.3, where the states are stored using a binary tree. A drawback of this structure is that the states that have been explored do not need the extra pointer. Thus, when $\mathcal{U}$ is a small subset of $\mathcal{R}$ (which occurs when Algorithm 5.1 is nearly finished), a large amount of memory is wasted.

2. To fix the wasted memory problem with the first structure, we could maintain a separate linked list of pointers to the unexplored states. This structure requires two additional pointers for each unexplored state and the same list maintenance overhead. In addition, there will be some memory management overhead, since the size of the set $\mathcal{U}$ may grow and shrink several times during the computation. This structure is illustrated in Figure 5.4 using the same binary tree as the previous example.

3. A more memory-efficient version of the linked-list structure is to use an array to store the states. With this structure, we implement $\mathcal{U}$ as a queue, and we exploit the property that once a state is removed from $\mathcal{U}$, it will never again be added to $\mathcal{U}$.

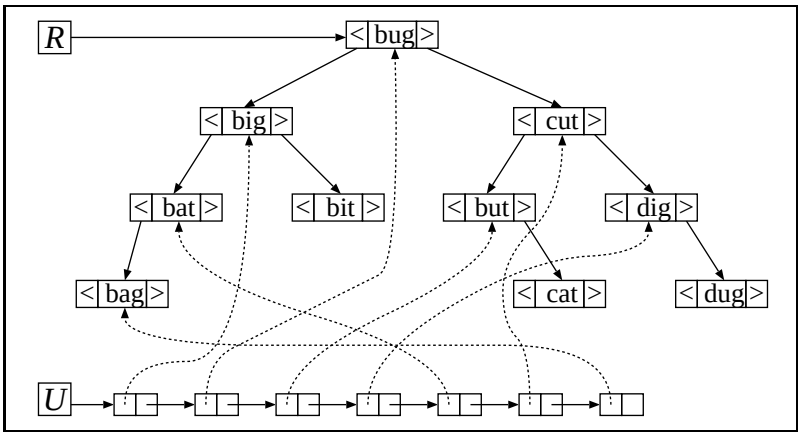


Figure 5.4: Storing unexplored states using a linked list

We use an array to store the states in the set $\mathcal{R}$, and each node of our data structure representing $\mathcal{R}$ stores an array index instead of a state. This is illustrated in Figure 5.5, using the same binary tree of the previous example. The set $\mathcal{R} \setminus \mathcal{U}$ occupies the beginning of the array, and the set $\mathcal{U}$ occupies the end of the array. To maintain $\mathcal{U}$, we simply store the index of the first unexplored state, which is the head of the queue. The tail of the queue is also stored as a count of the number of states in the set $\mathcal{R}$. Thus, when a new state is discovered and added to $\mathcal{R}$, it is added to the end of the array and the tail pointer is incremented. This structure requires an additional integer per state.

We prefer the third data structure, because no list manipulations are required. With the other two structures, we have the choice of storing the states in the nodes themselves (as depicted in Figure 5.3 and Figure 5.4), or storing the states in a separate structure, at a cost of an additional integer per state (as depicted in Figure 5.5). A separate structure may be desired to simplify node manipulation, especially if the states are of varying size.

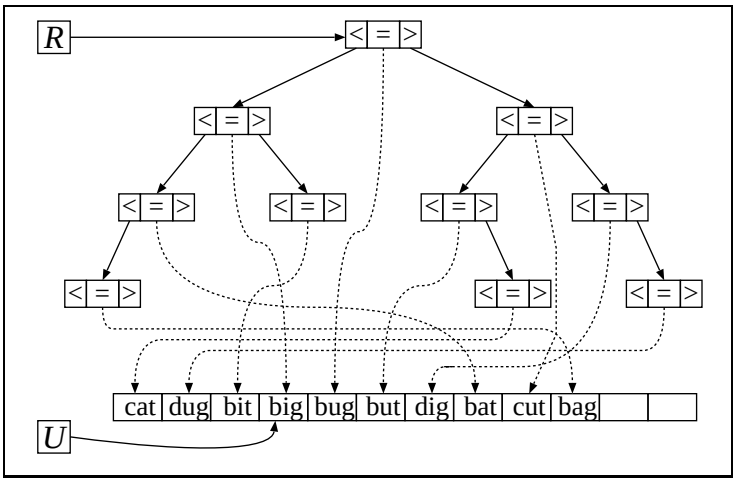


Figure 5.5: Storing unexplored states using an array

In addition, if the separate structure is an array, then the compression step is simplified.

5.2.4 Compression

After the state space has been generated, we may compress our data structure of choice somewhat since we no longer need to handle insertions. Since we still need to search for states, we need a mechanism for ordering the states according to $\hat{\Psi}$, so that we can perform a binary search. One possibility is to use a single array of states, sorted according to $\hat{\Psi}$. This may complicate our binary search if the sizes of the states vary. In this case, our indexing function Ψ corresponds to the reachable subset of $\hat{\Psi}$. If we used a binary tree during generation, then the array can be built by performing an inorder traversal of the tree. If we used a hash table, then the array must be built using a sorting routine. An example of such a sorted array is shown in Figure 5.6.

Another technique is to use two arrays, one to store the states according to some other indexing function Ψ, and another to order the states according to $\hat{\Psi}$. Usually, the second

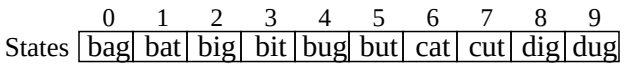


Figure 5.6: Compression using a sorted array of states

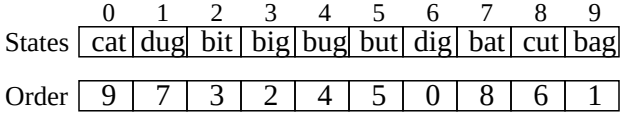


Figure 5.7: Compression using an unordered array of states and an ordering array

array is implemented as an array of integers, where each integer is an index into the array of states. As with the single-array technique, we can use an inorder traversal to generate the second array if we used a binary tree during generation, otherwise a sorting routine is necessary. An example of this two-array technique is shown in Figure 5.7.

The main difference between these two techniques (aside from the memory requirements) is the actual indexing function Ψ. Using the first technique, we are confined to using an indexing function that corresponds to the ordering of the potential states, which is equivalent to *lexical* order. With the second technique, we can use any indexing function. The price of this generality is an array of integers of size $|\mathcal{S}|$. Usually, the ordering we use is *canonical* order, the order in which the states were discovered using Algorithm 5.1. This is the natural order of the states if we use a separate data structure for state storage as shown in Figure 5.5. With some linear solvers such as Gauss-Seidel, the ordering of the states can affect the convergence rate. Lexical ordering of the states is not necessarily a good order, and in fact it has been shown [70] that lexical ordering can cause Gauss-Seidel to perform quite poorly.

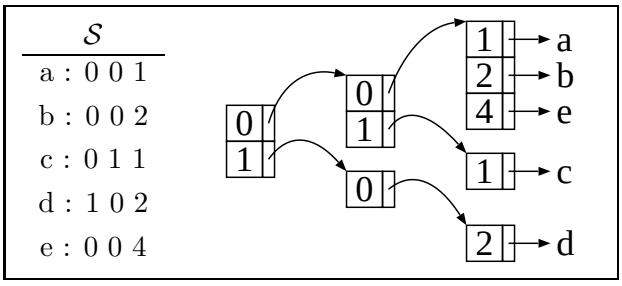


Figure 5.8: Chiola's multi-level technique to store the reachability set of a SPN

5.3 Multi-level trees for structured models

We extend work by Chiola [19], who defined a multi-level structure to store reachable states of a stochastic Petri net (SPN). Since the state of a SPN with K places can be represented by a vector $s \in \mathbb{N}^K$, he proposes a structure in which each component of the vector is stored in a different level. An example of this idea is illustrated in Figure 5.8, copied from [19]. In the figure, we have $K = 3$, and the state space $\mathcal{S} = \{a, b, c, d, e\}$ is stored in a three-level tree. The first level stores the first component of each state and partitions $\mathcal{S}$ based upon the first component. The second level then further subdivides $\mathcal{S}$ based on the second component. Finally, the third level stores the third component of each state. A state in $\mathcal{S}$ can be found by tracing a path from the first through the third levels.

We extend this technique to structured models in general. In our case, a state is composed of K substates, and so we store the substates of submodel k at level k. Chiola's technique is equivalent to our technique when the model is composed of K SPNs, where each SPN contains exactly one place.

For a more formal definition of our technique, we define the set function $\mathcal{B}_k : \mathcal{S}_K \times \cdots \times$

$\mathcal{S}_{k+1} \rightarrow 2^{\mathcal{S}_k \times \cdots \times \mathcal{S}_1}$ as

$$\mathcal{B}_k(s_K, \ldots, s_{k+1}) \equiv \{(s_k, \ldots, s_1) : (s_K, \ldots, s_1) \in \mathcal{S}\} \tag{5.1}$$

for all submodels k. Intuitively, for fixed local states $s_K, \ldots, s_{k+1}$, the set $\mathcal{B}_k(s_K, \ldots, s_{k+1})$ is the set of local states $s_k, \ldots, s_1$ "beyond" s_{k+1} such that the given combination of local states $(s_K, \ldots, s_1)$ is a reachable state. The collection of all sets $\mathcal{B}_k(s_K, \ldots, s_{k+1})$ for a fixed k, for all possible argument combinations $(s_K, \ldots, s_{k+1}) \in \mathcal{S}_K \times \cdots \times \mathcal{S}_{k+1}$, is called *level* k. Note that level K consists of $\mathcal{B}_K$ only, which is equal to $\mathcal{S}$. Equation 5.1 can also be written as a recurrence

$$\mathcal{B}_k(s_K, \ldots, s_{k+1}) = \bigcup_{s_k \in \mathcal{S}_k} \{s_k\} \times \mathcal{B}_{k-1}(s_K, \ldots, s_k) \tag{5.2}$$

with terminating condition

$$\mathcal{B}_1(s_K, \ldots, s_2) = \{s_1 : (s_K, \ldots, s_1) \in \mathcal{S}\} \tag{5.3}$$

for the case $k = 1$. An example illustrating Equation 5.2 and Equation 5.3 is shown in Figure 5.9 for a model consisting of 3 submodels. The local states of the submodels are listed in the figure.

We represent a set $\mathcal{B}_k(s_K, \ldots, s_{k+1})$ according to Equation 5.2 by storing each substate s_k along with a "downward" pointer to the set $\mathcal{B}_{k+1}(s_K, \ldots, s_k)$. In the terminal case, given by Equation 5.3, we simply represent $\mathcal{B}_1(s_K, \ldots, s_2)$ as a set of substates. That is, downward pointers are not needed at level 1.

One data structure that can be used to store the set of substate-pointer pairs is a binary tree, such as an AVL or splay tree. At level 1 we have ordinary AVL or splay trees, and at

Level 3	$\mathcal{S} = \mathcal{B}_3 = \{b\} \times \mathcal{B}_2(b) \cup \{c\} \times \mathcal{B}_2(c) \cup \{d\} \times \mathcal{B}_2(d)$		
	$= \{bag, bat, big, bit, bug, but, cat, cut, dig, dug\}$		
Level 2	$\mathcal{B}_2(b) = \{a\} \times \mathcal{B}_1(b,a) \cup \{i\} \times \mathcal{B}_1(b,i) \cup \{u\} \times \mathcal{B}_1(b,u)$		
	$= \{ag, at, ig, it, ug, ut\}$		
	$\mathcal{B}_2(c) = \{a\} \times \mathcal{B}_1(c,a) \cup \{i\} \times \mathcal{B}_1(c,i) \cup \{u\} \times \mathcal{B}_1(c,u)$		
	$= \{at, ut\}$		
	$\mathcal{B}_2(d) = \{a\} \times \mathcal{B}_1(d,a) \cup \{i\} \times \mathcal{B}_1(d,i) \cup \{u\} \times \mathcal{B}_1(d,u)$		
	$= \{ig, ug\}$		
Level 1	$\mathcal{B}_1(b,a) = \{g,t\}$	$\mathcal{B}_1(b,i) = \{g,t\}$	$\mathcal{B}_1(b,u) = \{g,t\}$
	$\mathcal{B}_1(c,a) = \{t\}$	$\mathcal{B}_1(c,i) = \{\}$	$\mathcal{B}_1(c,u) = \{t\}$
	$\mathcal{B}_1(d,a) = \{\}$	$\mathcal{B}_1(d,i) = \{g\}$	$\mathcal{B}_1(d,u) = \{g\}$
Locals	$\mathcal{S}_3 = \{b,c,d\}$	$\mathcal{S}_2 = \{a,i,u\}$	$\mathcal{S}_1 = \{g,t\}$

Figure 5.9: Mathematical representation of a multi-level structure

the other levels we have AVL or splay trees whose nodes contain an additional downward pointer. Nodes with a null downward pointer (corresponding to an empty set) are not stored in the tree. We call this structure a *multi-level tree*.

An example of a multi-level tree depicting the sets of Figure 5.9 is shown in Figure 5.10. In the example, we see that non-reachable combinations are not represented. For instance, in the level-2 set $\mathcal{B}_2(c)$, the node corresponding to substate i is absent, since there is no reachable state of the form $(c, i, \bullet)$, and as a result the set $\mathcal{B}_1(c,i)$ is empty. Thus, if the tree did contain a node for substate i, its corresponding downward pointer would be null.

Note that since a multi-level tree only stores reachable combinations, any path from the root of the level-K tree to a level-1 node corresponds to a reachable state (given by the last substate visited in each level-k tree) and vice-versa. Thus, the total number of level-1 nodes is exactly equal to the number of reachable states $|\mathcal{S}|$. To determine if a given state $(s_K, \ldots, s_1)$ is reachable in a multi-level tree, we start at level K and search for s_K. If

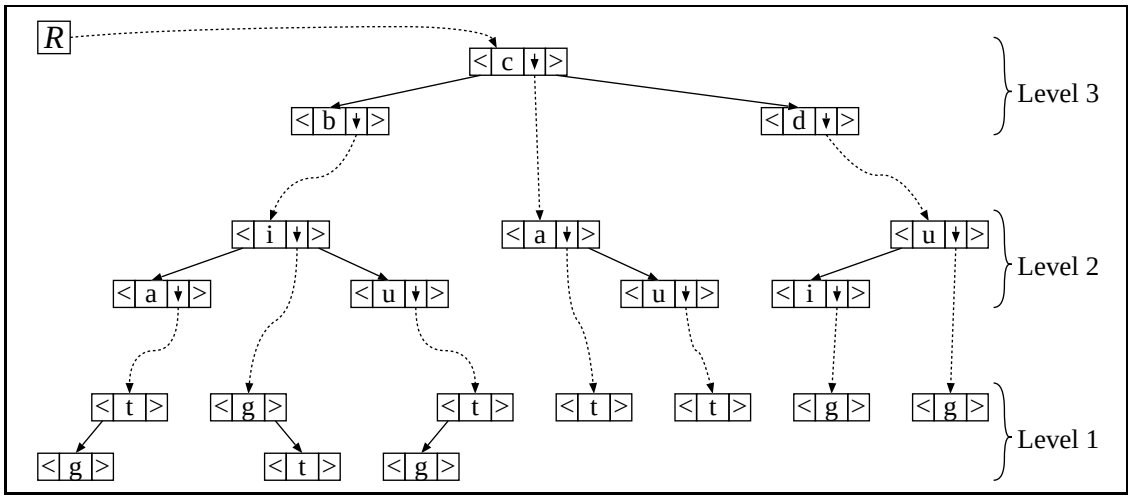


Figure 5.10: A multi-level tree representing the structure in Figure 5.9

we do not find s_K, then the state is not reachable. Otherwise, we follow the downward pointer associated with s_K and search for s_{K-1} in that level-$(K-1)$ tree. This process is continued until we either fail to find a substate s_k in the specified level-k tree, in which case the state is not reachable, or we find s_1 in the specified level-1 tree, in which case the state is reachable. Inserting states into a multi-level tree is a similar process and is specified in Algorithm 5.2.

As with traditional structures, we have the choice of storing the substates within the tree nodes, or storing the substates in a separate list structure. However, if the number of local states is small, then the number of bits required to encode the substate indices in the nodes will be relatively small. Since the number of nodes in a level-k tree can be at most $|\mathcal{S}_k|$, we can also use small indices for the left and right pointers. For instance, if the number of local states in level k is less than 256, we can use one-byte integers for the left, right, and substate indices.

```
Insert(tree t, level k, state (s_K, ..., s_1))
 1: if k = 1 then                                  • At the bottom level
 2:    if t does not contain s_1 then
 3:       Add s_1 to tree t
 4:    end if
 5: else                                           • Not at the bottom
 6:    Find s_k in tree t
 7:    if s_k is not in tree t then
 8:       Create node n for s_k in tree t
 9:    end if
10:    t_d ← the downward pointer associated with s_k
11:    Insert(t_d, k − 1, (s_K, ..., s_1))
12: end if
```

Algorithm 5.2: Inserting a state into a multi-level tree

Given that searches and insertions in AVL and splay trees with n nodes take $O(\log_2 n)$ time, we can compute the time complexity of searches and insertions in a multi-level tree. Since an insertion requires us to perform a (balanced) binary tree search at each level, the complexity of an insertion is

$$O\left(\sum_{k=1}^{K} \log_2 T_k\right) \tag{5.4}$$

where T_k is the number of nodes in the level-k tree encountered during our insertion. This is also the time required for a successful search. The worst case for an insertion or successful search has complexity

$$\begin{aligned} O\left(\sum_{k=1}^{K} \log_2 T_k\right) &= O\left(\sum_{k=1}^{K} \log_2 |\mathcal{S}_k|\right) \\ &= O\left(\log_2 \prod_{k=1}^{K} |\mathcal{S}_k|\right) \\ &= O(\log_2 |\hat{\mathcal{S}}|) \end{aligned}$$

which occurs when the trees encountered are as large as possible; namely, for each level k,

the tree has $|\mathcal{S}_k|$ nodes. Unsuccessful searches may require less time, since an unsuccessful search may terminate at a level other than level 1. In the best case, an unsuccessful search terminates at level K, which requires $O(T_K)$ time.

We now consider the average case of insertions and successful searches, where we assume that each level-k tree contains the same number of nodes. Since the number of trees at level k is exactly equal to the total number of nodes at level $k+1$, we have $T_k = \frac{N_k}{N_{k+1}}$, where N_k is the total number of nodes in level k of our multi-level tree. Substituting into Equation 5.4 gives us

$$\begin{aligned} O\left(\sum_{k=1}^{K} \log_2 T_k\right) &= O\left(\log_2\left(\frac{N_K}{1}\right) + \log_2\left(\frac{N_{K-1}}{N_K}\right) + \cdots + \log_2\left(\frac{N_1}{N_2}\right)\right) \\ &= O\left(\log_2\left(\frac{N_K}{1} \cdot \frac{N_{K-1}}{N_K} \cdots \frac{N_1}{N_2}\right)\right) \\ &= O\left(\log_2 N_1\right) \end{aligned}$$

where the denominator 1 is due to the fact that there is always exactly one tree at level K. Recall that N_1 is also equal to the number of states represented by the multi-level tree. Thus searches and insertions in a multi-level tree representing n states take $O(\log_2 n)$ time, just like ordinary "single-level" AVL and splay trees.

5.3.1 Representing unexplored states

Recall that Algorithm 5.1 requires data structures for both $\mathcal{R}$, the set of states discovered so far, and $\mathcal{U}$, the set of discovered states still to be explored. We can easily use a multi-level tree to represent $\mathcal{R}$, since it only requires searches and insertions. Unfortunately, the methods described earlier (in Figure 5.3, Figure 5.4, and Figure 5.5) for representing $\mathcal{U}$ cannot be applied to a multi-level tree, unless each node also stores an "upward" pointer.

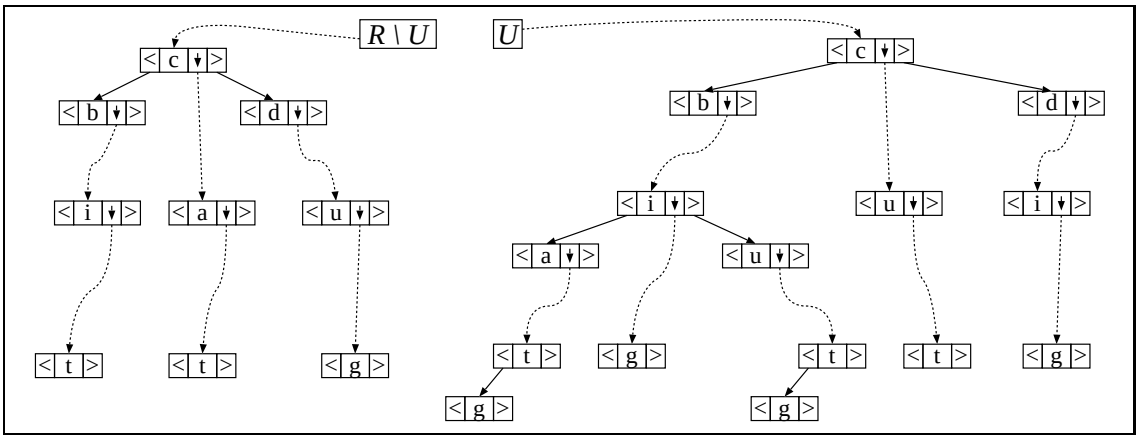


Figure 5.11: Representing $\mathcal{U}$ with a second multi-level tree

Instead, we consider three alternatives for maintaining the unexplored states when $\mathcal{R}$ is stored using a multi-level tree.

1. Store $\mathcal{U}$ as a linked list of states. Since complete states are not stored explicitly with a multi-level tree, we cannot simply use a pointer as in Figure 5.4. Instead, we must store the complete state in our linked list. As the set $\mathcal{U}$ can become quite large while Algorithm 5.1 progresses, this data structure for $\mathcal{U}$ could require a substantial amount of memory.

2. Store $\mathcal{U}$ using another multi-level tree. In this case, it may be desirable to modify Algorithm 5.1 so that we store $\mathcal{R} \setminus \mathcal{U}$ in one multi-level tree, and $\mathcal{U}$ in another. This avoids duplication of an unexplored state in both multi-level trees. An example of this two-tree technique is shown in Figure 5.11. The main drawback with this data structure is that insertions into and deletions from $\mathcal{U}$ are no longer constant operations. Also, the deletion of a specific node from an AVL or splay tree is a rather costly operation: it requires us to first search for the node, then delete the

```
Remove(tree t, level k, Out: state (s_K, ..., s_1))
 1: if k = 1 then                                   • At the bottom level
 2:    Choose some node n to remove from tree t
 3:    s_1 ← the substate stored in n
 4:    Remove n from tree t
 5: else                                            • Not at the bottom
 6:    Choose some node n to potentially remove
 7:    s_k ← the substate stored in n
 8:    t_d ← downward pointer in n
 9:    Remove(t_d, k − 1, (s_K, ..., s_1))
10:    if tree t_d is empty then                    • We removed the last node from t_d
11:       Remove n from tree t
12:    end if
13: end if
```

Algorithm 5.3: Choosing and removing a state from a multi-level tree

node, then re-balance the tree. We can avoid this overhead with the realization that Algorithm 5.1 allows us to choose the state we remove from $\mathcal{U}$. Algorithm 5.3 specifies how to choose and remove a state from a multi-level tree. The decision of which node to remove (lines 2 and 6) is different for AVL and splay trees. With AVL trees, we start at the root node and use the balance information to select the subtree with greater height (or arbitrarily choose if the left and right subtrees have the same height). This continues until we reach a leaf node, which is the node we remove. Using this selection algorithm, no rotations are required; we need only to update balance information along the search path. With splay trees, we remove the root node, and then perform a single splay operation to restore the tree.

3. Store $\mathcal{U}$ as a linked list for each tree in the multi-level structure. In these data structures, we say a level-k node is "unexplored" if the tree reached by following its downward pointer contains an unexplored node, and a level-1 node is "unexplored" if

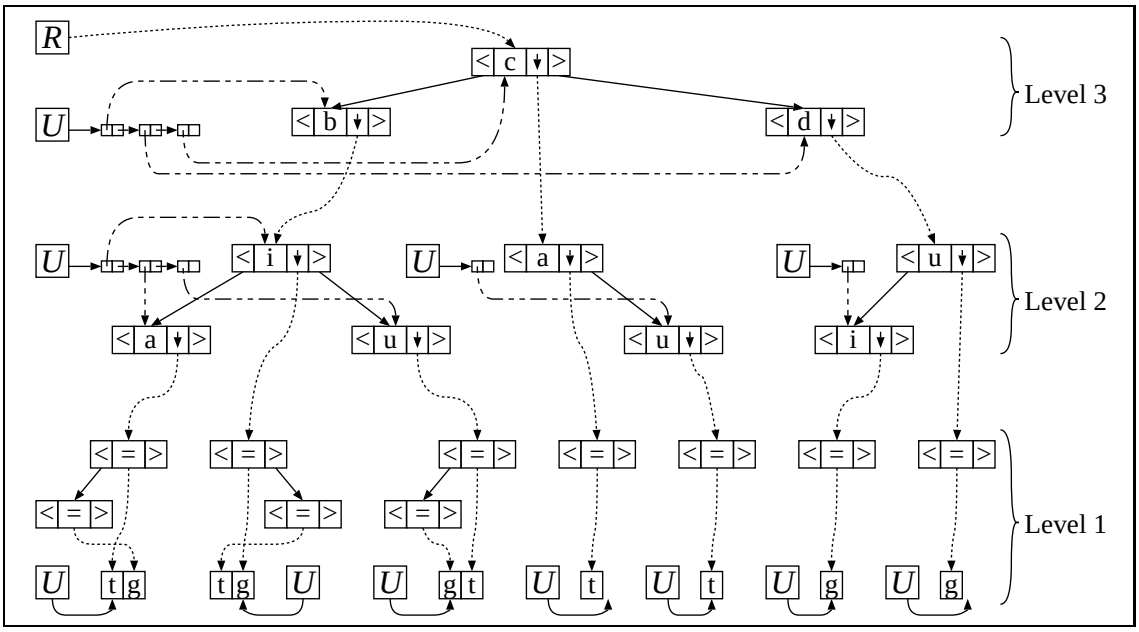


Figure 5.12: Representing $\mathcal{U}$ and $\mathcal{R}$ in one multi-level tree

the state corresponding to the path to the level-1 node is in the set $\mathcal{U}$. We can then find an unexplored state by following a path consisting only of unexplored nodes. The algorithm to remove an unexplored state is similar to that of Algorithm 5.3. Note that a level-1 node will never again become "unexplored" once it has been "explored", but this is not true for the other levels. Thus, we can use an array to represent the unexplored level-1 nodes as shown in Figure 5.5, while the unexplored level-k nodes are instead stored with a linked list as shown in Figure 5.4. This data structure for $\mathcal{U}$ uses an array for each level-1 tree and a linked list for each tree not at level 1. An example of such a structure is shown in Figure 5.12. The memory required for these lists is comparable to the previous technique where $\mathcal{U}$ is stored in its own multi-level tree. However, the time required to insert into or delete from this representation of $\mathcal{U}$ is only $O(K)$, since we can insert into or delete from the linked list at each level in

constant time.

We use either the second or third data structure for $\mathcal{U}$ in practice.

5.3.2 Exploiting locality

Suppose we search for a state $(s_K, \ldots, s_1)$ in a multi-level tree. First we search for s_K in the level-K tree representing $\mathcal{B}_K$. The node containing s_K gives us the downward pointer to the tree representing $\mathcal{B}_{K-1}(s_K)$. We then search for s_{K-1} in that level-$(K-1)$ tree and so on, until we finally find s_1 in the tree representing $\mathcal{B}_1(s_K, \ldots, s_2)$. If we then wish to search for the state $(s_K, \ldots, s_{k+1}, s'_k, \ldots, s'_1)$, we can immediately search for s'_k in the level-k tree representing $\mathcal{B}_k(s_K, \ldots, s_{k+1})$ instead of starting at the level-K tree, assuming we saved the pointer to the level-k tree. This idea can be exploited if we consider how events can modify the state of the model. We know by definition that when event e occurs in state $(s_K, \ldots, s_1)$, the new state must be of the form $(s_K, \ldots, s_{\mathit{first}(e)+1}, s'_{\mathit{first}(e)}, \ldots, s'_1)$. Upon exploration of a new state s, we remove s from $\mathcal{U}$. Either $\mathcal{U}$ is stored in its own multi-level tree, in which case s must then be inserted into $\mathcal{R} \setminus \mathcal{U}$, or $\mathcal{U}$ is implemented as pointers to nodes in $\mathcal{R}$. In either case, we can save the pointers to the trees representing $\mathcal{B}_K, \ldots, \mathcal{B}_1(s_K, \ldots, s_2)$. If event $e \in \mathcal{E}$ can occur in state s, we generate $s' = \mathit{Next}^e(s)$, and we know for certain that substates K through $\mathit{first}(e)+1$ are the same in s and s'. Then to search for s', we begin at level $\mathit{first}(e)$ with the tree $\mathcal{B}_{\mathit{first}(e)}(s_K, \ldots, s_{\mathit{first}(e)+1})$. We showed in [27] that this technique can reduce execution time by as much as 20% at a negligible cost (an array of $K-1$ pointers).

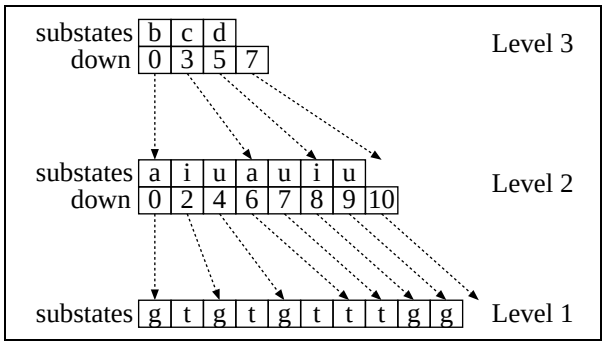


Figure 5.13: The compressed version of the multi-level tree of Figure 5.10

5.3.3 Compression

Once the state space has been generated, we never need to insert or delete states. Each of the trees in the multi-level structure can then be compressed into an ordered array, eliminating the left and right pointers. All of the arrays for a given level are combined into a single array, allowing the use of integer array indices for the next-level pointers. Then an array element at level $k > 1$ simply stores a substate index and a pointer to the next level. At level 1, there is no need to store a pointer to the next level, so the level-1 array stores only the substate indices. For any given level k, if we know that $|\mathcal{S}_k| < 256$, then we use one-byte substate indices, and if $256 \leq |\mathcal{S}_k| < 65,536$ then we use two-byte substate indices. Otherwise, we use four-byte substate indices. Since most nodes are level-1 nodes, this means our compressed representation of $\mathcal{S}$ requires slightly more than $|\mathcal{S}|$ or $2 \cdot |\mathcal{S}|$ bytes, depending on the size of $\mathcal{S}_1$. If $\mathcal{S}_1$ is very large, then the compressed representation will require about $4 \cdot |\mathcal{S}|$ bytes. An example of a compressed multi-level structure is illustrated in Figure 5.13. In the figure, the actual substates are shown in the substate arrays (instead of their indices) for clarity.

The level-1 array of substate indices also provides a method for computing $\Psi(s)$. When we search for state $(s_K, \ldots, s_1) \in \mathcal{S}$, the position of the final component s_1 in the level-1 array of substate indices is exactly $\Psi(s_K, \ldots, s_1)$. Since the time to perform a binary search on an array of size n is $O(\log_2 n)$, the search time for the compressed structure is the same as for the multi-level tree structure. Thus the time required to compute $\Psi(s)$ is $O(\log_2 |\mathcal{S}|)$ on average.

To compute our reward function, we must be able to efficiently enumerate the states. If the states are stored explicitly in an array, we can simply traverse the entire array. If the states are instead stored in a compressed multi-level structure, we use an array of K indices. Index k in the array specifies the current level-k array element. The first state is given by the first array element at each level. To obtain the next state, we advance the level-1 index. If an advancing index at any level k crosses a subtree boundary, then we advance the index of level $k + 1$. This continues until each array has been completely visited, which occurs when the level-1 index has visited $|\mathcal{S}|$ elements. Thus, the complexity of enumerating $\mathcal{S}$ is the sum of the sizes of the arrays at each level, which is slightly larger than $|\mathcal{S}|$.

5.4 Generating local states first

For some structured models, it is possible to build the set of local states $\mathcal{S}_k$ in isolation. If this is the case, then it can be beneficial to generate $\mathcal{S}_K, \ldots, \mathcal{S}_1$ before generating $\mathcal{S}$. A state s can be represented as an integer vector $\mathbf{i}$, where $\mathbf{i}[k]$ is the index of substate s_k. If the local states are known, then we know that $\mathbf{i}[k]$ can assume $|\mathcal{S}_k|$ possible values. We will use $\mathbf{i}$ when we talk about states with the understanding that $\mathbf{i}$ represents the state

$({\Psi_K}^{-1}(\mathbf{i}[K]), \ldots, {\Psi_1}^{-1}(\mathbf{i}[1]))$.

In this section, we discuss data structures to store $\mathcal{S}$ that can be used when the local states are generated *a priori*. We can still use Algorithm 5.1 with no changes to generate $\mathcal{S}$. If we have a logical-product-form model, then we can modify Algorithm 5.1 to deal directly with substate indices instead of substates. For instance, in [57], after the local states are generated, a matrix $\mathbf{W}$ is generated for each event e and each of the K submodels, where

$$\mathbf{W}_k^e[\mathbf{i}[k], \mathbf{j}[k]] = \begin{cases} 1 & \text{if } \mathbf{i}[k] \xrightarrow{e} \mathbf{j}[k] \\ 0 & \text{otherwise} \end{cases} \tag{5.5}$$

for each submodel index $\mathbf{i}[k], \mathbf{j}[k] \in \{0, \ldots, |\mathcal{S}_k| - 1\}$. The $\mathbf{W}$ matrices are used to compute the *Next* function using Equation 4.5, where $Next_k^e(s_k)$ is determined by the non-zero entries in row $\Psi_k(s_k)$ of matrix $\mathbf{W}_k^e$.

5.4.1 Traditional structure

Since our state space $\mathcal{S}$ now contains states represented as vectors of integer indices, we can simply use one of the traditional structures discussed earlier (hash tables, AVL trees, or splay trees) to store these vectors. The benefit of doing this instead of simply storing the states as described earlier is a reduction of storage requirements. A substate of submodel k can now be encoded in $\lceil \log_2(|\mathcal{S}_k|) \rceil$ bits, which may be substantially less than if we were to directly encode the substate using techniques described in Section 5.2.1. Of course, we must also store a copy of each substate of submodel k in a table. Typically, a given substate s_k will appear many times in states of $\mathcal{S}$, so the overhead of these extra substate tables is negligible.

This technique can also be used without prior knowledge of the local states. During

generation, we maintain tables of known substates for each submodel. Each substate encountered so far will have an index that can be used in the integer vector representation of a state. When a new substate is encountered, we simply add it to the appropriate table and assign it a unique index.

5.4.2 Bit vectors

Another way to represent the set $\mathcal{S}$ is to specify which states in $\hat{\mathcal{S}}$ are reachable. That is, we use a bit vector of size $|\hat{\mathcal{S}}|$, where the ones in the vector correspond to the states belonging to $\mathcal{S}$, and the zeroes in the vector correspond to the states in $\hat{\mathcal{S}} \setminus \mathcal{S}$. This technique is discussed thoroughly in [57] for the case of structured models. An interesting extension is discussed in [13], where equivalent portions of the bit vector are merged to reduce storage requirements.

To use this straightforward data structure, we need exact knowledge of $\hat{\mathcal{S}}$ *a priori*, since we must allocate $|\hat{\mathcal{S}}|$ bits. While this data structure can be used for models in general, it is usually reserved for structured models whose local states can be generated in isolation, so that we can use $\hat{\mathcal{S}} = \mathcal{S}_K \times \cdots \times \mathcal{S}_1$. This usually results in a smaller set $\hat{\mathcal{S}}$, and thus requires less memory.

The model in Figure 4.5 can be used to illustrate this savings. If we consider the model as a single Petri net, we have a safe net with 16 places, which has potential states $\hat{\mathcal{S}} = \{0,1\}^{16}$, yielding $2^{16} = 65,536$ potential states. If we instead consider the model as a structured model containing four submodels, where each submodel is the Petri net depicted in Figure 4.3, then we have $|\mathcal{S}_4| = |\mathcal{S}_3| = |\mathcal{S}_2| = |\mathcal{S}_1| = 6$, resulting in $6^4 = 1,296$ potential states.

We use Algorithm 5.1 to generate $\mathcal{S}$, where $\mathcal{R}$ is represented by the bit vector and $\mathcal{U}$ is represented by a linked-list stack or queue. This requires exactly $|\hat{\mathcal{S}}|$ bits to store $\mathcal{S}$, and requires $O(U \cdot \log_2(|\hat{\mathcal{S}}|))$ additional memory during exploration, where U is the maximum number of elements in the set of unexplored states $\mathcal{U}$. Using this data structure, the cost of inserting or searching for a state s in $\mathcal{R}$ is $O(1)$, once we compute the potential index $\hat{\Psi}(s)$ of the state (which has cost $O(K)$). While this technique is quite fast, it could require a significant amount of memory. The bit vector alone requires $\frac{|\hat{\mathcal{S}}|}{|\mathcal{S}|}$ bits per state. Unfortunately, it is possible for $\hat{\mathcal{S}}$ to be substantially larger than $\mathcal{S}$, in which case this fraction will be quite large.

After discovering the set $\mathcal{S}$, we can generate an array of states, where each state is represented as an integer in the range $\{0 \ldots |\hat{\mathcal{S}}| - 1\}$, which is the potential index of the state. This array requires $O(|\mathcal{S}| \cdot \log_2(|\hat{\mathcal{S}}|))$ bits of storage. Once this final representation of $\mathcal{S}$ has been created, we can destroy the bit vector.

5.4.3 Multi-level arrays

Recall that each tree in our multi-level tree structure represents a set $\mathcal{B}_k(s_K, \ldots, s_{k+1})$ according to Equation 5.2, where each node in the tree represents a substate-pointer pair. If we have prior knowledge of the local states, we can instead represent a set $\mathcal{B}_k(s_K, \ldots, s_{k+1})$ as an array of "downward" pointers, where element $\mathbf{i}[k]$ of the array is a pointer to the set $\mathcal{B}_{k-1}(s_K, \ldots, s_{k+1}, {\Psi_k}^{-1}(\mathbf{i}[k]))$. Empty sets are represented by null pointers. Level 1 sets are represented by arrays of bits: bit $\mathbf{i}[1]$ of the array representing $\mathcal{B}_1(s_K, \ldots, s_2)$ is set if and only if

$${\Psi_1}^{-1}(\mathbf{i}[1]) \in \mathcal{B}_1(s_K, \ldots, s_2).$$

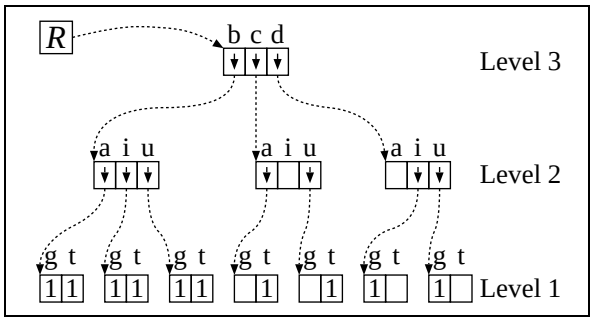


Figure 5.14: A multi-level array representing the structure in Figure 5.9

Thus, we replace each level-k tree in our multi-level tree structure with an array of size $|\mathcal{S}_k|$. We call this structure a *multi-level array.*

An example of a multi-level array depicting the sets of Figure 5.9 is shown in Figure 5.14. In the figure, null pointers and zero bits are omitted. Also, for clarity, the array elements are labeled with the substates corresponding to the element index. That is, element i of each array in level k is labeled with ${\Psi_k}^{-1}(i)$.

Since the arrays give us direct access to the elements in each set $\mathcal{B}$, the time required to search for a substate at a given level is $O(1)$. Thus, searches and insertions into a multi-level array require only $O(K)$ time. The techniques for representing $\mathcal{U}$ discussed for multi-level trees apply to multi-level arrays also: we can either store $\mathcal{U}$ as a second multi-level array or store $\mathcal{U}$ using a linked list of unexplored nodes for each array. We could also use an extra bit for each array element to indicate if it is an unexplored node or not. While quite memory efficient, this technique requires a linear search through an array at each level to find the next state to explore in $\mathcal{U}$.

We can also use multi-level arrays when we cannot generate the local states in isolation, with some additional overheads, similar to the technique discussed for traditional structures.

During generation, for each level k we must also maintain the set of known local states $\mathcal{S}_k$, which grows as the computation progresses. When we create an array at level k, we allocate space for $|\mathcal{S}_k|$ elements, according to the current set $\mathcal{S}_k$. As the set $\mathcal{S}_k$ increases in size, we must also increase the sizes of the level-k arrays. Instead of doing this all at once, we increase the sizes only as necessary. To do this, we store the array size along with each array. Whenever we access an element past the end of the array for reading (during a search), the elements are considered to be null pointers or zero bits. If we instead want to write to the array (during an insertion), we must replace the old array with a new, larger array whose size $|\mathcal{S}_k|$ is based on the most recent set $\mathcal{S}_k$.

5.5 Experimental results

This section provides a comparison of all the techniques described in this chapter. We use a variety of models as benchmarks; these are described in detail in Appendix B. Each model has an integer parameter N that affects the number of states. Depending on the model, N may represent the number of machines in the system, the number of jobs circulating in the system, or a similar parameter whose increase will drastically increase the number of system states.

Table 5.1 shows the state space sizes of the benchmark models for various choices of the model parameter N. The first column shows the name of the benchmark model and the value for N. The column labeled "Local states" shows $|\mathcal{S}_k|$, the number of local states for submodel k, for all submodels $k \in \{1, \ldots, K\}$. The column labeled "Potential states" shows $|\hat{\mathcal{S}}|$, the number of potential states, which is the product of the sizes of the local states.

Model	Size	Local states $\lvert\mathcal{S}_1\rvert, \ldots, \lvert\mathcal{S}_K\rvert$	Potential states $\lvert\hat{\mathcal{S}}\rvert$	Reachable states $\lvert\mathcal{S}\rvert$
Kanban	1	4, 4, 4, 4	256	160
	2	10, 10, 10, 10	10,000	4,600
	3	20, 20, 20, 20	160,000	58,400
	4	35, 35, 35, 35	1,500,625	454,475
	5	56, 56, 56, 56	9,834,496	2,546,432
	6	84, 84, 84, 84	49,787,136	11,261,376
	7	120, 120, 120, 120	207,360,000	41,644,800
	8	165, 165, 165, 165	741,200,625	133,865,325
	9	220, 220, 220, 220	2,342,560,000	384,392,800
	10	286, 286, 286, 286	6,690,585,616	1,005,927,208
FMS	1	6, 8, 6, 3	864	120
	2	21, 35, 20, 6	88,200	3,444
	3	56, 111, 50, 10	3,108,000	48,590
	4	125, 286, 105, 15	56,306,250	438,600
	5	246, 637, 196, 21	644,985,432	2,895,018
	6	441, 1274, 336, 28	5,285,734,272	15,126,440
	7	736, 2346, 540, 36	33,566,192,640	65,886,768
Phils	4	34, 34	1,156	322
	6	34, 34, 34	39,304	5,778
	8	34, 34, 34, 34	1,336,336	103,682
	10	34, 34, 34, 34, 34	45,435,424	1,860,498
	12	34, 34, 34, 34, 34, 34	1,544,804,416	33,385,282
Slot	2	15, 15	225	52
	3	15, 15, 15	3,375	504
	4	15, 15, 15, 15	50,625	5,136
	5	15, 15, 15, 15, 15	759,375	53,856
	6	15, 15, 15, 15, 15, 15	11,390,625	575,296
	7	15, 15, 15, 15, 15, 15, 15	170,859,375	6,225,792

Table 5.1: State space sizes for benchmark models

Finally, the column labeled "Reachable states" shows $|\mathcal{S}|$, the number of states reachable from the initial state of the model. Notice that, for these models, the number of reachable states is much smaller than the number of potential states; in some cases this difference is several orders of magnitude.

Table 5.2 shows the amount of memory required to store $\mathcal{S}$ using a hash table or a

Model	Size	Memory required in bytes States	Hash table	Binary tree
Kanban	1	640	1,484	1,920
	2	29,320	46,188	55,200
	3	436,064	678,508	700,800
	4	3,553,312	5,377,384	5,453,700
	5	20,199,040	24,423,876	30,557,184
FMS	1	1,560	1,324	1,440
	2	44,769	27,644	41,328
	3	631,670	416,796	583,080
	4	5,701,800	3,534,116	5,263,200
	5	37,635,234	25,818,220	34,740,216
Phils	4	3,864	3,012	3,864
	6	104,004	50,900	69,336
	8	2,488,368	859,636	1,244,184
	10	55,814,940	21,680,140	22,325,976
Slot	2	208	612	624
	3	6,048	5,468	6,048
	4	82,176	48,332	61,632
	5	1,077,120	437,860	646,272
	6	13,807,104	5,860,668	6,903,552

Table 5.2: Memory usage for traditional techniques

binary tree. For these data structures, we must store each reachable state explicitly; the memory required to do so is listed in the column labeled "States". During generation of $\mathcal{S}$, we must also store the hash table or binary tree, whose memory requirements are shown in the appropriate column. For our implementation, both hash tables and binary trees store pointers to the states. Thus the memory required for generation is the *sum* of the "States" column and the appropriate column for our choice of data structure. We use a hash table with chaining whose size is doubled when the table becomes sufficiently filled. The "Hash table" column is computed by summing the size of an array of M pointers (where M is the hash table size) and two pointers per object stored in the hash table. The "Binary

Model	Size	Memory required in bytes				
		Locals	Multi-tree	1-byte M-tree	Bit vector	Multi-array
Kanban	1	16	3,328	6,352	44	565
	2	64	82,560	69,848	1,262	6,930
	3	144	754,800	623,888	20,012	52,911
	4	264	5,684,740	3,597,256	187,591	309,798
	5	432	31,336,048	15,474,432	1,229,324	1,399,735
	6	656	137,396,032	62,727,712	6,223,404	5,350,399
	7	944	—	—	25,920,012	17,715,263
	8	1,304	—	—	92,650,091	52,947,209
	9	1,744	—	—	—	140,635,036
	10	2,272	—	—	—	350,755,020
FMS	1	46	2,576	6,280	120	847
	2	238	52,928	86,216	11,037	15,467
	3	811	671,704	718,368	388,512	173,137
	4	2,106	5,771,648	4,289,368	7,038,294	1,331,204
	5	4,630	37,075,528	28,948,568	80,623,191	7,874,554
	6	9,076	190,521,328	113,224,840	—	36,864,866
	7	16,353	—	—	—	147,849,208
Phils	4	292	4,456	5,040	157	455
	6	438	79,704	90,304	4,925	10,705
	8	584	1,429,720	1,619,376	167,054	194,619
	10	730	25,654,232	29,056,512	5,679,440	3,494,821
	12	876	—	—	193,100,564	62,714,543
Slot	2	60	912	2,208	41	130
	3	90	8,960	23,640	434	1,760
	4	120	93,184	251,016	6,341	23,638
	5	150	985,760	2,692,728	94,934	289,594
	6	180	10,640,304	29,182,752	1,423,841	3,431,024
	7	210	116,059,088	318,963,264	21,357,434	39,947,520

Table 5.3: Memory usage for structured techniques

tree" column is computed by summing the size of each tree node; in our case this is exactly $12 \cdot |\mathcal{S}|$. After generation, the hash table or binary tree can be destroyed, and the memory required for $\mathcal{S}$ is that given by the "States" column, plus an array of indices (unless we sort the states).

Table 5.3 shows the memory requirements for structured techniques. For these data

structures, explicit storage for each state is not required. Instead, we must store $\mathcal{S}_k$, the substates reached in submodel k, for each submodel $k \in \{1, \ldots, K\}$. The total memory required to store these substates is shown in the column labeled "Locals". The remaining columns in the table are as follows.

Multi-tree: The maximum memory required for a multi-level tree. We use distinct trees for $\mathcal{R}$ and $\mathcal{U}$, so the maximum memory required is greater than the final memory required. Tree nodes use four-byte pointers. Note that the multi-level tree requires slightly more memory than a binary tree, but since we store the local substates instead of the states, the total memory requirements are much less.

1-byte M-tree: A multi-level tree that uses one-byte integers instead of pointers for bottom level trees. We use a single tree for $\mathcal{R}$ and $\mathcal{U}$. The memory reported is for AVL trees; if splay trees are used an additional byte is saved for each bottom-level node. Note that there is some overhead associated with this data structure: for small $\mathcal{S}$ this structure actually requires more memory than a multi-level tree that uses four-byte pointers.

Bit vector: The memory required to store a vector of bits of size $|\hat{\mathcal{S}}|$. This approach also requires a data structure to store $\mathcal{U}$; the memory required by this structure is not reported here.

Multi-array: The memory needed for a single multi-level array. To represent unexplored states, we use an extra bit for each array element. These extra bits are included in the reported memory usage.

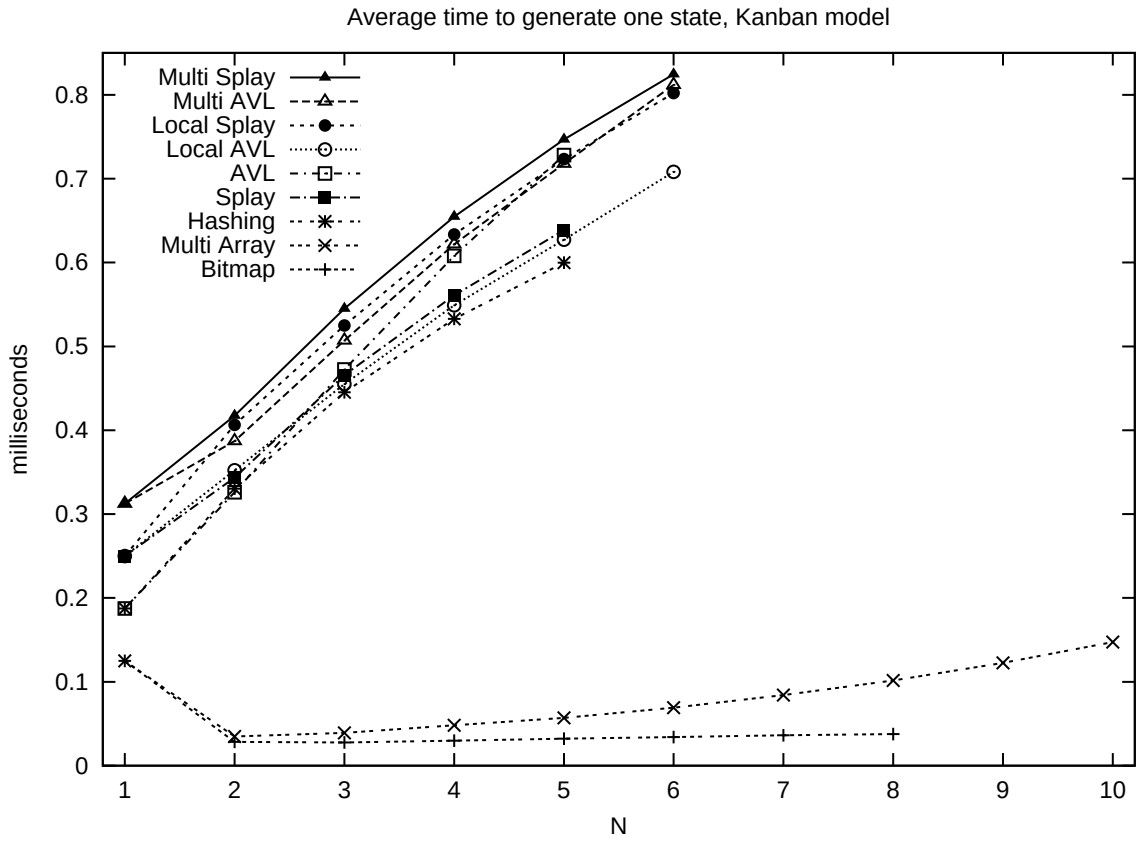


Figure 5.15: Traditional generation times for Kanban

Note that the missing data in some of the columns is due to excessive memory requirements of that approach for the given model. In particular, we could compute the memory requirements for the bit vector column as $\frac{1}{8} \cdot |\hat{\mathcal{S}}|$, but instead we intentionally leave the portions of the column blank to stress that the approach could not be used on our machine.

Timing results are shown in Figure 5.15, Figure 5.16, Figure 5.17, and Figure 5.18. The figures show the average time to generate a single state, in milliseconds. This is computed by dividing the total generation time by the number of reachable states. The times reported are for executions on a 400-MHz Pentium II machine running the Linux operating system. We compare the following techniques.

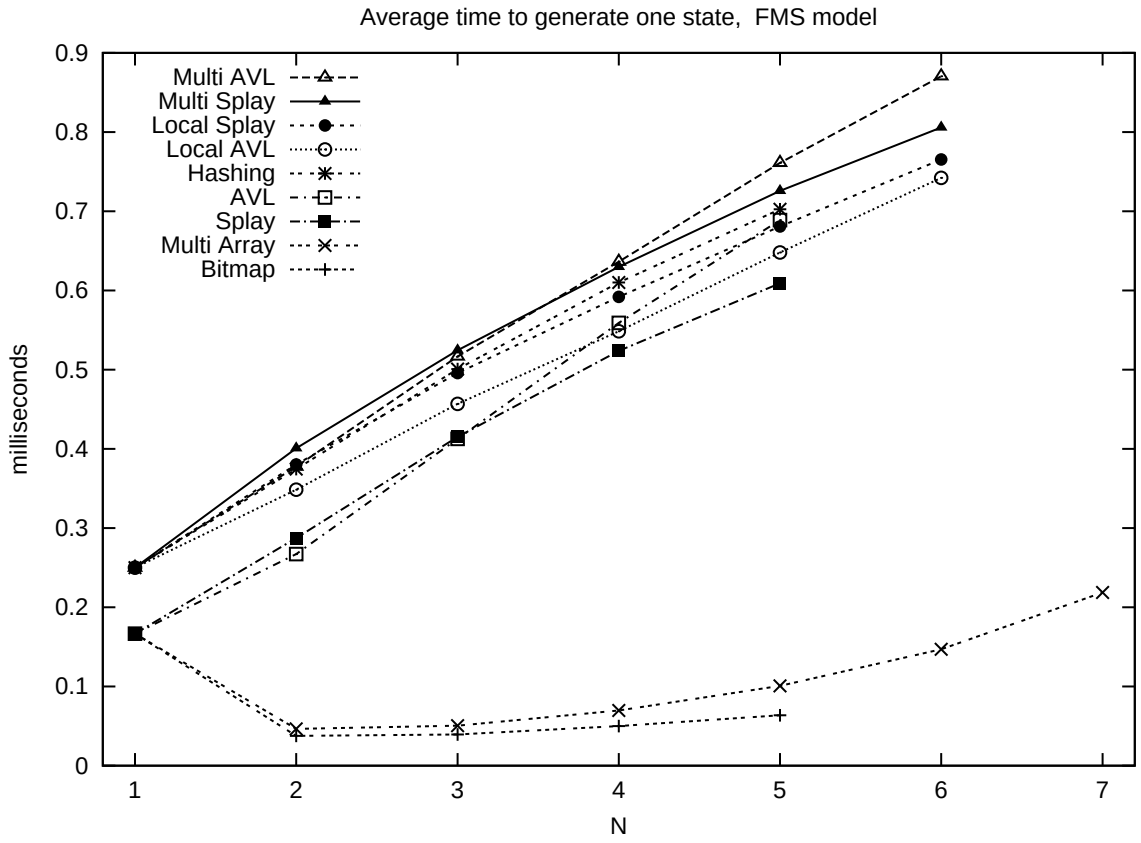


Figure 5.16: Traditional generation times for FMS

Multi Splay: A multi-level tree based on splay trees.

Multi AVL: A multi-level tree based on AVL trees.

Local Splay: A multi-level tree based on splay trees that utilizes event locality.

Local AVL: A multi-level tree based on AVL trees that utilizes event locality.

AVL: A traditional AVL tree.

Splay: A traditional splay tree.

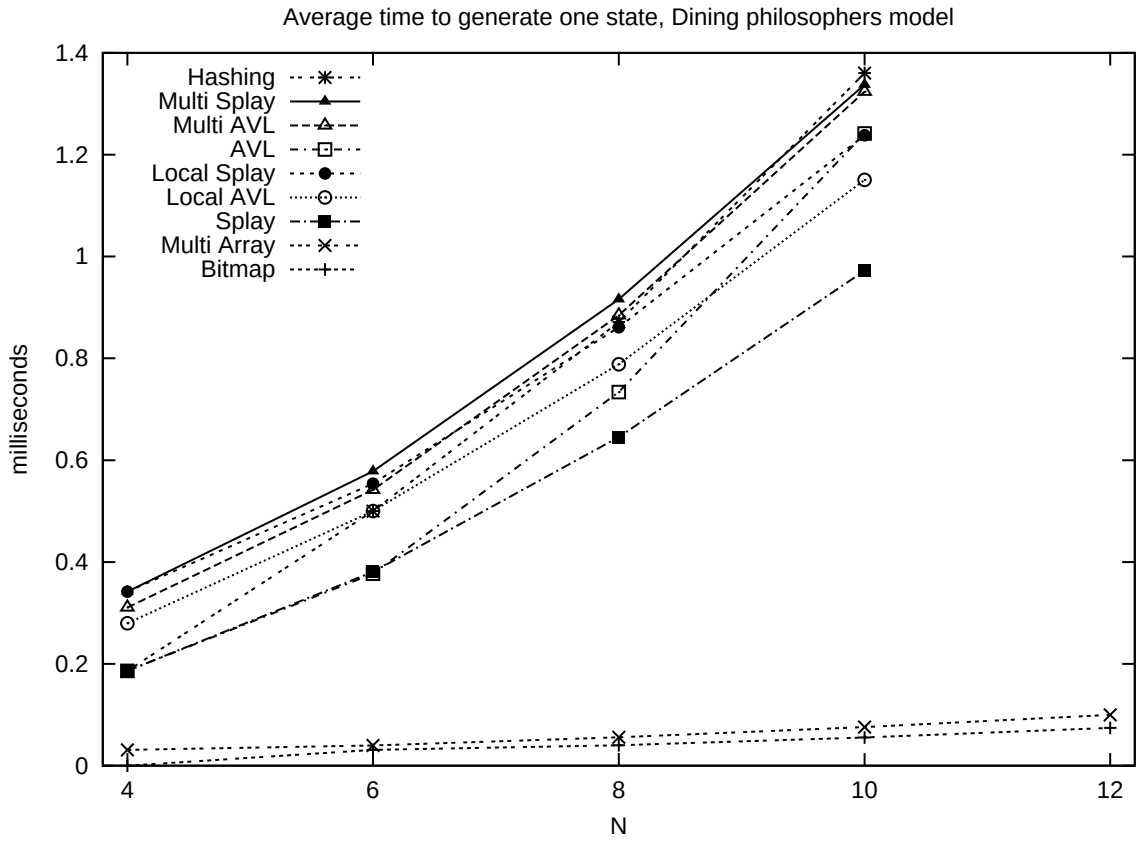


Figure 5.17: Traditional generation times for Dining Philosophers

Hashing: A hash table that doubles in size when the number of entries in the table becomes too large. We use a universal hashing function that simply treats the state, an array of state variables that are bounded by 256, as a large base-256 integer n; the location of the state in the table is then given by $n \bmod M$, where M is a prime number corresponding to the hash table size. Note that, since M is prime, $n \bmod M$ will depend on every state variable.

Multi Array: A multi-level array. Local states are generated *a priori* in isolation to provide a fair comparison with the bit vector approach. Note that a linear search is

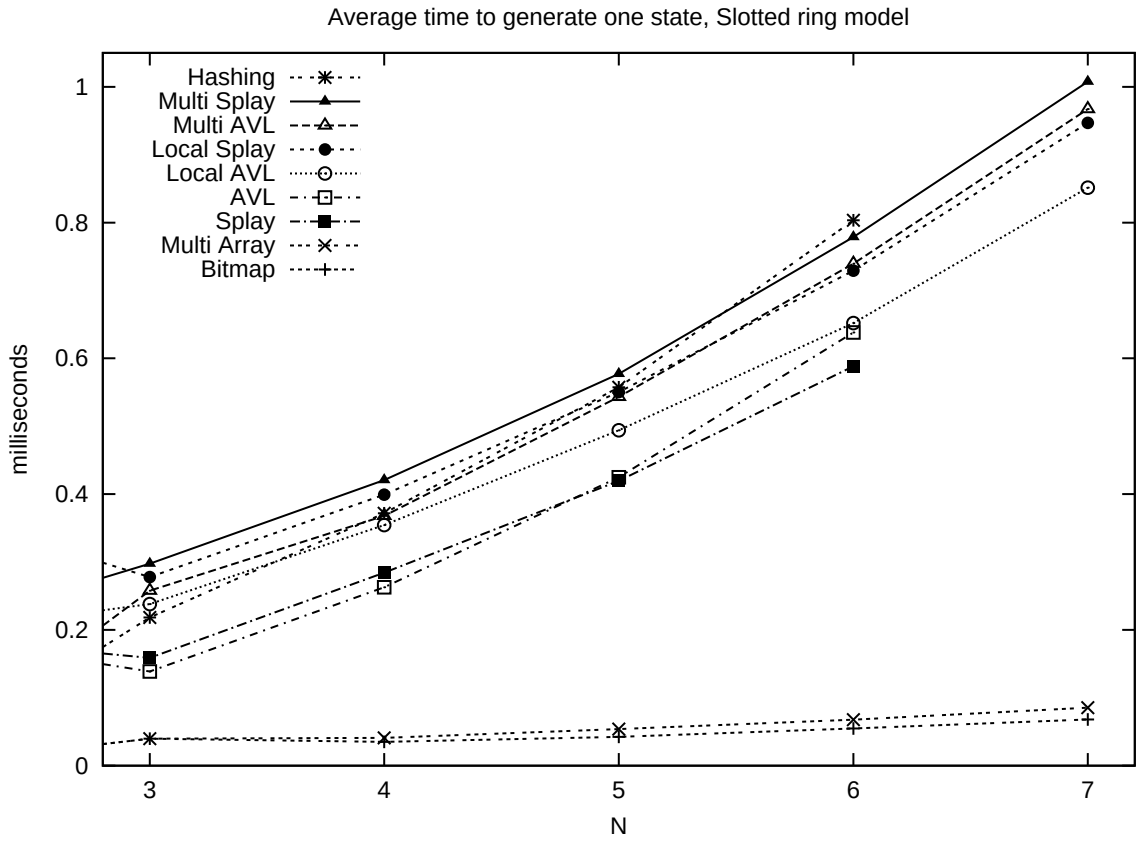


Figure 5.18: Traditional generation times for Slotted Ring

required to determine the next unexplored state; by eliminating the linear search we can save CPU time at the expense of additional storage requirements.

Bitmap: A vector of bits. Local states are generated *a priori*; this is included in the total CPU time required to generate $\mathcal{S}$.

From the figures, we notice several trends.

- The structures that require substantial searches (hashing and tree-based approaches) require roughly the same amount of time, which is about an order of magnitude greater than the approaches with near constant time searches (vectors of bits or multi-level

arrays).

- Exploiting event locality can significantly reduce CPU requirements.
- Splay trees perform better than AVL trees when the trees contain a large number of nodes; otherwise, AVL trees perform better. Since the tree sizes for each level in the multi-level tree structure are quite small, multi-level AVL performs better than multi-level splay.
- Hashing is a bit unpredictable: in some cases it performs quite well, in others it does not. In particular, when the size of a state is large (in the dining philosophers or slotted ring models, for instance) the cost of computing our hash function becomes high.

5.6 Conclusion

Overall, we see that structured approaches are much more memory-efficient than traditional approaches due to the requirement of traditional approaches to explicitly represent each state. When a multi-level structure is used, arrays are preferred over binary trees due to both reduced memory requirements and faster search and insertion operations. Exploiting the locality of an event during state space generation can reduce CPU time with almost no cost in terms of memory.

Taking into consideration the memory and CPU requirements of each of the data structures, it is clear that both the multi-level approach using arrays and the vector of bits approach are promising. However, when the number of potential states is substantially larger than the number of actual states, using a vector of bits is extremely expensive.

Given the greater flexibility and lower memory requirements, we feel that the multi-level approach is the stronger of the two and deserves additional study.

Chapter 6

Symbolic State Space Generation

In this chapter, we investigate algorithms for generating the set of reachable states $\mathcal{S}$ *symbolically*. That is, instead of storing and exploring each state explicitly, we use efficient data structures and manipulations on *sets* of states. Techniques of this sort come at a price, usually a loss of modeling flexibility. Early work in this area by Clarke and others [17, 32] required that models be specified using logic statements. Properties to be verified were also specified as logic statements. Similar techniques were later applied to state space generation for a restricted class of Petri net models by Pastor's group [78, 79, 80, 81].

Our contributions in this area [68] are the following. We first develop an efficient encoding scheme, based on a combination of our multi-level structure from the previous chapter and decision diagrams. Then, using our new encoding scheme, we develop manipulation algorithms that are specifically designed for state space generation. The algorithms can be applied to any *logical-product-form* model. The concept of event locality discussed in the previous chapter is exploited by these routines as well. The result is a technique that can be used to generate and store enormous sets extremely efficiently in terms of both memory and execution time.

The remainder of the chapter is organized as follows. Section 6.1 gives a brief overview

of decision diagrams as they relate to our work. Further information on this interesting topic can be found in [10, 11, 43, 54, 63, 95]. Section 6.2 gives a summary of the technique used by Pastor. Section 6.3 presents our encoding scheme and specialized manipulation algorithms. Section 6.4 describes how to perform logical queries on $\mathcal{S}$ once it has been generated. Section 6.5 gives experimental results in which our technique is compared to other techniques. Finally, Section 6.6 contains concluding remarks.

6.1 Decision diagrams

Multi-valued decision diagrams [54, 95] (MDDs) are directed, acyclic graphs used to represent K-variable integer functions of the form

$$f : \{0, \ldots, N_K - 1\} \times \cdots \times \{0, \ldots, N_1 - 1\} \rightarrow \{0, \ldots, M - 1\}.$$

Nodes in the MDD are either *terminal* or *non-terminal*. The terminal nodes correspond to the "return values" of the function and are labeled with an integer $0, \ldots, M - 1$. Non-terminal nodes are labeled with a variable x_k, and contain N_k pointers to other nodes. These pointers correspond to the *cofactors* of f, where a cofactor is defined as

$$f_{x_k=c} \equiv f(x_K, \ldots, x_{k+1}, c, x_{k-1}, \ldots, x_1)$$

for variable x_k and constant c. A non-terminal node representing function f is then written as the $(N_k + 1)$-tuple $(x_k, f_{x_k=0}, \ldots, f_{x_k=N_k-1})$. Since a terminal node labeled with m represents the integer constant m, every MDD node represents some integer function.

The paths in an *ordered* MDD (OMDD) visit non-terminal nodes according to some total ordering on the variables $x_K, \ldots, x_1$. That is, if a path through the OMDD visits

non-terminal nodes labeled with x_i and x_j, then x_i is visited before x_j if and only if $i > j$. We say a non-terminal OMDD node labeled with x_k is a *level-k* node, and a terminal node is a level-0 node. Note that the ordering property implies that all downward pointers from level-k nodes are to level-j nodes, where $j < k$.

A *reduced* OMDD (ROMDD) has the following additional properties.

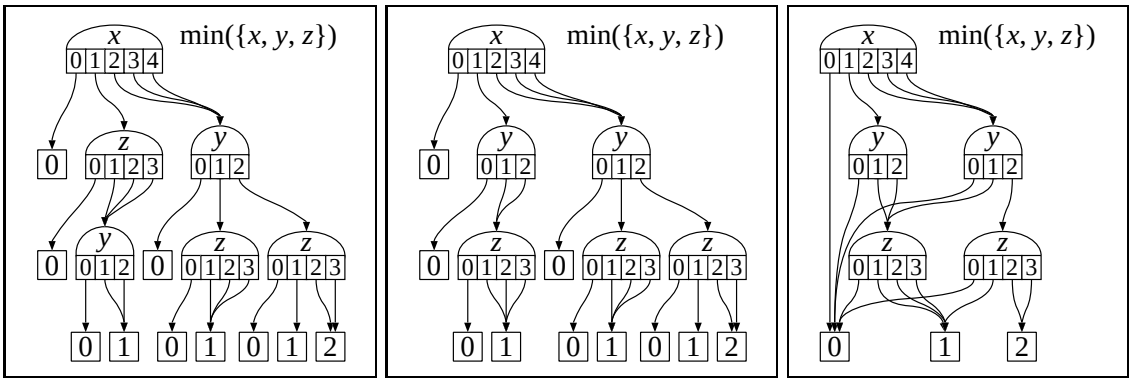
1. There are no duplicate terminal nodes. That is, at most one terminal node is labeled with integer m.

2. There are no duplicate non-terminal nodes. That is, given two non-terminal nodes $(x_i, f_{x_i=0}, \ldots, f_{x_i=N_i-1})$ and $(x_j, g_{x_j=0}, \ldots, g_{x_j=N_j-1})$, we must have either $x_i \neq x_j$ or $f_{x_i=n} \neq g_{x_i=n}$ for some $n \in \{0, \ldots, N_i - 1\}$.

3. All non-terminal nodes depend on the value of their variable. That is, given a non-terminal node $(x_k, f_{x_k=0}, \ldots, f_{x_k=N_k-1})$, we must have $f_{x_k=i} \neq f_{x_k=j}$ for some $i, j \in \{0, \ldots, N_k - 1\}$.

It has been shown [95] that ROMDDs are a canonical structure: given any integer function and a variable ordering, there is exactly one ROMDD representation for that function. ROMDDs are similar to the "shared trees" found in [106]. It has also been shown that ROMDDs can be extremely sensitive to the variable ordering. Bryant [10] showed that the logic function of $2n$ variables $x_1 \cdot x_2 + \cdots + x_{2n-1} \cdot x_{2n}$ produces a graph with $2n + 2$ nodes, while the function $x_1 \cdot x_{n+1} + \cdots + x_n \cdot x_{2n}$ (which differs only in the variable ordering) produces a graph with 2^{n+1} nodes.

Figure 6.1 shows example MDDs for the integer function $\min(\{x, y, z\})$, where the variable bounds are $x \in \{0, \ldots, 4\}$, $y \in \{0, \ldots, 2\}$, and $z \in \{0, \ldots, 3\}$. Figure 6.1(a)



(a) MDD (b) OMDD (c) ROMDD

Figure 6.1: MDD representations of $\min(\{x, y, z\})$

shows an MDD that is neither ordered nor reduced; Figure 6.1(b) shows an MDD ordered by (x, y, z); Figure 6.1(c) shows *the* reduced ordered MDD representation for the variable ordering (x, y, z).

Much work has been done with the special case of MDDs applied to K-variable logic functions, which have the form $f : \{0, 1\}^K \to \{0, 1\}$. An MDD representing such a function is called a *binary* decision diagram (BDD) [10, 11, 63], since each non-terminal node has exactly two outgoing arcs. Many other forms of decision diagrams have been proposed [43], but for our work, MDDs and BDDs are the most relevant. For the remainder of this chapter, we use "MDD" and "BDD" to mean "ROMDD" and "ROBDD", respectively.

6.1.1 Manipulating MDDs

To obtain full benefit from using decision diagrams, we must be able to manipulate them efficiently. One such manipulation is the Case operator, which is defined in [95] as

$$\mathsf{Case}(F, G^0, \ldots, G^{m-1})(x_K, \ldots, x_1) = G^i(x_K, \ldots, x_1) \quad \text{if } F(x_K, \ldots, x_1) = i$$

where the range of F is $\{0, \ldots, m-1\}$. In other words, Case selects the appropriate G function based on the returned value of function F. Algorithm 6.1 shows how to compute Case recursively, based on the relation

$$\mathsf{Case}(F, G^0, \ldots, G^{m-1})_{x=i} = \mathsf{Case}(F_{x=i}, G^0_{x=i}, \ldots, G^{m-1}_{x=i}). \tag{6.1}$$

Note that, in the algorithm, all functions $G^0, \ldots, G^{m-1}$ and F are represented as MDDs and Case returns an MDD. The terminal cases of Equation 6.1 are when

1. F is a constant (Case returns the appropriate G^i)
2. $G^i = G$ for all i (Case returns G)
3. $G^i = i$ for all i (Case returns F)

which are checked in lines 1-6 of Algorithm 6.1.

A few important aspects of an implementation of Algorithm 6.1 are worth mentioning. Since Case operates on ROMDDs, we must be sure that the MDD returned by Case is both ordered and reduced. We assume a fixed variable ordering $x_K, \ldots, x_1$ for the input MDDs to Case. Line 10 of the algorithm sets x_k to the "top" variable of the input MDDs. That is, of all the nodes in the MDDs representing the functions $F, G^0, \ldots, G^{m-1}$, the node that is visited first according to the variable ordering has label x_k. Since the MDDs for functions $F, G^0, \ldots, G^{m-1}$ are ordered, we can quickly determine x_k by taking the maximum variable label of the first nodes of these MDDs. By setting x_k to the top variable, we guarantee that the recursive calls to Case will process nodes according to the variable ordering.

MDD reduction is usually achieved by representing all MDDs in a single graph [11]; this implies that two MDDs represent the same function if and only if they are represented by

```
Case(F, G^0, ..., G^{m-1})
 1: if F is a constant r then                          • Check terminal cases
 2:    return G^r
 3: else if G^0 = G^1 = ··· = G^{m-1} then
 4:    return G^0
 5: else if G^0 = 0 ∧ ··· ∧ G^{m-1} = m − 1 then
 6:    return F
 7: else if cache contains entry for (F, G^0, ..., G^{m-1}) then
 8:    return cache entry result;
 9: end if
10: let x_k be the top variable of F, G^0, ..., G^{m-1}   • Not a terminal case
11: for i ← 0 to N_k − 1 do
12:    H^i ← Case(F_{x_k=i}, G^0_{x_k=i}, ..., G^{m-1}_{x_k=i})   • Apply Equation 6.1
13: end for
14: if H^0 = H^1 = ··· = H^{N_k−1} then
15:    R ← H^0                                        • Reduction rule 3.
16: else
17:    R ← UniqueTableInsert(x_k, H^0, ..., H^{N_k−1})   • Reduction rule 2.
18: end if
19: add [(F, G^0, ..., G^{m-1}), R] to cache
20: return R
```

Algorithm 6.1: The Case operator on MDDs

the same node in the graph. Assuming the input MDDs to Case are reduced, we only need to make sure that the node returned by Case satisfies the reduction rules. Lines 14-17 of Algorithm 6.1 handle the reduction rules for non-terminal nodes. Lines 14-15 ensure that no non-terminal node is created that violates rule 3: all non-terminal nodes must depend on their variable. Lines 16-17 check rule 2: all non-terminal nodes are unique. This is done with a call to UniqueTableInsert, which looks for node $(x_k, H^0, \ldots, H^{N_k-1})$ in a "uniqueness" table of all the nodes in the graph. If the node is found, then it is returned; otherwise no such node exists in the graph and one is created.

Another important feature of Algorithm 6.1 is the *cache*. This is not a cache in the traditional sense, but rather a table of previously-computed results. Whenever an MDD

node is computed using Case, the result is saved in a cache (line 19). Before proceeding with a Case computation (assuming it is not a terminal case), the cache is checked to see if the computation has already been performed (line 7). If so, the previous result is returned immediately (line 8).

To fully appreciate the effect of using a cache, we must examine the complexity of Algorithm 6.1. Without using a cache, a call to Case will generate exactly N_k recursive calls to Case when the top node is labeled with x_k. The cost C_k of a call to Case with top variable x_k is then

$$C_k = U + N_k \cdot C_{k-1}$$

where U is the complexity of a call to UniqueTableInsert, and the terminating case of the recurrence is $C_1 = U + N_1$. The worst case complexity of a call to Case is then $O(\prod_{k=1}^{K} N_k)$, assuming constant time complexity for UniqueTableInsert. When a cache is used, it can be shown (see Appendix C) that the worst-case complexity is bounded by $O(|F| \cdot |G^0| \cdot \cdots \cdot |G^{m-1}| \cdot \max(\{N_K, \ldots, N_1\}))$, where $|F|$ denotes the number of MDD nodes required to represent function F. Thus, the cache reduces the maximum number of Case computations from the number of possible variable assignments to the number of possible distinct calls to Case, which is the product of the sizes of the MDDs representing the input functions. Both the "uniqueness" table and cache are usually implemented with hash tables, and are commonly used for MDD manipulation algorithms [10, 11].

An example showing the computation of the MDD representing the function

$$f(x, y, z) \equiv (\min(\{x, y, z\}) = z) \equiv \mathsf{Case}(\min(\{x, y\}), z \leq 0, z \leq 1, z \leq 2)$$

MDD node for $\mathsf{Case}(\min(\{x,y\}), z \leq 0, z \leq 1, z \leq 2)$:

$\mathsf{Case}(\min(\{x,y\}), z \leq 0, z \leq 1, z \leq 2)_{x=0} \equiv \mathsf{Case}(0, z \leq 0, z \leq 1, z \leq 2) \equiv z \leq 0$
$\mathsf{Case}(\min(\{x,y\}), z \leq 0, z \leq 1, z \leq 2)_{x=1} \equiv \mathsf{Case}(\min(\{1,y\}), z \leq 0, z \leq 1, z \leq 2)$
$\mathsf{Case}(\min(\{x,y\}), z \leq 0, z \leq 1, z \leq 2)_{x=2} \equiv$
$\mathsf{Case}(\min(\{x,y\}), z \leq 0, z \leq 1, z \leq 2)_{x=3} \equiv$
$\mathsf{Case}(\min(\{x,y\}), z \leq 0, z \leq 1, z \leq 2)_{x=4} \equiv \mathsf{Case}(y, z \leq 0, z \leq 1, z \leq 2)$

MDD node for $\mathsf{Case}(\min(\{1,y\}), z \leq 0, z \leq 1, z \leq 2)$:

$\mathsf{Case}(\min(\{1,y\}), z \leq 0, z \leq 1, z \leq 2)_{y=0} \equiv \mathsf{Case}(0, z \leq 0, z \leq 1, z \leq 2) \equiv z \leq 0$
$\mathsf{Case}(\min(\{1,y\}), z \leq 0, z \leq 1, z \leq 2)_{y=1} \equiv$
$\mathsf{Case}(\min(\{1,y\}), z \leq 0, z \leq 1, z \leq 2)_{y=2} \equiv \mathsf{Case}(1, z \leq 0, z \leq 1, z \leq 2) \equiv z \leq 1$

MDD node for $\mathsf{Case}(y, z \leq 0, z \leq 1, z \leq 2)$:

$\mathsf{Case}(y, z \leq 0, z \leq 1, z \leq 2)_{y=0} \equiv z \leq 0$
$\mathsf{Case}(y, z \leq 0, z \leq 1, z \leq 2)_{y=1} \equiv z \leq 1$
$\mathsf{Case}(y, z \leq 0, z \leq 1, z \leq 2)_{y=2} \equiv z \leq 2$

Figure 6.2: Computing $f = \mathsf{Case}(\min(\{x,y\}), z \leq 0, z \leq 1, z \leq 2)$

is shown in Figure 6.2. Note that x, y and z are *integer* variables, and f is a *boolean* function. The bounds for variables x, y and z are as before: $x \in \{0, \ldots, 4\}$, $y \in \{0, \ldots, 2\}$, and $z \in \{0, \ldots, 3\}$. The function f can assume values 0 or 1. Illustrations of the MDDs used in the computation are shown in Figure 6.3. In the figure, many of the non-terminal nodes are annotated with the function they represent. The input MDDs are shown in Figure 6.3(a) and Figure 6.3(b), and the resulting MDD is shown in Figure 6.3(c).

Note that the computation in Figure 6.2 has duplications: the cofactors $x = 2, x = 3$ and $x = 4$ for the first node and $y = 1$ and $y = 2$ for the second node. These duplications are due to the duplicate downward pointers of the MDD in Figure 6.3. In Algorithm 6.1, these duplications will result in cache "hits"; thus the computation is performed only once.

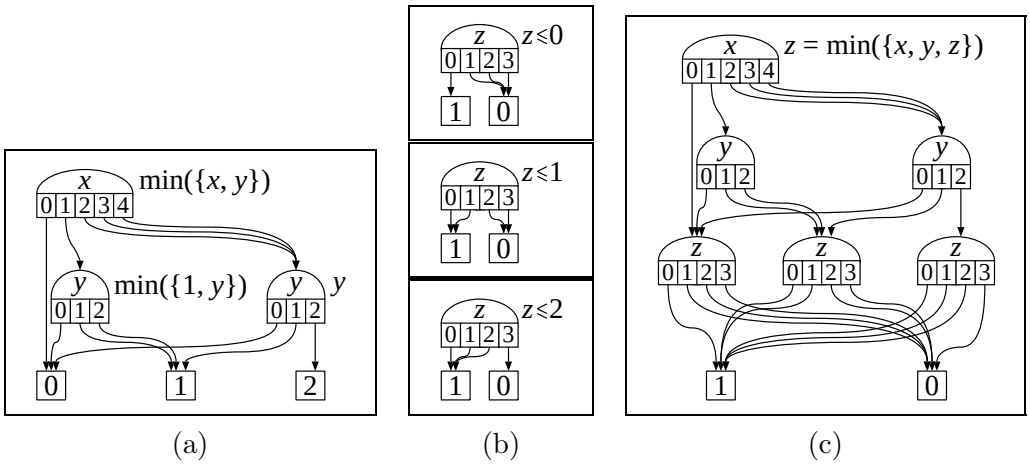


Figure 6.3: MDDs for the Case computation of Figure 6.2

6.2 Generating $\mathcal{S}$ with BDDs

Much work has been done on techniques for using BDDs to generate $\mathcal{S}$ for Petri nets by Pastor et al. [78, 79, 80, 81]. In [80], the authors describe a technique for generating $\mathcal{S}$ for *safe* Petri nets. In this case, each place can contain either zero or one tokens; thus a boolean variable x_p can be used to encode the number of tokens in place p. A BDD can be used to represent the characteristic function of $\mathcal{S}$:

$$\chi_{\mathcal{S}}(x_{p_K}, \ldots, x_{p_1}) = 1 \quad \Leftrightarrow \quad [x_{p_K}, \ldots, x_{p_1}] \in \mathcal{S}$$

where the places of the Petri net are $\mathcal{P} = \{p_K, \ldots, p_1\}$.

An example showing this type of BDD encoding for $\mathcal{S}$ is depicted in Figure 6.4. In the figure, p_k is used as both a place of the Petri net and the boolean variable representing the number of tokens in that place. More sophisticated BDD encodings are discussed in [78, 79, 81], where the authors use invariants to reduce the number of required variables.

Once we have determined how to encode a set of states with a BDD, the reachability set $\mathcal{S}$

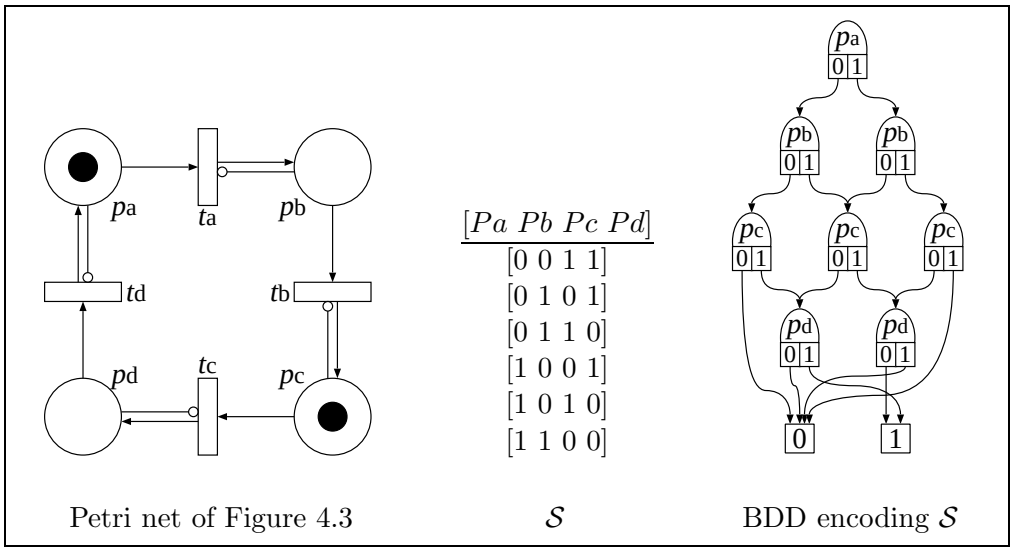


Figure 6.4: A BDD encoding reachable markings for a simple Petri net

can be generated by manipulating BDDs. For instance, set operations can be performed using the Case operator on the characteristic functions of the sets:

$$\chi_{\mathcal{A}\cup\mathcal{B}} = \mathsf{Case}(\chi_{\mathcal{A}}, \chi_{\mathcal{B}}, 1), \qquad \chi_{\mathcal{A}\cap\mathcal{B}} = \mathsf{Case}(\chi_{\mathcal{A}}, 0, \chi_{\mathcal{B}}), \qquad \chi_{\mathcal{A}\setminus\mathcal{B}} = \mathsf{Case}(\chi_{\mathcal{B}}, \chi_{\mathcal{A}}, 0). \quad (6.2)$$

Given a set of markings $\mathcal{X}$ encoded as a BDD, the authors in [80] show how to compute $\Delta(\mathcal{X})$, the set of markings reached from $\mathcal{X}$ by firing a single transition. This enables us to generate $\mathcal{S}$ using BDDs as shown in Algorithm 6.2, which is a simplified version of the algorithm in [80]. Note that iteration n of Algorithm 6.2 adds the markings that are n-step reachable from the initial marking. In the algorithm, sets $\mathcal{O}, \mathcal{S}$, and $\Delta(\mathcal{S})$ are encoded as BDDs and the computations for Δ and set union are implemented as binary BDD operators. Note that the choice of variable encoding will affect Δ, $\mathcal{O}$, and $\mathcal{S}$. The simple encoding for safe Petri nets produces a simple Δ; other encodings [78, 79] produce more complex Δ functions. In particular, if the Petri net is not safe, the number of tokens in each place

```
BDDexplore(Safe Petri Net M, marking m°)
1: O ← ∅                          • O: old reachability set
2: S ← {m°}                       • S: current reachability set
3: while S ≠ O do
4:     O ← S
5:     S ← Δ(S) ∪ S
6: end while
7: return S
```

Algorithm 6.2: Generating $\mathcal{S}$ using BDDs

must be encoded with multiple boolean variables. If place p is N-bounded, the number of tokens in p can be encoded using either a *one-hot* encoding with N variables, where a boolean variable x_p^n is set if the number of tokens in place p is exactly n, or by using a *binary* encoding with $\lceil \log_2(N+1) \rceil$ variables. The *one-hot* encoding requires more boolean variables, resulting in MDDs for $\mathcal{O}$ and $\mathcal{S}$ with greater numbers of nodes, and increased computational requirements. On the other hand, the *binary* encoding produces a more complex Δ function which may require more BDD operations, so it is not necessarily a better choice.

It is important to note that the number of iterations required by Algorithm 6.2 is bounded by the *sequential depth* of the Petri net. That is, the number of iterations is at most the maximum number of firings required to reach a marking from the initial marking, a quantity bounded by the diameter of the reachability graph. Thus, while each iteration can require substantial computation, the number of iterations is usually quite small. In contrast, the number of iterations required by an explicit search such as Algorithm 5.1 is $O(|\mathcal{S}|)$.

6.3 Generating $\mathcal{S}$ with MDDs

In our work [68], we combine ideas from the multi-level structure in the previous chapter and the BDD generation technique described in the previous section. An MDD can be used to store the reachable states of any structured model. If the structured model is composed of K submodels with local state spaces $\mathcal{S}_K, \ldots, \mathcal{S}_1$, then we will use an MDD to store the characteristic function

$$\chi_{\mathcal{S}}(\Psi_K(s_K), \ldots, \Psi_1(s_1)) = 1 \quad \Leftrightarrow \quad (s_K, \ldots, s_1) \in \mathcal{S}$$

where $\Psi_k : \mathcal{S}_k \rightarrow \{0, \ldots, |\mathcal{S}_k| - 1\}$ is an indexing function for the local states of submodel k. Thus, from now on, our MDDs have exactly $M = 2$ terminal nodes, labeled 0 and 1. Note that variable k of the MDD can assume $|\mathcal{S}_k|$ possible values. If our model is a safe Petri net, and each submodel consists of a single place, then this encoding is equivalent to the simple BDD encoding described in [80].

Figure 6.5 shows how the example state space from Figure 5.9, Figure 5.10, and Figure 5.14 is represented by an MDD using our encoding. In the figure, pointers to the terminal node 0 are omitted for clarity, and the variable labels are omitted since all level-k nodes have the same variable label. The node pointers are annotated with the local states they represent to improve readability. Similar to the multi-level structures in the previous chapter, a state is reachable if the path corresponding to that state reaches the terminal node 1. For instance, the state (cat) is reachable, because following the downward pointers for $x_3 = c, x_2 = a, x_1 = t$ leads to terminal node 1. State (big) is also reachable. Following the downward pointer for $x_3 = b$, we find that we are already at terminal node 1. This

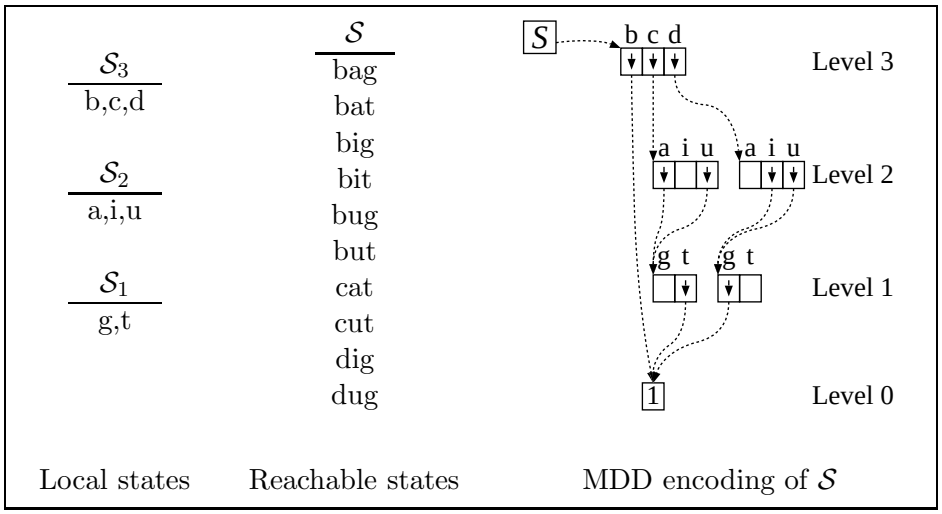


Figure 6.5: The MDD equivalent of Figure 5.14

implies that any state $(b\bullet\bullet)$ is reachable. In general, when a downward pointer skips over level k, that means that submodel k can be in any possible local state. The extreme case of this occurs when $\mathcal{S} = \hat{\mathcal{S}}$, implying that any submodel can be in any local state; thus we have $\chi_{\mathcal{S}} = 1$ and so the MDD contains only the terminal node 1.

A fundamental difference between an MDD encoding of $\mathcal{S}$ and a multi-level structure encoding $\mathcal{S}$ from the previous chapter is the correspondence between states and paths. With both structures, a state corresponds to a unique path from the "root" node encoding $\mathcal{S}$ to some bottom-level node. However, in the case of a multi-level structure, a path corresponds to exactly one state. This is not necessarily true in the case of MDDs, as can be seen in Figure 6.5: the path from the level-3 node to terminal node 1 through $x_3 = b$ corresponds to states of the form $(b\bullet\bullet)$. Thus, we can still talk about *the* path corresponding to a given state, but there may be more than one state corresponding to a given path.

To use an algorithm similar to Algorithm 6.2 for generating $\mathcal{S}$, we must be able to

compute the Δ function for this encoding technique. That is, we must be able to manipulate the MDD representation of $\mathcal{S}$ to simulate the occurrence of events in the model. However, unlike the approach of Pastor et al., we develop specialized MDD manipulations for handling the occurrence of events. The approach we use requires a *logical-product-form* model. This allows us to determine the states that enable an event and the new states reached after an event occurs by looking at MDD nodes in isolation. We first show how to do this for local events, then for synchronizing events.

6.3.1 Occurrence of local events

An event e that is a local event of submodel k has the special property of affecting submodel k only. This means that the next state function for e can be written

$$Next^e((s_K, \ldots, s_1)) = \{s_K\} \times \cdots \times \{s_{k+1}\} \times Next_k^e(s_k) \times \{s_k - 1\} \times \cdots \times \{s_1\}$$

for function $Next_k^e$ applied to submodel k only. Thus for any state $[\alpha, s_k, \beta]$ and any local event event $e \in \mathcal{E}_k$ such that $Next_k^e(s_k) \neq \emptyset$, we have $[\alpha, s_k, \beta] \xrightarrow{e} [\alpha, s'_k, \beta]$, for any $s'_k \in Next_k^e(s_k)$. This means that we need to add the states $[\alpha, s'_k, \beta]$ to $\mathcal{S}$. Consider some path α to a level-k node representing function f in the MDD encoding $\mathcal{S}$. The set $\mathcal{B} = \{\beta : [\alpha, s_k, \beta] \in \mathcal{S}\}$ has characteristic function $\chi_{\mathcal{B}} = f_{x_k=\Psi(s_k)}$. States $[\alpha, s'_k, \beta]$ can be added for all $\beta \in \mathcal{B}$ by the assignment

$$f_{x_k=\Psi(s'_k)} \leftarrow \mathsf{Union}(f_{x_k=\Psi(s_k)}, f_{x_k=\Psi(s'_k)}) \tag{6.3}$$

where Union is the MDD manipulation for set union. To do this for all paths α we must perform this operation for every level-k node. Note that the absence of a level-k node in

a path implies that submodel k can be in any possible local state; in this case, the states reached when e occurs are already represented in the MDD. After performing the operation of Equation 6.3, we have

$$(s_K, \ldots, s_{k+1}, s_k, s_{k-1}, \ldots, s_1) \in \mathcal{S} \quad \Rightarrow \quad (s_K, \ldots, s_{k+1}, s'_k, s_{k-1}, \ldots, s_1) \in \mathcal{S} \tag{6.4}$$

for all currently known reachable states with submodel k in local state s_k. To see this, consider the path corresponding to state $(s_K, \ldots, s_1)$ from the top node of the MDD representing $\chi_{\mathcal{S}}$ to the terminal node 1. If this path does not pass through a level-k node, then we have

$$\forall y_k \in \mathcal{S}_k, \quad (s_K, \ldots, s_{k+1}, y_k, s_{k-1}, \ldots, s_1) \in \mathcal{S}$$

and Equation 6.4 trivially holds. Otherwise, the path encounters a level-k node representing a function f. Since the path ultimately reaches node 1, we know that the set represented by characteristic function $f_{x_k=s_k}$ contains element $(s_{k-1}, \ldots, s_1)$. This implies that the set represented by characteristic function $f_{x_k=s'_k}$ also contains element $(s_{k-1}, \ldots, s_1)$ since we just applied Equation 6.3. This means that the path corresponding to $(s_K, \ldots, s_{k+1}, s'_k, s_{k-1}, \ldots, s_1)$ reaches node 1.

Figure 6.6 shows an example MDD and the application of Equation 6.3 for a local event that changes the state of submodel 2 from 1 to 3. We apply Equation 6.3 to the level-2 nodes i, h, g and f in Figure 6.6(a):

Node i After performing the union, $i_{x_2=3} = i_{x_2=2} = i_{x_2=1} = i_{x_2=0}$, so node i is removed and the incoming pointers to i are re-directed to $i_{x_2=0}$.

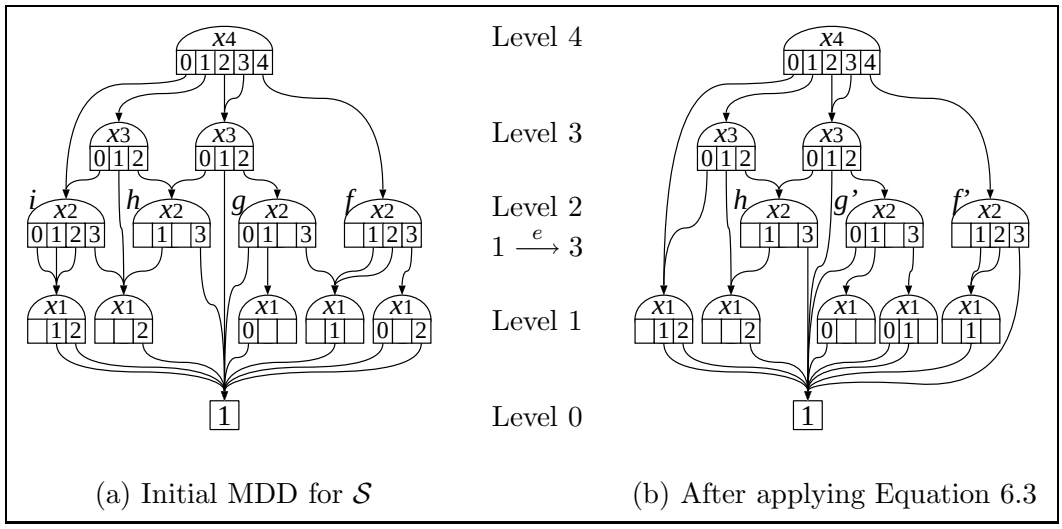


Figure 6.6: Application of Equation 6.3 for an event local to submodel 2

Node h Since $h_{x_2=3} = 1$, the union does not add any states; thus $h_{x_2=3}$ remains unchanged.

Node g The union causes the creation of a new node $g_{x_2=3}$.

Node f The union would create a new node where all the pointers are to terminal node 1; thus $f_{x_2=3} = 1$.

Note that while the manipulations in this example added 9 states to $\mathcal{S}$, our new MDD actually has fewer nodes.

We can repeatedly apply Equation 6.3 to a level-k node for all events e local to submodel k, all states $s_k \in \mathcal{S}_k$, and all new states $s'_k \in Next^e_k(s_k)$. This is the basis for the DoLocals operator shown in Algorithm 6.3. Given an MDD encoding a set of states $\mathcal{R}$, DoLocals($\mathcal{R}$) returns the set of states that can be reached by a state in $\mathcal{R}$ via any sequence of local events (including none). The algorithm visits each MDD node in $\chi_{\mathcal{R}}$ and, based on

```
DoLocals(R)
 1: if R is a constant r then                          • Check terminal cases
 2:    return r
 3: else if cache contains the entry for R then
 4:    return cache entry result
 5: end if
 6: let x_k be the top variable of R                   • Not a terminal case
 7: Changed ← ∅
 8: for i ← 0 to N_k − 1 do
 9:    H^i ← DoLocals(R_{x_k=i})
10:    if H^i ≠ 0 then
11:       Changed ← Changed ∪ {i}
12:    end if
13: end for
14: while Changed ≠ ∅ do
15:    remove some element i from Changed
16:    for each event e local to submodel k do
17:       for each j ∈ Next^e_k(i) do
18:          F ← Union(H^i, H^j)                      • Apply Equation 6.3
19:          if F ≠ H^j then
20:             Changed ← Changed ∪ {j}
21:             H^j ← F
22:          end if
23:       end for
24:    end for
25: end while
26: if H^0 = H^1 = ··· = H^{N_k−1} then
27:    N ← H^0
28: else
29:    N ← UniqueTableInsert(x_k, H^0, ..., H^{N_k−1})
30: end if
31: add [R, N] to cache
32: return N
```

Algorithm 6.3: Adding states due to local events

the variable label x_k of the node, applies Equation 6.3 until no new states are added. Note that only the modified pointers of the node are explored more than once. This is done by maintaining a set (*Changed*) of the modified pointers. Also, notice that we use a cache to reduce complexity (lines 3 and 32) and we return a reduced MDD (lines 27-31).

6.3.2 Occurrence of synchronizing events

Since a synchronizing event e will in general affect several submodels, the manipulations to simulate the occurrence of event e must involve several MDD nodes. Given an MDD representing some set of reachable states $\mathcal{S}$, our goal is to add the states that can be reached from states in $\mathcal{S}$ via event e. This is done in three distinct operations.

1. **Determine the states that enable event e.** Given event e, we can easily determine $\mathcal{E}(e)$, the set of all potential states that enable e, by computing the characteristic function

$$\chi_{\mathcal{E}(e)}(s_K, \ldots, s_1) = (Next_K^e(s_k) \neq \emptyset) \wedge \cdots \wedge (Next_1^e(s_1) \neq \emptyset)$$

   which holds because of the property of *logical-product-form* models. If the local states $\mathcal{S}_k$ are generated *a priori* for all submodels, then the MDD for $\chi_{\mathcal{E}(e)}$ can also be computed once *a priori* and used throughout the generation process. Otherwise, we must recompute $\chi_{\mathcal{E}(e)}$ whenever a new local state is added to $\mathcal{S}_k$. Either way, once we have obtained the set $\mathcal{E}(e)$, we can determine the *reachable* states that enable e as $\mathcal{E}(e) \cap \mathcal{S}$. Figure 6.7(a) shows the MDD for $\mathcal{E}(e)$ for a synchronizing event e that changes the submodels as shown. Figure 6.7(b) shows the result of the set intersection between $\mathcal{E}(e)$ and $\mathcal{S}$, where $\mathcal{S}$ is taken from Figure 6.6(b). Of course, the set intersection is done by manipulating MDDs according to the set operations in Equation 6.2.

2. **Determine the states reached after e occurs.** Next, we transform the set of states that enable e into the set of states reached after e occurs. This is done by

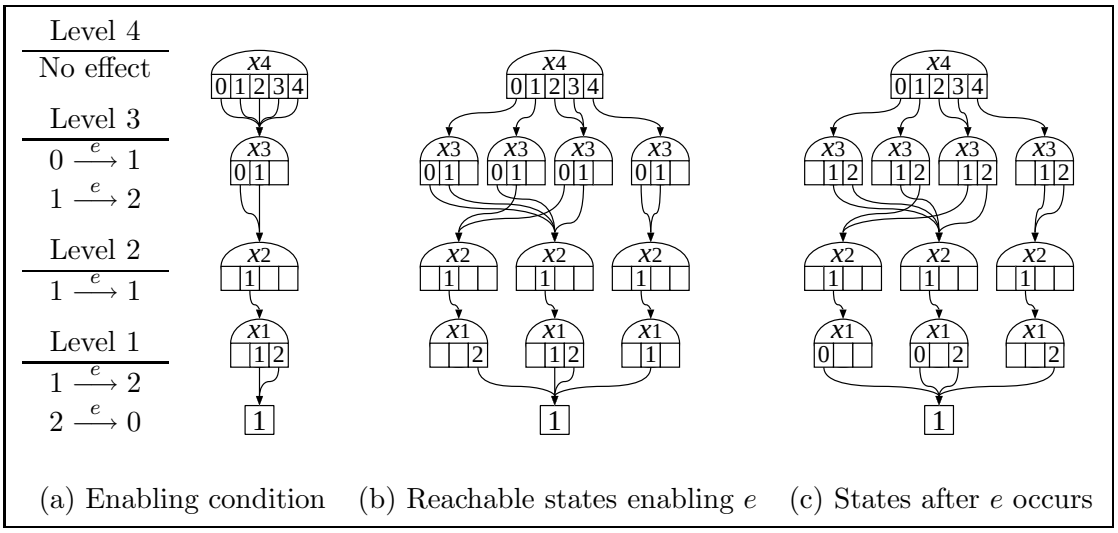


Figure 6.7: MDD example for a synchronizing event e

the Occurs operator shown in Algorithm 6.4. The operator is based on the idea that for submodel k, if event e causes the local state transition $i \xrightarrow{e} j$, then we need to perform the assignment

$$f_{x_k=j} \leftarrow \mathsf{Occurs}(e, f_{x_k=i}) \tag{6.5}$$

for each level-k MDD node f, which in effect "replaces i with j" at node f. If it is possible for different states to reach a state j (i.e., $i \xrightarrow{e} j$ and $i' \xrightarrow{e} j$) we must perform a set union instead of the direct assignment of Equation 6.5; this is in fact done in line 17 of Algorithm 6.4. Note that Equation 6.5 works properly only if the model is in logical product form. Our implementation of Occurs requires the parameter for the current MDD level (x_k in Algorithm 6.4). When an arc skips over variable x_k, that means all possible local states for submodel k enable event e. However, if event e affects submodel k, then the occurrence of e may not lead to all local states. Thus, we may need to do some processing and create a level-k node, even though the original

```
Occurs(x_k, R, e)
 1: if R = 0 then                                          • Check terminal cases
 2:    return 0
 3: else if k < Last(e) then                               • e does not affect R
 4:    return R
 5: else if cache contains entry for (x_k, R, e) then
 6:    return cache entry result
 7: end if
 8: let x_k' be the top variable of R                      • Not a terminal case
 9: while (e ∉ E_k) and (k > k') do                        • e does not affect level k,
10:    k ← k − 1                                          • and level k is skipped in R
11: end while
12: for i ← 0 to N_k − 1 do
13:    H^i ← 0
14: end for
15: for i ← 0 to N_k − 1 do
16:    for each j ∈ Next_k^e(i) do
17:       H^j ← Union(H^j, Occurs(x_{k−1}, R_{x_k=i}, e))   • Apply Equation 6.5
18:    end for
19: end for
20: if H^0 = H^1 = ··· = H^{N_k−1} then
21:    N ← H^0
22: else
23:    N ← UniqueTableInsert(x_k, H^0, ..., H^{N_k−1})
24: end if
25: add [(x_k, R, e), N] to cache
26: return N
```

Algorithm 6.4: Determining states due to synchronizing events

MDD does not contain a level-k node. Therefore, level k can be safely skipped only if it is skipped in the MDD and the event does not affect submodel k (see lines 8-11). As with other MDD manipulation algorithms, Algorithm 6.4 returns a reduced MDD (lines 19-23) and maintains a cache (line 24) to reduce complexity. Figure 6.7(c) shows the result of the Occurs operator on the MDD from Figure 6.7(b), assuming event e behaves as shown.

```
MDDexplore(Logical Product-Form Model M, state s°)
 1: S ← {s°_K} × ··· × {s°_1}                    • Set S to the initial state
 2: repeat
 3:    S ← DoLocals(S)
 4:    O ← S
 5:    for each synchronizing event e do
 6:       A ← Intersect(S, E(e))                 • A : states in which e is Active
 7:       F ← Occurs(x_K, A, e)                  • F : states reached after e occurs
 8:       S ← Union(S, F)
 9:    end for
10: until O = S
11: return S
```

Algorithm 6.5: Generating $\mathcal{S}$ using MDDs

3. **Add these new states to $\mathcal{S}$.** Once we have determined the new states using the Occurs operator, we can simply perform a set union to add the new states to $\mathcal{S}$.

6.3.3 Complete generation algorithm

The algorithms for simulating the occurrence of local and synchronizing events are put together to generate all reachable states using a fixed-point iteration in Algorithm 6.5. In the algorithm, $\mathcal{S}$ is represented using an MDD, which is initialized to the set containing only the initial state s° (line 1). Each iteration of the algorithm finds all states that can be reached from the current set of reachable states by a sequence of local events (line 3), then adds the states reached from the current set of reachable states by a single synchronizing event (lines 5-9). The iterations continue until no new states are reached when a synchronizing event occurs. Thus, the number of iterations required by MDDexplore is bounded by one (to recognize that $\mathcal{S}$ has not changed) plus the *synchronizing depth* of the model. This quantity

is defined as $\max(\{d(s) : s \in \mathcal{S}\})$, where $d(s)$ is

$$d(s) = \min(\{n \in I\!N : \ \exists e^1, \ldots, e^m \in \mathcal{E}, \text{with only } n \text{ of } e^1, \ldots, e^m \text{ synchronizing events}, \\ s^\circ \xrightarrow{e^1} s^1 \xrightarrow{e^2} \cdots \xrightarrow{e^m} s\}),$$

the number of synchronizing events required to reach state s from the initial state.

The actual number of iterations may be smaller if the synchronizing events are processed in the proper order: if e^i and e^{i+1} are synchronizing events in the minimum sequence, then event e^{i+1} is considered in the same iteration as e^i only if e^i is considered before e^{i+1}. This concept is similar to the notion of "chaining order" described in [88]. Note that if our model is a safe Petri net that is decomposed so that each place becomes its own submodel, then all events are synchronizing. In this case, Algorithm 6.5 performs the same computation as the BDD approach described in [88], although our specialized operators may be more efficient than the BDD operations required for Δ.

6.4 Logical queries on the state space

Once we have obtained an MDD encoding of $\mathcal{S}$, we can use it to answer various classes of logical queries, by performing MDD operations. For instance, suppose we want to compute the set of reachable states satisfying some boolean condition q. First, we build the set $\mathcal{Q}$ of potential states that satisfy q. Then, to determine the reachable states in which condition q is enabled, we simply compute $\mathcal{S} \cap \mathcal{Q}$, using MDD operators.

Let us first consider a simple query whose condition is enforced at a single submodel k. Given a condition $f_k : \mathcal{S}_k \to \{0, 1\}$, we compute the set of potential states $\mathcal{Q}$ in which f_k is

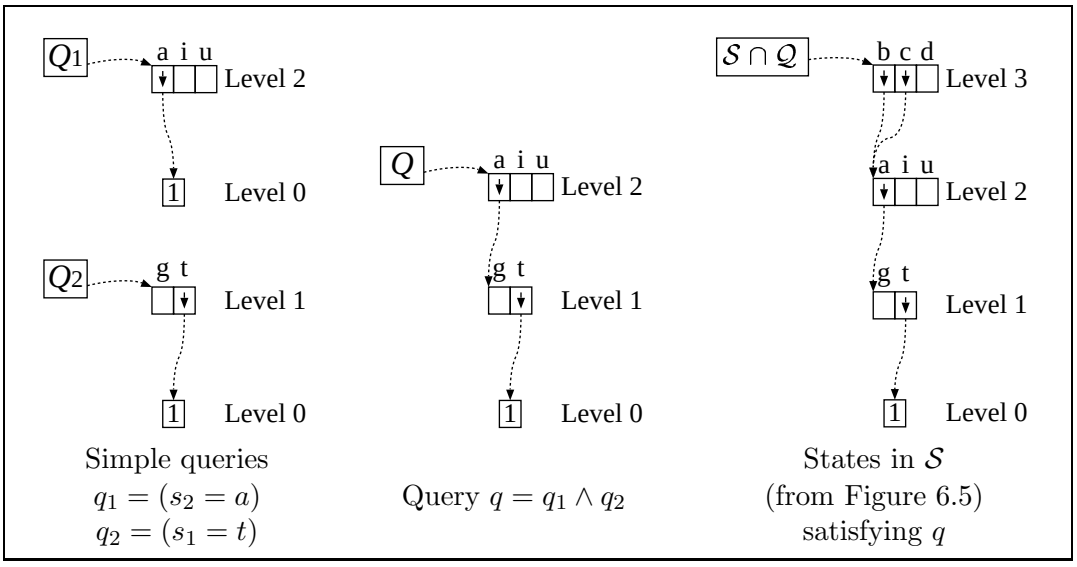


Figure 6.8: Performing a query on $\mathcal{S}$

satisfied:

$$\mathcal{Q} = \mathcal{S}_K \times \cdots \times \mathcal{S}_{k+1} \times \{s_k : f_k(s_k) = 1\} \times \mathcal{S}_{k-1} \times \cdots \times \mathcal{S}_1.$$

More complex queries are obtained by combining simple queries. This is illustrated in Figure 6.8, which operates on the set of reachable states in Figure 6.5. In the example, we have condition q_1 which simply requires that submodel 2 be in local state a, and condition q_2 which requires that submodel 1 be in local state t. The MDDs representing the potential states in which one of these conditions holds are trivial to compute. Combining conditions is done via set operations; in this case, to determine the potential states satisfying $q_1 \wedge q_2$, we perform the set intersection $\mathcal{Q}_1 \cap \mathcal{Q}_2$. The set of reachable states enabling $q_1 \wedge q_2$ is found by performing the set intersection $\mathcal{S} \cap \mathcal{Q}$.

An important example is determining the set of reachable states in which no events are enabled. Such a state is said to be *absorbing*, since the system must remain in that state

once it is reached. To determine the set of absorbing states, we first compute $\mathcal{Q}_k^e$, the set of local states in submodel k in which event e is disabled, by

$$\mathcal{Q}_k^e = \{s_k : Next_k^e(s_k) = \emptyset\}$$

for all events e and submodels k. Next, we compute the set of potential states in which event e is disabled by

$$\mathcal{Q}^e = \mathcal{Q}_K^e \cup \cdots \cup \mathcal{Q}_1^e$$

for all events e. We then compute the set of potential states in which all events are disabled:

$$\mathcal{Q} = \bigcap_{e \in \mathcal{E}} \mathcal{Q}^e.$$

Finally, $\mathcal{S} \cap \mathcal{Q}$ gives us the set of reachable absorbing states.

Another important class of queries deals with the possibility of a condition b occurring after a condition a has occurred. To determine if this is possible, we build the subsets $\mathcal{Q}_a$ and $\mathcal{Q}_b$ of $\hat{\mathcal{S}}$ in which potential states satisfy conditions a and b, respectively. Then, we run Algorithm 6.5 a second time, except that in line 1, we initialize $\mathcal{S}$ to the set $\mathcal{S} \cap \mathcal{Q}_a$: the reachable states satisfying condition a. In the worst case, this requires as many iterations as for the original generation of $\mathcal{S}$. The computed value of $\mathcal{S}$ gives us all states that can be reached from reachable states satisfying condition a. Finally, $\mathcal{S} \cap \mathcal{Q}_b$ gives us the set of states satisfying condition b that can be reached from some reachable state satisfying condition a.

6.5 Experimental results

This section shows how well our MDD-based algorithm compares with BDD-based algorithms from the literature. We use the benchmark models described in Appendix B, as we did in the previous chapter. We compare various encodings and different decompositions for Algorithm 6.5. Since an MDD encoding for $\mathcal{S}$ will grow to a maximum size and then contract during generation, the final memory required to store $\mathcal{S}$ can be much less than the peak memory required during generation. We report both measurements from our experiments.

6.5.1 Dining philosophers model

First, we examine the dining philosophers model. Figure 6.9 shows the effect of different decompositions and encoding techniques on generation times and memory usage. For the dining philosophers model, we compare the following.

BDD-1 : A decomposition of a single place per submodel, using Algorithm 6.5. Since the dining philosophers model is a safe Petri net, this decomposition is equivalent to using the BDD-based generation algorithm described in [88].

BDD-2 : We consider an entire philosopher and fork as a submodel, as shown in Figure B.3. We use a BDD-based generation algorithm with a binary encoding: states are encoded using $v_k = \lceil \log_2(|\mathcal{S}_k|) \rceil$ boolean variables for each submodel k.

MDD-1 : We use Algorithm 6.5 with a decomposition of a philosopher and fork as a submodel.

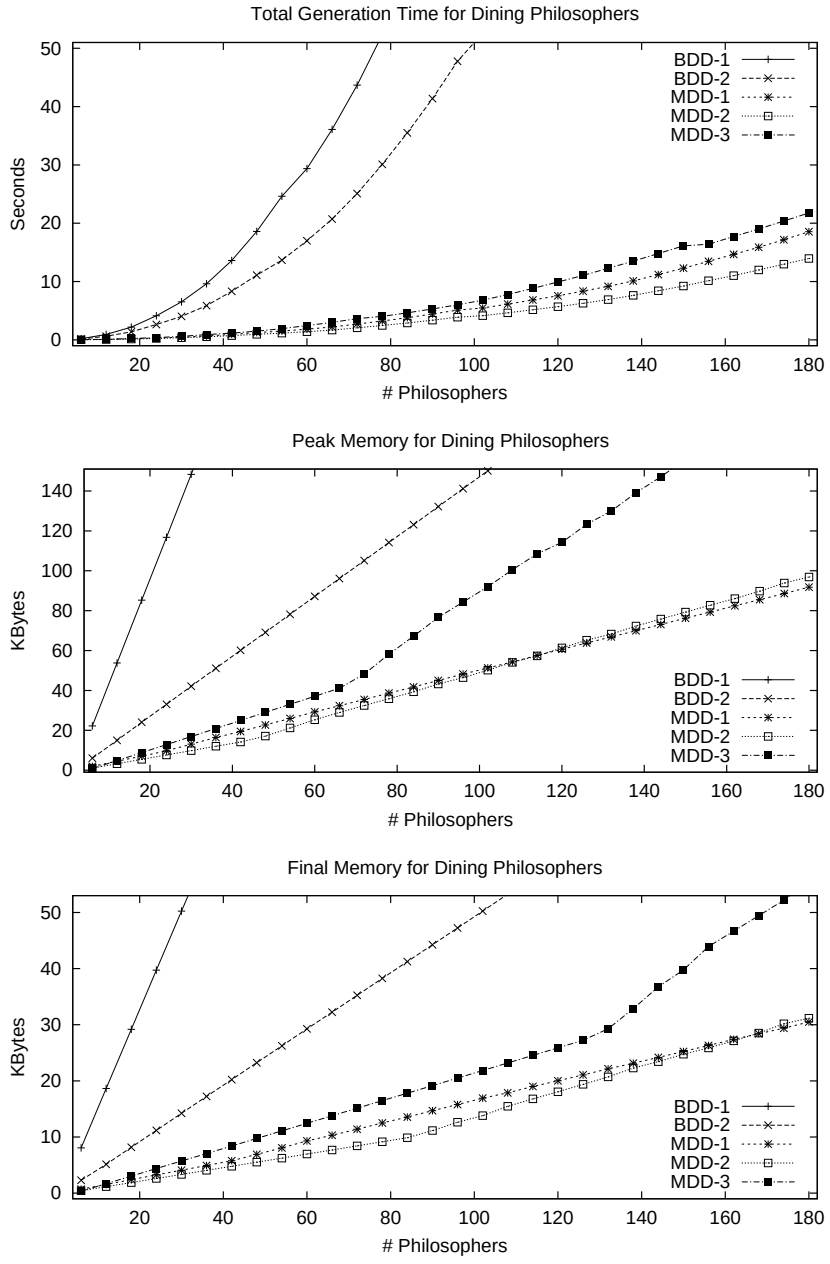


Figure 6.9: Symbolic generation times and memory usage for Dining Philosophers

MDD-2 : We use Algorithm 6.5 with a decomposition of two adjacent philosophers and their forks as a submodel. That is, we combine two adjacent submodels from the **MDD-1** decomposition.

MDD-3 : We use Algorithm 6.5 with a decomposition of three adjacent philosophers and their forks as a submodel. That is, we combine three adjacent submodels from the **MDD-1** decomposition.

We only consider models whose number of philosophers is a multiple of six. This allows us to divide the philosophers into groups of two or three (for cases **MDD-2** and **MDD-3**) with no remainder. Of course, this is not a requirement; we are free to use a decomposition where the submodels contain different numbers of philosophers. Note that the "bumps" in the curves are artifacts of a simple compression technique used for the storage of MDD nodes: if the total number of MDD nodes is less than 2^{8b}, then an MDD node uses an array of b-byte integers to store the downward pointers. As Figure 6.9 indicates, Algorithm 6.5 using MDD-based encodings is more memory- and CPU-efficient than the BDD-based encodings we considered. In this case, our binary BDD encoding performs better than a simple BDD encoding but not as well as our MDD encoding.

The number of philosophers grouped into each submodel affects the efficiency of Algorithm 6.5. In general, larger submodels produce MDDs with fewer, larger nodes (i.e., the nodes contain more downward pointers). Since there is a memory and CPU overhead associated with storing and processing a node, this overhead becomes dominant if the MDD contains a large number of small nodes. However, if nodes are too large, then the MDD may contain many "sparse" nodes (i.e., nodes with a large number of downward pointers to node 0). These sparse nodes can be costly both in terms of memory (if a large node is used to represent only a few states) and CPU (if a large node is processed but only a few states are added). This implies that submodels should neither be too large nor too small. For the dining philosophers model, we see from Figure 6.9 that three philosophers grouped

# of Phils.	# of states $\lvert\mathcal{S}\rvert$	# of Nodes Final	# of Nodes Peak	Memory (bytes) Final	Memory (bytes) Peak	# of Iters.	CPU (sec.)
10	1.86×10^{6}	17	45	938	2,616	2	0.04
20	3.46×10^{12}	37	105	2,178	6,336	2	0.16
50	2.23×10^{31}	97	285	5,898	18,924	2	1.04
100	4.97×10^{62}	197	585	13,832	50,206	2	4.23
200	2.47×10^{125}	397	1,185	35,344	110,390	2	17.5
300	1.23×10^{188}	597	1,785	55,904	169,486	2	39.05
400	6.10×10^{250}	797	2,385	75,172	227,494	2	75.26
500	3.03×10^{313}	997	2,985	94,372	285,094	2	113.63
600	1.51×10^{376}	1,197	3,585	114,116	343,238	2	174.25
700	7.48×10^{438}	1,397	4,185	133,452	400,974	2	252.56
800	3.72×10^{501}	1,597	4,785	152,448	458,302	2	343.02
900	1.85×10^{564}	1,797	5,385	171,716	516,038	2	421.99
1000	9.18×10^{626}	1,997	5,985	190,984	573,638	2	533.86

Table 6.1: Results for Dining Philosophers, two philosophers per submodel

together is too many, while one or two philosophers produces good results.

Table 6.1 shows detailed results for the decomposition of two philosophers in each submodel. In the table, we report the final and peak number of non-terminal nodes in the MDD, the final and peak memory requirements, the number of iterations required by Algorithm 6.5, and the total time to generate $\mathcal{S}$ in seconds. Notice that Algorithm 6.5 requires only two iterations for this model. This is the case for any number of adjacent philosophers in a submodel. The synchronizing depth of the model grows as N, the number of philosophers, since the longest sequences of synchronizing events required are those to reach the states where all N forks are taken (i.e., every philosopher has a left fork or every philosopher has a right fork). However, any such state can be reached from the initial state through $N!$ different sequences of length N, since the order in which philosophers take a fork is irrelevant. Thus, there is always a sequence that respects the order in which the synchronizing events are considered in Algorithm 6.5.

Also, notice that the final number of MDD nodes is exactly $2N - 3$. This is because the final number of MDD nodes per submodel is exactly four (except at the "top level" of the MDD, which always has a single node), no matter how many adjacent philosophers are grouped per submodel. The meaning of these four nodes is illustrated in Appendix D for the case of one philosopher per submodel. Hence, the final number of MDD nodes is $4(N/g - 1) + 1$, where g is the number of adjacent philosophers grouped into a submodel. The peak number of nodes is also linear: $6N - 15$. It is interesting to note that the number of reachable states is given by (see Appendix D for the derivation)

$$|\mathcal{S}| = Fib(3N + 1) + Fib(3N - 1)$$

where N is the number of philosophers and Fib denotes the *Fibonacci* sequence [58]:

$$\begin{aligned} Fib(0) &= 0 \\ Fib(1) &= 1 \\ Fib(n+2) &= Fib(n) + Fib(n+1), \quad \forall n \in \mathbb{N}. \end{aligned}$$

Since terms of the Fibonacci sequence are bounded by

$$\phi^{n-2} \leq Fib(n) \leq \phi^{n-1}, \qquad \phi = \frac{1+\sqrt{5}}{2},$$

for positive integers n, the number of states grows exponentially with the number of philosophers. Since the number of MDD nodes grows linearly with N, and each node N contains a fixed number of pointers to nodes, Algorithm 6.5 requires $O(N \log_2(N))$ memory.

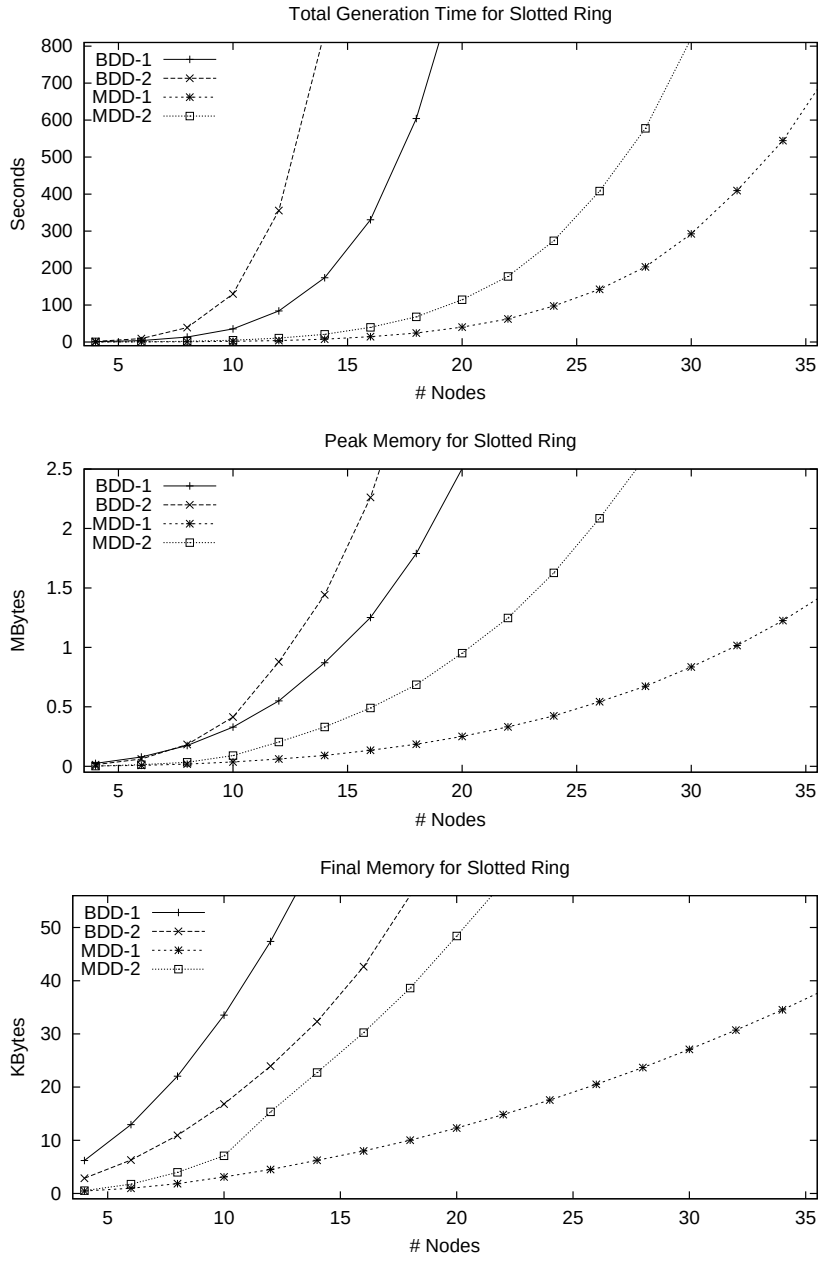


Figure 6.10: Symbolic generation times and memory usage for Slotted Ring

6.5.2 Slotted ring model

Figure 6.10 shows the effect of different decompositions and encoding techniques on gener-

ation times and memory usage for the slotted ring model, where we compare the following.

BDD-1 : A decomposition of a single place per submodel, using Algorithm 6.5. Since the slotted ring model is a safe Petri net, this decomposition is equivalent to using the BDD-based generation algorithm described in [88].

BDD-2 : We consider a network node as a submodel, as shown in Figure B.5. We use a BDD-based generation algorithm with a binary encoding: states are encoded using $v_k = \lceil \log_2(|\mathcal{S}_k|) \rceil$ boolean variables for each submodel k.

MDD-1 : We use Algorithm 6.5 with a decomposition of a single network node as a submodel.

MDD-2 : We use Algorithm 6.5 with a decomposition of two adjacent network nodes as a submodel. That is, we combine two adjacent submodels from the **MDD-1** decomposition.

We only consider cases where the number of network nodes is a multiple of two. Similar to the dining philosophers model, Algorithm 6.5 using MDD-based encodings is more memory- and CPU-efficient than the BDD-based encodings we considered. In this case, the simple BDD encoding performs better than the binary BDD encoding of a submodel. This is due to the complexity of the Δ function for this case. However, the final binary BDD encoding requires less memory than the simple BDD encoding, since fewer variables are used. We see from Figure 6.9 that the best results are obtained from the slotted ring model using MDDs with the decomposition of a single network node per submodel.

Table 6.2 shows detailed results for the decomposition of a single network node per submodel. In the table, we report the final and peak number of non-terminal nodes in

# of Nodes	# of states $\|\mathcal{S}\|$	MDD Nodes Final	MDD Nodes Peak	Memory (bytes) Final	Memory (bytes) Peak	# of Iters.	CPU (sec.)
10	8.29×10^{9}	60	691	3,186	38,527	7	1.73
20	2.73×10^{20}	220	4,546	12,601	263,002	12	40.26
30	1.04×10^{31}	480	15,101	27,741	875,327	17	292.66
40	4.16×10^{41}	840	37,066	48,606	2,149,372	22	1,286.06
50	1.72×10^{52}	1,300	76,308	85,486	5,342,493	27	4,168.64
60	7.29×10^{62}	1,860	139,852	149,106	11,902,318	32	11,465.30
70	3.13×10^{73}	2,520	236,733	212,736	20,666,298	37	27,374.60
80	1.36×10^{84}	3,280	376,222	281,626	33,020,245	42	59,872.00

Table 6.2: Results for Slotted Ring, one node per submodel

the MDD, the final and peak memory requirements, the number of iterations required by Algorithm 6.5, and the total time to generate $\mathcal{S}$ in seconds. Notice that the number of iterations is $N/2 + 2$, and the final number of nodes is $N^2/2 + N$ for a slotted ring model with N network nodes.

6.5.3 FMS model

Figure 6.11 shows the effect of different decompositions on generation times and memory usage for the FMS model using Algorithm 6.5. Table 6.3 depicts the decompositions used by specifying the submodel assignment for each place in the model. Notice that the "A" and "B" decompositions differ only in the ordering of the submodels. From Figure 6.11, we see that the order of the submodels affects the efficiency of Algorithm 6.5, but the sizes of the submodels have a greater impact. It is clear from the plots that the 19-submodel decompositions give the best performance in terms of both memory and CPU for this model.

Table 6.4 shows detailed results for decomposition "19-B". In the table, we report the final and peak number of non-terminal nodes in the MDD, the final and peak memory

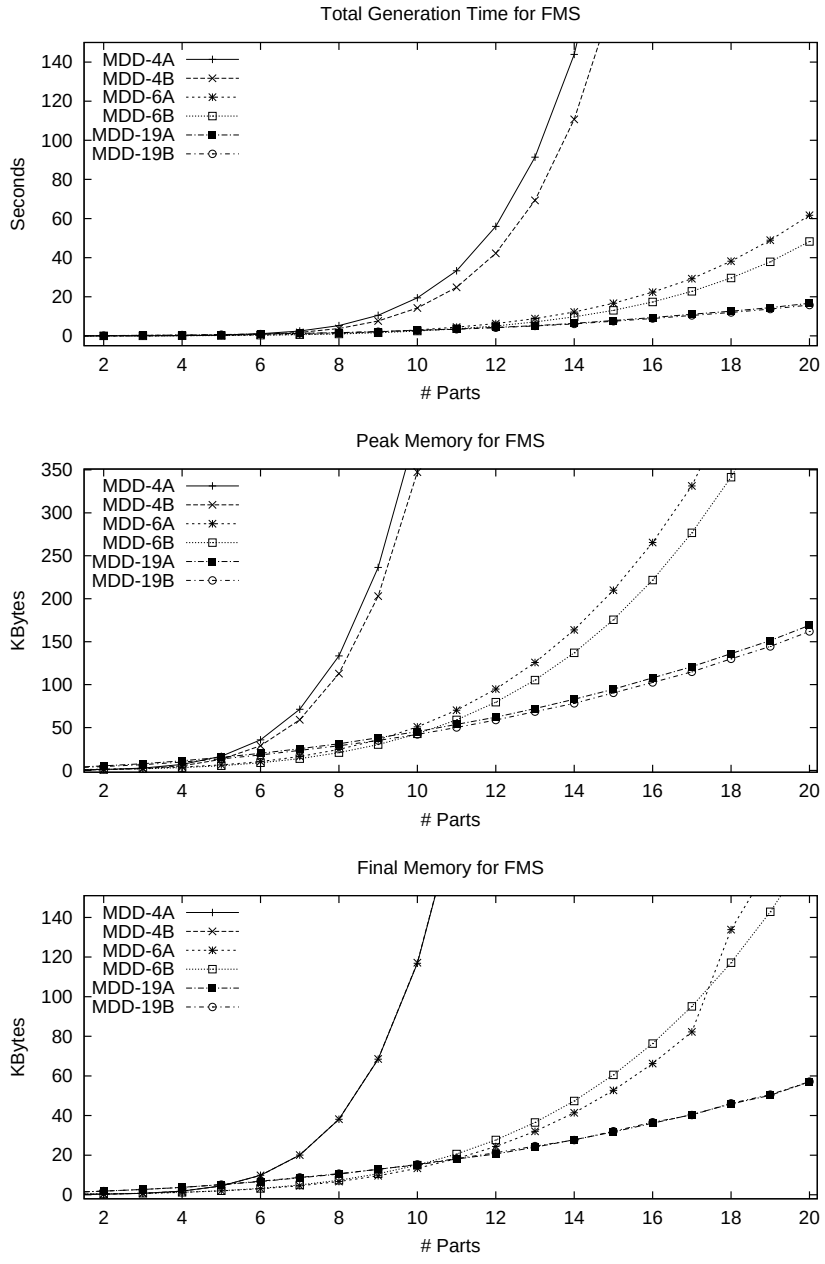


Figure 6.11: Symbolic generation times and memory usage for FMS

requirements, the number of iterations required by Algorithm 6.5, and the total time to generate $\mathcal{S}$ in seconds. For this decomposition, the number of iterations required is $N + 5$, where N is the number of jobs in the system. Considering the final number of nodes per level

	Submodel assignment					
Place	**4A**	**4B**	**6A**	**6B**	**19A**	**19B**
P_1	4	1	6	1	19	1
$P_1 w M_1$	4	1	6	1	18	2
$P_1 M_1, M_1$	4	1	6	1	17	3
$P_1 d$	4	1	5	2	16	4
$P_1 s$	4	1	5	2	15	5
$P_1 w P_2$	3	2	5	2	14	6
P_{12}	3	2	4	3	13	7
$P_{12} w M_3$	3	2	4	3	12	8
$P_{12} M_3, M_3$	3	2	4	3	11	9
$P_{12} s$	3	2	4	3	10	10
P_2	2	3	3	5	9	11
$P_2 w M_2$	2	3	3	5	8	12
$P_2 M_2, M_2$	2	3	3	5	7	13
$P_2 d$	2	3	2	4	6	14
$P_2 w P_1$	3	2	2	4	5	15
$P_2 s$	2	3	2	4	4	16
P_3	1	4	1	6	3	17
$P_3 M_2$	1	4	1	6	2	18
$P_3 s$	1	4	1	6	1	19

Table 6.3: FMS decompositions

# of Jobs	# of states $\lvert\mathcal{S}\rvert$	MDD Nodes Final	MDD Nodes Peak	Memory (bytes) Final	Memory (bytes) Peak	# of Iters.	CPU (sec.)
5	2.90×10^{6}	149	387	5,301	13,920	10	0.64
10	2.50×10^{9}	354	962	15,695	43,022	15	2.80
25	8.54×10^{13}	1,419	4,037	93,676	268,272	30	30.56
50	4.24×10^{17}	4,694	13,676	466,442	1,367,182	55	259.43
75	6.98×10^{19}	9,844	28,973	1,311,737	3,870,611	80	1,013.69
100	2.70×10^{21}	16,869	49,898	2,804,725	8,322,130	105	2,721.11
125	4.68×10^{22}	25,769	76,448	7,468,732	22,653,472	130	6,668.86
150	4.84×10^{23}	36,544	108,623	13,886,395	41,659,845	155	13,175.70
175	3.51×10^{24}	49,194	146,423	22,444,916	66,902,036	180	33,139.80
200	1.95×10^{25}	63,719	189,848	33,382,402	100,941,777	205	49,419.40

Table 6.4: Results for FMS, 19 submodel decomposition

of the MDD, we find that two levels have 1 node, 14 levels have $N+1$ nodes, and 3 levels have $\frac{1}{2}(N+1)(N+2)$ nodes. Thus the final number of nodes is $2+14(N+1)+\frac{3}{2}(N^2+3N+2)$.

Of course, the size of each node increases as $N \log_2(N)$ for this decomposition (since each place is N-bounded), so the final memory required is $O(N^3 \log_2(N))$.

6.5.4 Kanban model

Figure 6.12 shows the effect of different decompositions on generation times and memory usage for the Kanban model using Algorithm 6.5. For the 4-submodel decomposition, we group places $\{Pm_k, Pback_k, Pkan_k, Pout_k\}$ for submodel k. The 3-submodel decomposition is identical, except we combine submodels 2 and 3. The 16-submodel decompositions treat each place as a submodel. The ordering used for "16-A" is $Pm_k, Pback_k, Pkan_k, Pout_k$ for $k \in \{1, \ldots, 4\}$, and the ordering for "16-B" is $Pkan_k, Pm_k, Pback_k, Pout_k$. From Figure 6.12, we see that the order of the submodels affects the efficiency of Algorithm 6.5, but the sizes of the submodels have a greater impact. Note that the final memory required for the 3-submodel decomposition is $O(1)$; this is because this decomposition gives us $\hat{\mathcal{S}} = \mathcal{S}$, and so the characteristic function is identically one. Of course, the peak memory for this decomposition is not constant; in fact, as the number of jobs N increases, the CPU time and peak memory for the 3-submodel decomposition grows quite high, as the second submodel (the combination of submodels 2 and 3 from the 4-submodel decomposition) has a large number of states:

$$|\mathcal{S}_{2,3}| = \frac{N+1}{4}\left(\frac{N(2N+1)(3N^2+3N-1)}{30} + \frac{3N^2(N+1)}{2} + \frac{13N(2N+1)}{6} + 6N + 4\right)$$

which is $O(N^5)$. In contrast, the 4-submodel decomposition produces submodels with

$$|\mathcal{S}_1| = |\mathcal{S}_2| = |\mathcal{S}_3| = |\mathcal{S}_4| = \frac{(N+1)(N+2)(N+3)}{6}$$

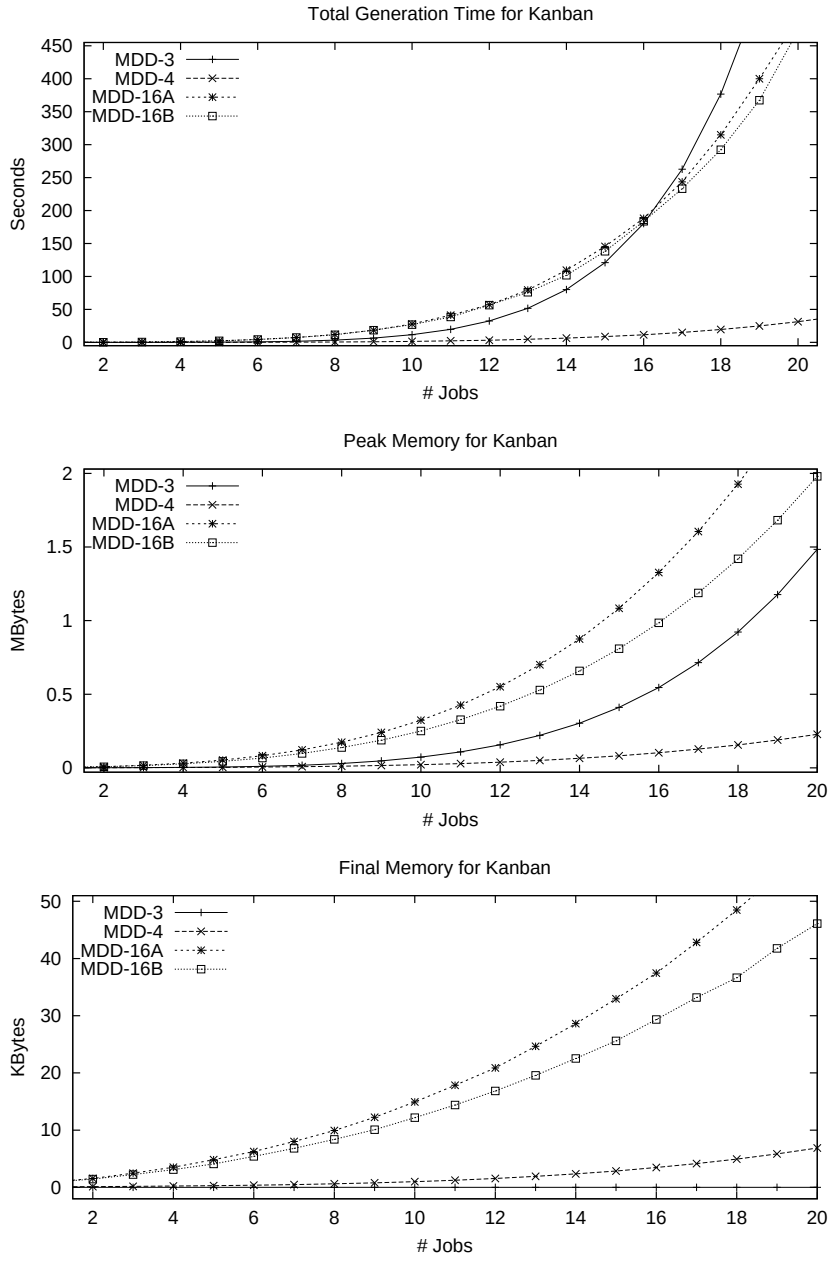


Figure 6.12: Symbolic generation times and memory usage for Kanban

which is only $O(N^3)$. The best overall case for this model is the 4-submodel decomposition.

Table 6.5 shows detailed results for the 4-submodel decomposition. In the table, we report the final and peak number of non-terminal nodes in the MDD, the final and peak

# of Jobs	# of states $\lvert\mathcal{S}\rvert$	MDD Nodes Final	MDD Nodes Peak	Memory (bytes) Final	Memory (bytes) Peak	# of Iters.	CPU (sec.)
10	1.01×10^{9}	12	87	1,018	22,318	21	1.52
20	8.05×10^{11}	22	167	7,049	238,473	41	31.42
30	4.99×10^{13}	32	247	27,494	1,068,108	61	206.93
40	9.94×10^{14}	42	327	89,121	3,822,555	81	801.21
50	1.04×10^{16}	52	407	197,687	9,510,714	101	2,331.34
60	7.22×10^{16}	62	487	383,962	20,901,637	121	5,669.65
70	3.74×10^{17}	72	567	678,433	39,354,916	141	11,998.50
80	1.56×10^{18}	82	647	1,116,424	67,918,203	161	23,356.20
90	5.55×10^{18}	92	727	1,738,219	109,737,442	181	41,462.30

Table 6.5: Results for Kanban, 4 submodel decomposition

memory requirements, the number of iterations required by Algorithm 6.5, and the total time to generate $\mathcal{S}$ in seconds. For this decomposition, the number of iterations is $2N + 1$. Considering the final number of nodes per level, we see that no nodes are required at levels 1 and 4, one node is required at level 2, and $N + 1$ nodes are required at level 3. Thus the final number of nodes is exactly $N + 2$. The peak number of nodes is also linear: $8N + 7$. Since each node must store $O(N^3)$ pointers of size $O(\log_2(N))$, Algorithm 6.5 requires $O(N^4 \log_2(N))$ memory to generate and store $\mathcal{S}$.

For this model, we can compute a closed-form expression for the total number of states. Since the 3-submodel decomposition gives us $\mathcal{S} = \hat{\mathcal{S}}$, we have

$$
\begin{aligned}
|\mathcal{S}| &= |\mathcal{S}_1| \cdot |\mathcal{S}_{2,3}| \cdot |\mathcal{S}_4| \\
&= \left(\frac{(N+1)(N+2)(N+3)}{6}\right) \cdot \\
&\quad \frac{N+1}{4}\left(\frac{N(2N+1)(3N^2+3N-1)}{30} + \frac{3N^2(N+1)}{2} + \frac{13N(2N+1)}{6} + 6N + 4\right) \\
&\quad \cdot \left(\frac{(N+1)(N+2)(N+3)}{6}\right)
\end{aligned}
$$

which means the number of states is $O(N^{11})$.

6.6 Conclusion

The MDD-based algorithm we presented is very promising for generating and storing large sets of reachable states. Experimentally, we have shown that Algorithm 6.5 can generate extremely large sets (10^{600} in one case) with acceptable memory and CPU requirements. For the models we studied, our algorithm shows remarkable improvements as compared to other BDD-based algorithms. The algorithm is quite flexible: any *logical-product-form* decomposition may be used. A drawback is that the efficiency of Algorithm 6.5 can be strongly affected by the choice of decomposition, in terms of both the contents of each submodel and the ordering of the submodels. Similar effects are seen in BDD-based algorithms: although there is no choice of "partition", since this is governed by the encoding method used for discrete (non-boolean) variables, it is well-known that variable ordering is of great importance [10, 11]. Some efforts have been made to find good variable orderings [9], including dynamic variable ordering [45, 90]. In future work, we plan to investigate heuristics for choosing decompositions that cause Algorithm 6.5 to perform well.

Slight implementation modifications may improve the performance of Algorithm 6.5. In particular, performance suffers from the presence of many large, sparse nodes because nodes are stored in full in our implementation. The ability to represent sparse nodes efficiently while still maintaining full storage for dense nodes could greatly enhance both memory and CPU efficiency.

Chapter 7

Transition Rate Matrix Storage

This chapter discusses storage of the transition rate matrix using Kronecker algebra. In a Kronecker-based representation, elements of $\hat{\mathbf{R}}$, the transition rate matrix of the underlying CTMC based on the potential states, are computed as needed from a Kronecker expression involving several small matrices. While a Kronecker representation requires only a fraction of the storage necessary for the entire matrix $\mathbf{R}$, the price to be paid is the introduction of CPU overhead during the numerical solution. This idea was introduced by Plateau [84, 85, 86] for a high-level formalism called *stochastic automata networks*. Plateau's idea was adapted for use with Petri nets by Donatelli [41, 42], and since then has received wide attention by many researchers, who have improved techniques and applied Kronecker representations to a variety of models [12, 14, 16, 18, 29, 44, 56, 91].

Several alternatives to Kronecker representations are worth mentioning here. In [39], the authors describe a technique in which the elements of $\mathbf{R}$ are generated "on the fly". Generating a row of $\mathbf{R}$ involves determining the states reachable from the state corresponding to the current row, and determining the rate of transition to those states. Columns of $\mathbf{R}$ are generated by considering the behavior of the model "backwards". Another interesting alternative is storing the matrix $\mathbf{R}$ on disk [38]. The next block of rows or columns

is fetched while the current block is being processed. Of course, this does not work if the matrix $\mathbf{R}$ requires more storage than is available on disk. Finally, there has been some work on representing matrices using multiple terminal BDDs (MTBDDs)[46, 52]. The variables for the BDD correspond to binary representations for the row and column indices, and the terminal values correspond to the values of the elements of $\mathbf{R}$. While this is an interesting idea, there are some cases in which it does not work well. For instance, if every non-zero element in $\mathbf{R}$ has a different value, then an MTBDD encoding of $\mathbf{R}$ will require at least as much storage space as an explicit sparse representation of $\mathbf{R}$.

The remainder of the chapter is organized as follows. Section 7.1 introduces Kronecker algebra as it relates to our work. Section 7.2 discusses how storage of the matrices of a Kronecker representation can affect computational complexity. Section 7.3 shows how the transition rate matrix $\hat{\mathbf{R}}$ for any *Kronecker-product-form* model can be represented with Kronecker algebra. Section 7.4 briefly discusses the overheads present in state-of-the-art Kronecker-based techniques, in particular the overheads caused by logarithmic searches on the data structure for $\mathcal{S}$, and by spurious entries in $\hat{\mathbf{R}}$.

Our contributions [28] appear in Section 7.5, which describes how the logarithmic searches can be eliminated with the use of an appropriate data structure for $\mathcal{S}$, and in Section 7.6, which describes a new data structure called *matrix diagrams*. We show how the use of *matrix diagrams* can eliminate the spurious entries by representing $\mathbf{R}$ instead of $\hat{\mathbf{R}}$, and how common subproducts can be computed only once by using a cache. Section 7.7 gives experimental results in which our technique is compared to previous state-of-the-art techniques. Finally, Section 7.8 contains concluding remarks.

7.1 Kronecker algebra

First, we recall the definition of the Kronecker product [37]. The Kronecker product expresses the process of multiplying every element of one matrix by a second matrix. Given matrices $\mathbf{A} \in \mathbb{R}^{m \times n}$ and $\mathbf{B} \in \mathbb{R}^{p \times q}$,

$$\mathbf{A} = \begin{bmatrix} a_{0,0} & \cdots & a_{0,n-1} \\ \vdots & \ddots & \vdots \\ a_{m-1,0} & \cdots & a_{m-1,n-1} \end{bmatrix} \quad \text{and} \quad \mathbf{B} = \begin{bmatrix} b_{0,0} & \cdots & b_{0,q-1} \\ \vdots & \ddots & \vdots \\ b_{p-1,0} & \cdots & b_{p-1,q-1} \end{bmatrix},$$

their Kronecker product $\mathbf{A} \otimes \mathbf{B} \in \mathbb{R}^{mp \times nq}$ is given by

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} a_{0,0}\mathbf{B} & \cdots & a_{0,n-1}\mathbf{B} \\ \vdots & \ddots & \vdots \\ a_{m-1,0}\mathbf{B} & \cdots & a_{m-1,n-1}\mathbf{B} \end{bmatrix} =$$

$$\begin{bmatrix} a_{0,0}b_{0,0} & \cdots & a_{0,0}b_{0,q-1} & \cdots & a_{0,n-1}b_{0,0} & \cdots & a_{0,n-1}b_{0,q-1} \\ \vdots & & \vdots & & \vdots & & \vdots \\ a_{0,0}b_{p-1,0} & \cdots & a_{0,0}b_{p-1,q-1} & \cdots & a_{0,n-1}b_{p-1,0} & \cdots & a_{0,n-1}b_{p-1,q-1} \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ a_{m-1,0}b_{0,0} & \cdots & a_{m-1,0}b_{0,q-1} & \cdots & a_{m-1,n-1}b_{0,0} & \cdots & a_{m-1,n-1}b_{0,q-1} \\ \vdots & & \vdots & & \vdots & & \vdots \\ a_{m-1,0}b_{p-1,0} & \cdots & a_{m-1,0}b_{p-1,q-1} & \cdots & a_{m-1,n-1}b_{p-1,0} & \cdots & a_{m-1,n-1}b_{p-1,q-1} \end{bmatrix}.$$

Instead of storing $\mathbf{A} \otimes \mathbf{B}$ explicitly, which requires $O(mnpq)$ storage, we can store $\mathbf{A}$ and $\mathbf{B}$ and compute elements of $\mathbf{A} \otimes \mathbf{B}$ as needed by

$$(\mathbf{A} \otimes \mathbf{B})[i,j] = \mathbf{A}\left[\left\lfloor \frac{i}{p} \right\rfloor, \left\lfloor \frac{j}{q} \right\rfloor\right] \mathbf{B}[i \bmod p, j \bmod q]$$

which only requires $O(mn + pq)$ storage, at the cost of some CPU overhead.

This process can be generalized to the Kronecker product of K matrices

$$\mathbf{A} = \mathbf{A}_K \otimes \cdots \otimes \mathbf{A}_1 = \bigotimes_{k=K}^{1} \mathbf{A}_k$$

$$\mathbf{A}_3 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad \mathbf{A}_2 = \begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} \\ \frac{1}{4} & \frac{1}{5} & \frac{1}{6} \end{bmatrix} \quad \mathbf{A}_1 = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 0 & 5 & 6 \\ 7 & 8 & 0 & 9 \end{bmatrix}$$

$$\mathbf{A} = \mathbf{A}_3 \otimes \mathbf{A}_2 \otimes \mathbf{A}_1 = \left[\begin{array}{cccccccccccc}
0 & 1 & 2 & 3 & 0 & \frac{1}{2} & 1 & \frac{3}{2} & 0 & \frac{1}{3} & \frac{2}{3} & 1 \\
4 & 0 & 5 & 6 & 2 & 0 & \frac{5}{2} & 3 & \frac{4}{3} & 0 & \frac{5}{3} & 2 \\
7 & 8 & 0 & 9 & \frac{7}{2} & 4 & 0 & \frac{9}{2} & \frac{7}{3} & \frac{8}{3} & 0 & 3 \\
0 & \frac{1}{4} & \frac{1}{2} & \frac{3}{4} & 0 & \frac{1}{5} & \frac{2}{5} & \frac{3}{5} & 0 & \frac{1}{6} & \frac{1}{3} & \frac{1}{2} \\
1 & 0 & \frac{5}{4} & \frac{3}{2} & \frac{4}{5} & 0 & 1 & \frac{6}{5} & \frac{2}{3} & 0 & \frac{5}{6} & 1 \\
\frac{7}{4} & 2 & 0 & \frac{9}{4} & \frac{7}{5} & \frac{8}{5} & 0 & \frac{9}{5} & \frac{7}{6} & \frac{4}{3} & 0 & \frac{3}{2} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}\right.$$

$$\left.\begin{array}{cccccccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 2 & 3 & 0 & \frac{1}{2} & 1 & \frac{3}{2} & 0 & \frac{1}{3} & \frac{2}{3} & 1 \\
4 & 0 & 5 & 6 & 2 & 0 & \frac{5}{2} & 3 & \frac{4}{3} & 0 & \frac{5}{3} & 2 \\
7 & 8 & 0 & 9 & \frac{7}{2} & 4 & 0 & \frac{9}{2} & \frac{7}{3} & \frac{8}{3} & 0 & 3 \\
0 & \frac{1}{4} & \frac{1}{2} & \frac{3}{4} & 0 & \frac{1}{5} & \frac{2}{5} & \frac{3}{5} & 0 & \frac{1}{6} & \frac{1}{3} & \frac{1}{2} \\
1 & 0 & \frac{5}{4} & \frac{3}{2} & \frac{4}{5} & 0 & 1 & \frac{6}{5} & \frac{2}{3} & 0 & \frac{5}{6} & 1 \\
\frac{7}{4} & 2 & 0 & \frac{9}{4} & \frac{7}{5} & \frac{8}{5} & 0 & \frac{9}{5} & \frac{7}{6} & \frac{4}{3} & 0 & \frac{3}{2}
\end{array}\right]$$

Figure 7.1: Example Kronecker product

(which can be written without parentheses since the Kronecker product is associative [37]). The expression for an element of $\mathbf{A}$ is closely related to the notion of a mixed-radix number representation system (see Section 4.2). If matrix $\mathbf{A}_k$ has r_k rows and c_k columns, then an element of $\mathbf{A}$ can be computed by

$$\mathbf{A}[i,j] = \prod_{k=1}^{K} \mathbf{A}_k[i_k, j_k] \tag{7.1}$$

where $\mathbf{i} = [i_K, \ldots, i_1]$ is the mixed-radix representation of i with respect to the basis $\mathbf{r} = [r_K, \ldots, r_1]$, and $\mathbf{j} = [j_K, \ldots, j_1]$ is the mixed-radix representation of j with respect to the basis $\mathbf{c} = [c_K, \ldots, c_1]$.

An example showing the Kronecker product of three small matrices is shown in Fig-

ure 7.1. Since the sizes of $\mathbf{A}_3$, $\mathbf{A}_2$, and $\mathbf{A}_1$ are 2×2, 2×3, and 3×4 respectively, we have $\mathbf{r} = [2, 2, 3]$ and $\mathbf{c} = [2, 3, 4]$ for this example. Then we compute $[1, 1, 2]$ as the mixed-radix representation of $11 = 1 \cdot 2 \cdot 3 + 1 \cdot 3 + 2$ with respect to basis $\mathbf{r}$ and $[1, 2, 0]$ as the mixed-radix representation of $20 = 1 \cdot 3 \cdot 4 + 2 \cdot 4 + 0$ with respect to basis $\mathbf{c}$; finally we can compute element $[11, 20]$ of $\mathbf{A}$ as

$$\mathbf{A}[11, 20] = \mathbf{A}_3[1, 1]\mathbf{A}_2[1, 2]\mathbf{A}_1[2, 0] = \frac{7}{6}.$$

Note that explicit storage of $\mathbf{A}$ requires space for 288 floating-point values (assuming full storage is used), while storage of matrices $\mathbf{A}_3, \mathbf{A}_2$ and $\mathbf{A}_1$ requires space for 22 floating-point values. Computing an element of $\mathbf{A}$ as needed requires 2 floating-point multiplications plus some integer arithmetic to compute the mixed-radix representations. Clearly, as the number of matrices involved in the Kronecker product increases, the memory savings and CPU overhead involved with computing elements as needed will both increase.

For the remainder of our work, we will consider Kronecker products of square matrices only. In particular, consider the Kronecker product of two identity matrices:

$$\mathbf{I}_n \otimes \mathbf{I}_m = \begin{bmatrix} \mathbf{I}_m & 0 & \cdots & 0 \\ 0 & \mathbf{I}_m & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \mathbf{I}_m \end{bmatrix}.$$

Thus the Kronecker product of identity matrices is an identity matrix whose size is the product of the sizes of the original matrices:

$$\mathbf{I}_{n_K} \otimes \cdots \otimes \mathbf{I}_{n_1} = \mathbf{I}_{\prod_{k=1}^{K} n_k}.$$

The Kronecker sum is defined [15, 96] in terms of Kronecker products involving identity

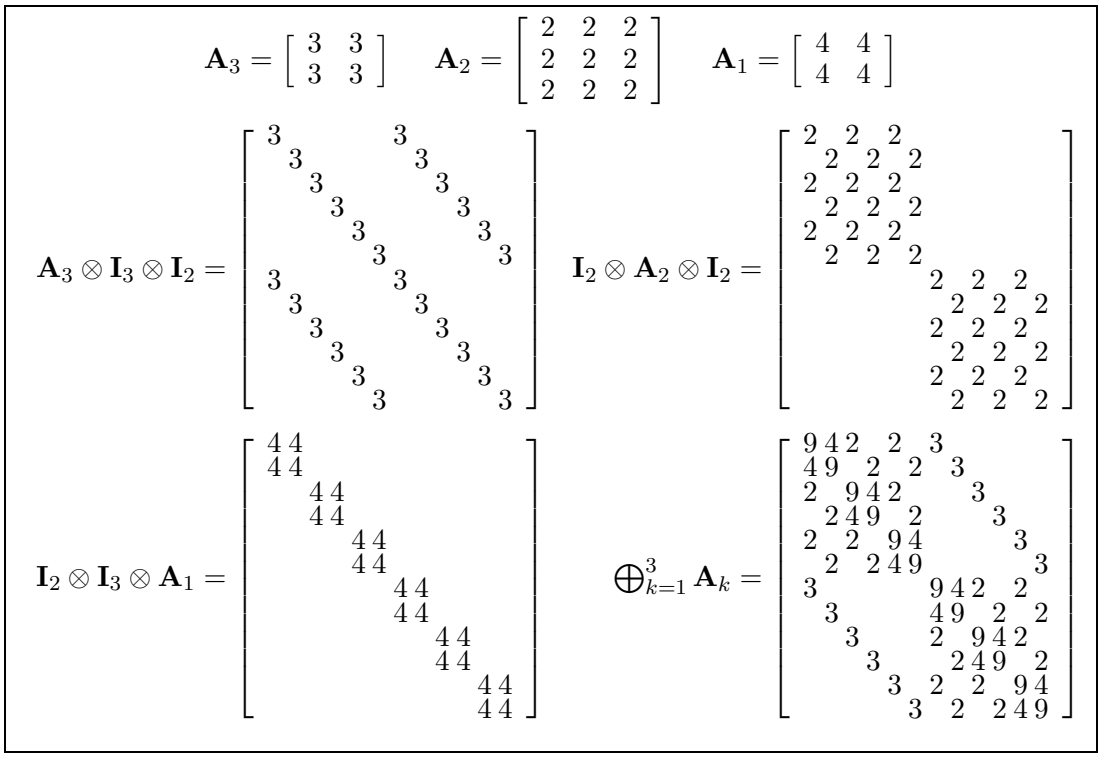


Figure 7.2: Example Kronecker sum

matrices as

$$\begin{aligned} \bigoplus_{k=1}^{K} \mathbf{A}_k &= \sum_{k=1}^{K} \mathbf{I}_{n_K} \otimes \cdots \otimes \mathbf{I}_{n_{k+1}} \otimes \mathbf{A}_k \otimes \mathbf{I}_{n_{k-1}} \otimes \cdots \otimes \mathbf{I}_{n_1} \\ &= \sum_{k=1}^{K} \mathbf{I}_{\prod_{i=k+1}^{K} n_i} \otimes \mathbf{A}_k \otimes \mathbf{I}_{\prod_{i=1}^{k-1} n_i} \end{aligned} \tag{7.2}$$

where matrix $\mathbf{A}_k$ is a square matrix of size n_k. An example showing the Kronecker sum of three small matrices is shown in Figure 7.2. The example shows each term of the sum from Equation 7.2 in addition to the complete sum.

7.2 Sparse Kronecker representations

In the previous section, we saw how the use of Kronecker operators allows us to represent large matrices implicitly, by storing much smaller matrices. In this section, we examine the effect of the choice of storage of the small matrices on complexity.

Looking at the example in Figure 7.1, we can see from a quick visual inspection that the matrix $\mathbf{A}$ is fairly sparse: more than half of its elements are zero. A zero element occurs when one of the elements of the small matrices is zero. In fact, the large blocks of all zeroes are caused by the two zero elements in matrix $\mathbf{A}_3$. In terms of memory requirements for this example, our choice of data structure for the small matrices (i.e., full, sparse by rows or sparse by columns) is not critical, since the small matrices will require very little storage space regardless of the data structure used. As the sizes of the $\mathbf{A}_k$ matrices grow, this choice will of course have greater impact.

As we saw in Chapter 2, using sparse storage can result in a dramatic decrease in the complexity of vector-matrix multiplications. This is especially true for a Kronecker representation. For instance, suppose we wish to multiply a vector by a matrix $\mathbf{A} = \mathbf{A}_K \otimes \cdots \otimes \mathbf{A}_1$, where $\mathbf{A} \in \mathbb{R}^{N \times N}$. For illustration, we will use a straightforward approach, in which each element of each column of $\mathbf{A}$ is explicitly computed and multiplied by the appropriate vector element. More sophisticated algorithms are discussed in detail in [15]. If each of the matrices $\mathbf{A}_k$ is stored in full, then the cost of a vector-matrix multiplication is exactly KN^2 floating-point multiplications. If instead we store the matrices $\mathbf{A}_k$ using column-wise sparse format, the cost of a vector-matrix multiplication is exactly $K\eta(\mathbf{A})$ floating-point multiplications. Thus, for every zero element in $\mathbf{A}$, we save K multiplications

if we use sparse storage for the small matrices.

Of course, our access requirements on $\mathbf{A}$ dictate how the small matrices $\mathbf{A}_K, \ldots, \mathbf{A}_1$ should be stored. If we require row (column) access to $\mathbf{A}$, we should store each matrix $\mathbf{A}_k$ by rows (columns). If we require both row and column access, we should use a sparse format that supports both row and column access for each matrix $\mathbf{A}_k$. Full storage should be used for each matrix $\mathbf{A}_k$ if we require access to specific elements of $\mathbf{A}$. For our work, we will require either row access or column access of $\mathbf{A}$. Thus, we will assume that each matrix $\mathbf{A}_k$ is stored using either row-wise or column-wise sparse storage.

7.3 Representing Q with Kronecker algebra

We assume that our model of interest is a *Kronecker-product-form* model consisting of K submodels. Without loss of generality, we assume that the local states $\mathcal{S}_k$ for each submodel k are sets of integers: $\mathcal{S}_k = \{0, \ldots, |\mathcal{S}_k| - 1\}$. This can be easily implemented by referring to the local state *indices* instead of the local states themselves. For each submodel k and each event e, we define [29] the matrix $\mathbf{W}_k^e$, a square matrix of size $\mathcal{S}_k$, as

$$\mathbf{W}_k^e[i_k, j_k] = \mathit{Rate}_k^e(i_k) \cdot \mathit{Prob}_k^e(i_k, j_k).$$

Note that elements in row i_k of $\mathbf{W}_k^e$ sum to $\mathit{Rate}_k^e(i_k)$. Also, note that $\mathbf{W}_k^e$ is the identity matrix if event e does not affect submodel k (i.e., $e \notin \mathcal{E}_k$). In particular, if e is a local event that affects only submodel k, then $\mathbf{W}_i^e$ is the identity for all i except $i = k$.

For *Kronecker-product-form* models, the matrix $\hat{\mathbf{R}}^e$ can be expressed as

$$\hat{\mathbf{R}}^e = \bigotimes_{k=K}^{1} \mathbf{W}_k^e, \tag{7.3}$$

where $\hat{\mathbf{R}}^e$ is the contribution of event e to the transition rate matrix of the underlying CTMC based on the potential states $\hat{\mathcal{S}}$. The transition rate matrix $\hat{\mathbf{R}}$ is then the sum of the $\hat{\mathbf{R}}^e$ matrices over all events e:

$$\hat{\mathbf{R}} = \sum_{e \in \mathcal{E}} \hat{\mathbf{R}}^e = \sum_{e \in \mathcal{E}} \bigotimes_{k=K}^{1} \mathbf{W}_k^e. \tag{7.4}$$

Looking at Equation 7.4, we see that a term of the summation corresponding to a local event e affecting submodel k becomes

$$\mathbf{I}_{|\mathcal{S}_K|} \otimes \cdots \otimes \mathbf{I}_{|\mathcal{S}_{k+1}|} \otimes \mathbf{W}_k^e \otimes \mathbf{I}_{|\mathcal{S}_{k-1}|} \otimes \cdots \otimes \mathbf{I}_{|\mathcal{S}_1|}.$$

Thus we can consider the synchronizing and local events in Equation 7.4 separately [29] as

$$\hat{\mathbf{R}} = \sum_{e \in \mathcal{E}_{synch}} \bigotimes_{k=K}^{1} \mathbf{W}_k^e + \bigoplus_{k=K}^{1} \mathbf{R}_k \tag{7.5}$$

where $\mathcal{E}_{synch}$ is the set of synchronizing events, and the matrix $\mathbf{R}_k$ is defined for each submodel k as

$$\mathbf{R}_k = \sum_{e \in \mathcal{E}_k \setminus \mathcal{E}_{synch}} \mathbf{W}_k^e.$$

Thus, $\mathbf{R}_k$ captures the behavior of the local events in submodel k.

For stationary analysis we need a representation for the matrix $\hat{\mathbf{Q}}$. Thus, in addition to the representation for $\hat{\mathbf{R}}$, we need a representation for the diagonal elements of $\hat{\mathbf{Q}}$ or (equivalently) for the row sums of $\hat{\mathbf{R}}$. This can be done either by storing the row sums explicitly in a full vector, or by encoding the diagonal of $\hat{\mathbf{Q}}$ as a second Kronecker expression involving the row sums of the $\mathbf{W}_k$ matrices [97]:

$$\mathit{RowSum}(\hat{\mathbf{R}}) = \mathit{RowSum}\left(\sum_{e \in \mathcal{E}} \bigotimes_{k=K}^{1} \mathbf{W}_k^e\right) = \sum_{e \in \mathcal{E}} \bigotimes_{k=K}^{1} \mathit{RowSum}(\mathbf{W}_k^e). \tag{7.6}$$

$Next_k^{Ta_k}$	$Next_k^{Tb_k}$	$Next_k^{Tc_k}$	$Next_k^{Td_k}$
$3 \to 1$	$1 \to 0$	$2 \to 1$	$0 \to 4$
$4 \to 2$	$5 \to 4$	$4 \to 3$	$1 \to 5$

Table 7.1: *Next* functions for the structured model of Figure 4.5

This is a vector Kronecker expression: each of the matrices involved consists of a single column. Of course, matrix $\hat{\mathbf{Q}}$ could also be represented as a single Kronecker expression. In practice, however, the diagonal of $\hat{\mathbf{Q}}$ is usually represented separately from the rest of the matrix; this is because iterative techniques such as Gauss-Seidel and Jacobi must explicitly access the diagonal elements.

As an example, we consider the structured Petri net of Figure 4.5, using the rates from Table 4.2. We show the *Next* functions in terms of the local state indices in Table 7.1, where local states are numbered according to Figure 4.3. The local $\mathbf{R}_k$ matrices for this model are shown in Figure 7.3. The $\mathbf{W}$ matrices are shown in Figure 7.4. We can compute the rate of transition from state $(1, 1, 2, 0)$ to state $(1, 1, 2, 4)$ using Equation 7.5, where the only contribution is from the Kronecker-sum term

$$\mathbf{I}_6[1, 1] \cdot \mathbf{I}_6[1, 1] \cdot \mathbf{I}_6[2, 2] \cdot \mathbf{R}_1[0, 4]$$

which equals 2.1. The rate of transition from state $(1, 1, 2, 4)$ to state $(0, 5, 2, 4)$ is given by the only non-zero term in the summation, which is the Kronecker-product term

$$\mathbf{W}_4^{Tsynch_{34}}[1, 0] \cdot \mathbf{W}_3^{Tsynch_{34}}[1, 5] \cdot \mathbf{I}_6[2, 2] \cdot \mathbf{I}_6[4, 4]$$

or $2 = (2.0)(1.0)$. It is important to note that this term is present for any pair of states

$$\mathbf{R}_4 = \begin{bmatrix} 0 & 0 & 0 & 0 & 2.6 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2.6 \\ 0 & 2.5 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2.5 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad \mathbf{R}_3 = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 2.3 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2.4 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2.4 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2.3 & 0 \end{bmatrix}$$

$$\mathbf{R}_2 = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1.7 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1.7 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad \mathbf{R}_1 = \begin{bmatrix} 0 & 0 & 0 & 0 & 2.1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2.5 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1.1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1.1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Figure 7.3: Local matrices for the structured model of Figure 4.5

$(1, 1, x, y)$ and $(0, 5, x, y)$, where $x, y \in \{0, \ldots, 5\}$. Of course, not all states $(1, 1, x, y)$ may be reachable; this issue will be discussed in the next section.

7.4 Kronecker overheads

As shown in the previous section, a very memory-efficient representation for $\hat{\mathbf{R}}$ is possible using Kronecker algebra. However, this compactness comes at a price. In this section we examine the CPU overheads connected with current Kronecker-based solution algorithms. These problems are studied in depth in [15]; we will provide a quick overview of the problems and then show in the next sections how they can be eliminated or alleviated by using our new data structures.

7.4.1 Overheads from using the potential states

As the Kronecker representation in Equation 7.5 describes a CTMC based on the *potential* states $\hat{\mathcal{S}}$, initial techniques [41, 84] analyzed these potential-based CTMCs. When this is

$$\mathbf{W}_2^{Tsynch_{12}} = \begin{bmatrix} 0&0&0&0&1.2&0 \\ 0&0&0&0&0&1.2 \\ 0&0&0&0&0&0 \\ 0&0&0&0&0&0 \\ 0&0&0&0&0&0 \\ 0&0&0&0&0&0 \end{bmatrix} \quad \mathbf{W}_1^{Tsynch_{12}} = \begin{bmatrix} 0&0&0&0&0&0 \\ 1.0&0&0&0&0&0 \\ 0&0&0&0&0&0 \\ 0&0&0&0&0&0 \\ 0&0&0&0&0&0 \\ 0&0&0&0&1.0&0 \end{bmatrix} \quad \begin{aligned} \mathbf{W}_4^{Tsynch_{12}} &= \mathbf{I}_6 \\ \mathbf{W}_3^{Tsynch_{12}} &= \mathbf{I}_6 \end{aligned}$$

$$\mathbf{W}_3^{Tb_2} = \begin{bmatrix} 1.5&0&0&0&0&0 \\ 0&2.5&0&0&0&0 \\ 0&0&2.5&0&0&0 \\ 0&0&0&1.5&0&0 \\ 0&0&0&0&1.5&0 \\ 0&0&0&0&0&2.5 \end{bmatrix} \quad \mathbf{W}_2^{Tb_2} = \begin{bmatrix} 0&0&0&0&0&0 \\ 1.0&0&0&0&0&0 \\ 0&0&0&0&0&0 \\ 0&0&0&0&0&0 \\ 0&0&0&0&0&0 \\ 0&0&0&0&1.0&0 \end{bmatrix} \quad \begin{aligned} \mathbf{W}_4^{Tb_2} &= \mathbf{I}_6 \\ \mathbf{W}_1^{Tb_2} &= \mathbf{I}_6 \end{aligned}$$

$$\mathbf{W}_3^{Tsynch_{23}} = \begin{bmatrix} 0&0&0&0&0&0 \\ 0&0&0&0&0&0 \\ 0&0&0&0&0&0 \\ 0&1.0&0&0&0&0 \\ 0&0&1.0&0&0&0 \\ 0&0&0&0&0&0 \end{bmatrix} \quad \mathbf{W}_2^{Tsynch_{23}} = \begin{bmatrix} 0&0&0&0&0&0 \\ 0&0&0&0&0&0 \\ 0&2.6&0&0&0&0 \\ 0&0&0&0&0&0 \\ 0&0&0&1.6&0&0 \\ 0&0&0&0&0&0 \end{bmatrix} \quad \begin{aligned} \mathbf{W}_4^{Tsynch_{23}} &= \mathbf{I}_6 \\ \mathbf{W}_1^{Tsynch_{23}} &= \mathbf{I}_6 \end{aligned}$$

$$\mathbf{W}_4^{Tsynch_{34}} = \begin{bmatrix} 0&0&0&0&0&0 \\ 2.0&0&0&0&0&0 \\ 0&0&0&0&0&0 \\ 0&0&0&0&0&0 \\ 0&0&0&0&0&0 \\ 0&0&0&0&1.5&0 \end{bmatrix} \quad \mathbf{W}_3^{Tsynch_{34}} = \begin{bmatrix} 0&0&0&0&1.4&0 \\ 0&0&0&0&0&1.0 \\ 0&0&0&0&0&0 \\ 0&0&0&0&0&0 \\ 0&0&0&0&0&0 \\ 0&0&0&0&0&0 \end{bmatrix} \quad \begin{aligned} \mathbf{W}_2^{Tsynch_{34}} &= \mathbf{I}_6 \\ \mathbf{W}_1^{Tsynch_{34}} &= \mathbf{I}_6 \end{aligned}$$

$$\mathbf{W}_4^{Tsynch_{41}} = \begin{bmatrix} 0&0&0&0&0&0 \\ 0&0&0&0&0&0 \\ 0&0&0&0&0&0 \\ 0&1.4&0&0&0&0 \\ 0&0&1.4&0&0&0 \\ 0&0&0&0&0&0 \end{bmatrix} \quad \mathbf{W}_1^{Tsynch_{41}} = \begin{bmatrix} 0&0&0&0&0&0 \\ 0&0&0&0&0&0 \\ 0&1.0&0&0&0&0 \\ 0&0&0&0&0&0 \\ 0&0&0&1.0&0&0 \\ 0&0&0&0&0&0 \end{bmatrix} \quad \begin{aligned} \mathbf{W}_3^{Tsynch_{41}} &= \mathbf{I}_6 \\ \mathbf{W}_2^{Tsynch_{41}} &= \mathbf{I}_6 \end{aligned}$$

Figure 7.4: Kronecker matrices for the structured model of Figure 4.5

done, storage of the state space is unnecessary; we need only store the local state spaces $\mathcal{S}_k$ (or $\hat{\mathcal{S}}_k$). However, these techniques compute the probability vector $\hat{\boldsymbol{\pi}}$, thus they require solution vectors of size $|\hat{\mathcal{S}}|$. As can be seen in Table 5.1, the number of potential states can be much larger than the number of actual states, resulting in a potentially fatal problem if

the memory requirements are excessive.

During computation of $\hat{\boldsymbol{\pi}}$, we must make sure that the probabilities associated with the unreachable states are zero. This can be done by using an initial probability vector with zero probabilities assigned to the unreachable states, and by skipping over the unreachable entries in $\hat{\mathbf{R}}$. This will guarantee that the solution vector will contain non-zero entries corresponding to the reachable states only. A common initial probability vector to use is a vector with a one corresponding to the initial state and zeroes corresponding to all other states.

Techniques that can access the matrix by rows (such as Power or Jacobi) do not need to worry about unreachable entries in a reachable row; this is because an unreachable state cannot be reached from a reachable state (see Equation 4.4). Thus, we only need to skip over the rows corresponding to unreachable states. This can be done by skipping a row whose corresponding entry in the iteration vector is zero. Alternatively, we can keep track of the set of indices with non-zero entries or we can generate $\mathcal{S}$ before the computation and retrieve the reachable indices during each iteration. These two alternatives are actually equivalent, since the set of indices with non-zero entries will eventually be identical to the set $\mathcal{S}$. Since testing each element of $\hat{\boldsymbol{\pi}}$ has an overall cost of $O(|\hat{\mathcal{S}}|)$, the latter technique is preferred when $\hat{\mathcal{S}}$ is much larger than $\mathcal{S}$.

Techniques that must access the matrix by columns (such as Gauss-Seidel) must skip over the unreachable columns; this is done in the same way that row-based techniques skip the unreachable rows. However, these column-access techniques introduce an additional difficulty: the presence of spurious entries in a reachable column due to unreachable states. These entries may arise because it is possible for an unreachable state to reach a reachable

state (see Equation 4.4). As long as the probabilities corresponding to the unreachable states remain zero in $\hat{\boldsymbol{\pi}}$ this problem is not too serious, since the spurious entries will be multiplied by zero. However, the complexity of the vector-matrix multiplication is affected by the number of spurious entries in the columns corresponding to the reachable states.

To summarize, the problems with a solution based on $\hat{\mathcal{S}}$ are the following.

- Vectors of size $|\hat{\mathcal{S}}|$ require more memory than vectors of size $|\mathcal{S}|$.
- We need to skip the unreachable rows / columns of $\hat{\mathbf{R}}$ while computing $\hat{\boldsymbol{\pi}}$.
- The cost of column-access vector-matrix multiplication is affected by spurious entries due to unreachable states.

In particular, the problem of excessive memory requirements motivates us to consider techniques that use solution vectors of size $|\mathcal{S}|$.

7.4.2 Overheads from using the actual states

When the number of potential states is substantially larger than the number of actual states, the memory required to store solution vectors of size $|\hat{\mathcal{S}}|$ can be excessive. Techniques based on $\mathcal{S}$ do not have this problem, as solution vectors of size $|\mathcal{S}|$ are used throughout the computation. Of course, these techniques must generate and store $\mathcal{S}$ before proceeding with numerical solution. Since we have knowledge of $\mathcal{S}$, skipping the unreachable rows or columns of $\hat{\mathbf{R}}$ is done by visiting each of the states in $\mathcal{S}$. Thus, techniques that use the actual states immediately address two of the problems of the potential-based techniques.

The main difficulty of using the actual states is that the Kronecker expression in Equation 7.5 encodes the matrix $\hat{\mathbf{R}}$, not $\mathbf{R}$. Thus, we need to transform "potential indices" when

dealing with $\hat{\mathbf{R}}$ into "actual indices" when dealing with $\boldsymbol{\pi}$. Every technique published so far [15, 56, 99] transforms these indices using some type of binary search(es) on the data structure used to store $\mathcal{S}$, thus introducing a logarithmic overhead. Techniques that access columns of $\hat{\mathbf{R}}$ usually eliminate the spurious entries when an attempt to transform the index fails. Thus, due to the logarithmic overhead, the spurious entries have an even worse effect on the complexity of the vector-matrix multiplication.

It has been shown [15] that the logarithmic overhead can be reduced from $O(\log_2 |\mathcal{S}|)$ to $O(\log_2 |\mathcal{S}_1|)$ using a sophisticated multiplication algorithm that "interleaves" the row and column components. Also, while each entry of $\mathbf{R}$ is conceptually a product of K real numbers, many entries share (sub)products of $2, \ldots, K-1$ of these numbers. An important advantage of the interleaving algorithm is that it can exploit these common subproducts, and thus reduce the number of floating-point multiplications required. However, the interleaving algorithm in [15] cannot be used with column-access techniques such as Gauss-Seidel. This means we must either use a slower-converging numerical solution algorithm or a less-efficient multiplication algorithm. Thus, as discussed in [15], we must choose between an algorithm with a larger number of iterations with a small per-iteration cost and an algorithm with fewer iterations but a higher per-iteration cost.

To summarize, the problems with a solution based on $\mathcal{S}$ are the following.

- The transformation of potential indices to actual indices introduces a logarithmic overhead.
- The spurious column entries (still) negatively affect performance.
- Algorithms that reduce the number of floating-point multiplications do not work with

column-access numerical solutions.

We will address these problems in the next sections.

7.5 Decision diagrams to store the state space

Our goal is to store the state space $\mathcal{S}$ in a data structure that does not require binary searches, so that searching for a given state does not incur a logarithmic overhead. We also must be able to compute the actual index for a reachable state using our data structure. Finally, given a reachable state, we must be able to quickly determine the "next" reachable state (i.e., the reachable state with actual index one higher than the current state) or that the current state is the "last" reachable state (i.e., the reachable state with actual index $|\mathcal{S}| - 1$).

The data structure we use is essentially an MDD whose nodes contain additional information. An example showing how to represent the set $\mathcal{S}$ for the structured Petri net of our running example (from Figure 4.5) is shown in Figure 7.5. Such a representation can be built either from the MDD generated using Algorithm 6.5, or from a multi-level structure generated using Algorithm 5.1. In the latter case, equivalent trees or arrays must be merged together.

7.5.1 State searches

To search for a given state in our data structure, we begin at the root node and follow the Full pointer for the current substate. If the Full pointer is not null, we follow the down pointer associated with the Full pointer, taking us to a node at the next level. This process

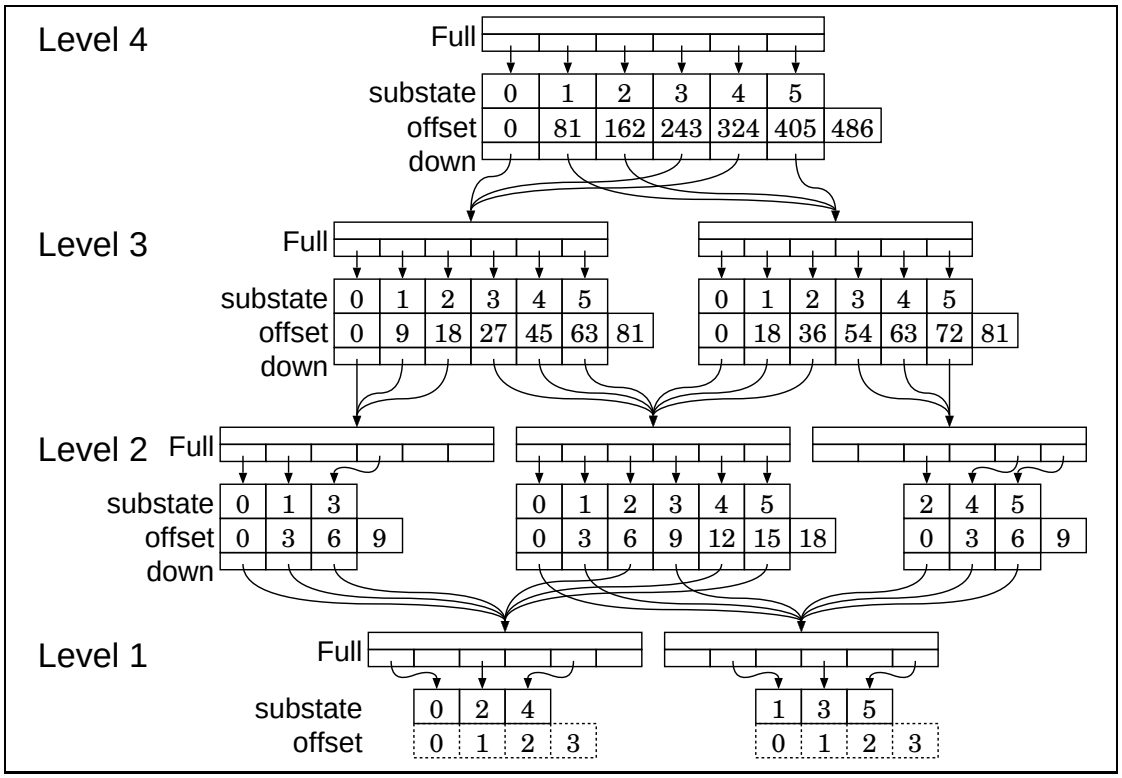


Figure 7.5: State space data structure

continues until we have encountered a null pointer, in which case the given state is not reachable, or we have followed the Full pointer at level one, in which case the given state is reachable. We do not need down pointers at level one. Since we have direct access to the Full and down pointers, the cost of a "search" at each level is constant. Thus the cost of a successful state search is always $O(K)$, and the cost of an unsuccessful state search is $O(K)$ in the worst case.

In our example of Figure 7.5, to search for state $(1, 1, 2, 4)$ we begin at the level-4 node and follow the pointer Full[1]. The pointer is not null, so we obtain the downward pointer to the second level-3 node. Following the downward pointer brings us to the level-3 node, where we follow Full[1] and then follow the downward pointer to the second level-2 node.

Next we follow Full[2] and follow the downward pointer to the first level-1 node. Since the pointer Full[4] at the level-1 node is not null, we conclude that state $(1, 1, 2, 4)$ is reachable. If we were to search for state $(0, 1, 2, 4)$ instead, we would find that the Full[2] pointer at the level-2 node is null; thus state $(0, 1, 2, 4)$ is not reachable. Note that in practice, we implement the Full arrays as arrays of integer offsets instead of arrays of pointers.

7.5.2 Computing state indices

To compute the actual index $\Psi(s)$ for state s, we simply take the sum of the offset values encountered during our search for the state. For instance, during the search for state $(1, 1, 2, 4)$, we encounter offsets of 81 at level 4, 18 at level 3, 6 at level 2, and 2 at level 1. Thus the actual index of state $(1, 1, 2, 4)$ is $107 = 81 + 18 + 6 + 2$.

The offset specifies the number of reachable states so far, not including the current substate. For instance, in our example of Figure 7.5 we see from the level-4 node that there are 81 states of the form $(0, \bullet, \bullet, \bullet)$, and there are 324 states of the form $(\{0, 1, 2, \text{or } 3\}, \bullet, \bullet, \bullet)$. Looking at the level-3 nodes we see that the number of states of the form $(x, 0, \bullet, \bullet)$ is 9 when x equals 0,3 or 4, and is 18 when x equals 1,2 or 5. We can also conclude that there are 27 states of the form $(4, \{0, 1, \text{or } 2\}, \bullet, \bullet)$. Note that the level-1 offset arrays always have the property that offset$[i] = i$; thus in practice we do not allocate offset arrays at level 1.

The final offset value of a node describes the total number of reachable states described by the node. For instance, in our example we see that the level-1 nodes each encode 3 (partial) states and the level-3 nodes each encode 81 (partial) states. Note that the level-4 node encodes 486 states; thus we have $|\mathcal{S}| = 486$. These "extra" offset values are not needed during the computation of state indices, but instead are used to compute the other offset

values in a bottom-up fashion during initialization. To compute the value of an element of offset for a level-k node when $k > 2$, we use

$$\mathsf{offset}[i+1] = \mathsf{offset}[i] + \#\text{ states encoded by node } \mathsf{down}[i]$$

which holds for all elements of offset (including the "extra" element) except for offset[0], which is always zero.

7.5.3 Determining the next reachable state

An important difference between the data structure presented here and the MDDs we used in the previous chapter is the sparse representation: except for array Full, information is stored only for the reachable substates. While this allows us to save some memory, its primary purpose is to allow us to quickly skip over unreachable states. Given any reachable state $(s_K, \ldots, s_1)$, we can easily determine the reachable state with actual index one higher than the current state. Using the sparse structure, we determine the next reachable substate s'_1 at level 1. If s'_1 exists, then $(s_K, \ldots, s_2, s'_1)$ is the next reachable state. When the substate at level k is the last reachable substate, we find the next reachable substate at level $k+1$. If the substates at all levels are the last reachable substates, then we know the current state is the last reachable state, with $\Psi((s_K, \ldots, s_1)) = |\mathcal{S}| - 1$. Otherwise, if we found the next reachable substate s'_k at level k, then we leave $s_K, \ldots, s_{k+1}$ unchanged, and find the first reachable substates $s'_{k-1}, \ldots s'_1$ at levels $k-1$ through 1. The state $(s_K, \ldots, s_{k+1}, s'_k, \ldots, s'_1)$ is then the next reachable state. Thus, this operation requires $2K-1$ steps in the worst case, and 1 step in the best (and most common) case, assuming we save pointers to the nodes and substates at each level.

Continuing our running example of Figure 7.5, suppose we know that state $(1, 1, 2, 2)$ has index 106, and we wish to determine the state with index 107. If we know that state $(1, 1, 2, 2)$ passes through the first node at level 1, we can simply find the next reachable substate after 2, which is 4. Thus state $(1, 1, 2, 4)$ has index 107. To determine the state with index 108, we look for the next reachable substate after 4, and find that there is none. Thus, we find the next reachable substate after 2 in the second level-2 node, and determine that it is substate 3. Then, we follow the downward pointer associated with substate 3 which brings us to the second level-1 node. Finally, we find the first reachable substate for that node, which is substate 1. Thus, the next reachable state is state $(1, 1, 3, 1)$, and it can be seen from the offsets that the index of this state is $108 = 81 + 18 + 9 + 0$. As a final example, we see that state $(5, 5, 5, 5)$ has index $485 = 405 + 72 + 6 + 2$; if we try to determine the next reachable state we will see that, at every level, the substates are the last reachable substates.

7.6 Matrix diagrams to store the transition rate matrix

We now introduce a new data structure for the storage of real matrices. Since our data structure borrows concepts from decision diagrams, we call it a *matrix diagram*. Like decision diagrams, a matrix diagram is represented as a K-level hierarchical structure. Each node of the matrix diagram is a matrix whose elements are sets of real values and pointers to nodes at the next level. The matrices at level 1 do not have downward pointers and the sets contain exactly one element; thus a level-1 matrix is equivalent to a real matrix. For instance, a level-2 matrix $\mathbf{A}_2$ might have $\mathbf{A}_2[2, 1] = \{(2.1, \mathbf{A}_1), (1.1, \mathbf{A}'_1)\}$, where $\mathbf{A}_1$

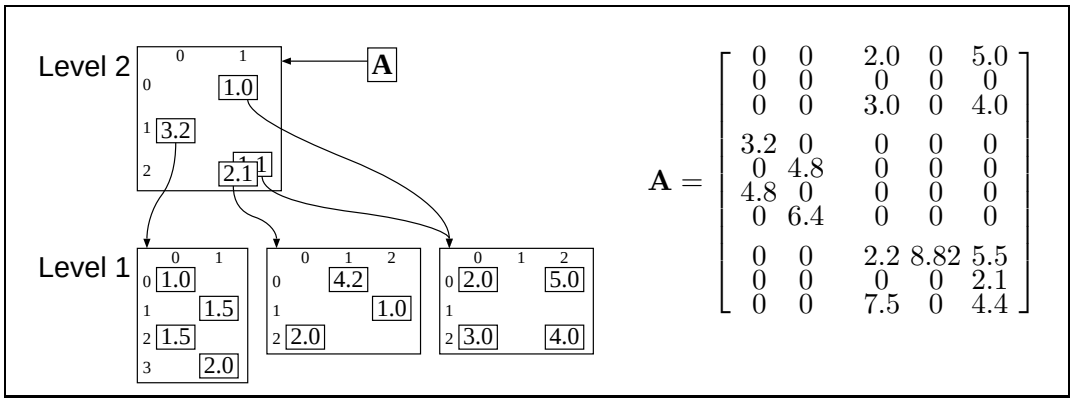


Figure 7.6: An example matrix diagram and the matrix it represents

and $\mathbf{A}'_1$ are level-1 matrices. The elements of the matrix represented by a matrix diagram can be computed by

$$\begin{aligned} &\mathit{Element}(\mathbf{A}_k, (r_k, \ldots, r_1), (c_k, \ldots, c_1)) = \\ &\qquad \sum_{(x, \mathbf{A}_{k-1}) \in \mathbf{A}_k[r_k, c_k]} x \cdot \mathit{Element}(\mathbf{A}_{k-1}, (r_{k-1}, \ldots, r_1), (c_{k-1}, \ldots, c_1)) \end{aligned} \tag{7.7}$$

with terminal condition $\mathit{Element}(\mathbf{A}_1, r_1, c_1) = \mathbf{A}_1[r_1, c_1]$.

An example matrix diagram is shown in Figure 7.6. Each matrix element is either the empty set, which is indicated by blank space, or a set of elements, which is indicated by stacked boxes. The boxes contain the real value and have attached the downward pointer to the matrix at the next level. At level 1, zeroes are represented by blank space and there are no downward pointers. If the matrix diagram in Figure 7.6 represents matrix $\mathbf{A}$, we can compute elements $\mathbf{A}[(1,1),(0,1)]$ as $(3.2)(1.5)$, and $\mathbf{A}[(2,2),(1,0)]$ as $(2.1)(2.0)+(1.1)(3.0)$. Note that the sizes of the matrices of a given level are allowed to vary. The matrix sizes are restricted so that the *Element* computation in Equation 7.7 is well-defined. More formally, we say a level-k matrix $\mathbf{A}_k$ is on *path* $[(r_K, \ldots, r_{k+1}), (c_K, \ldots, c_{k+1})]$ if there exist matrices

$\mathbf{A}_K, \ldots, \mathbf{A}_{k+1}$ at levels K through $(k+1)$ such that

$$\begin{aligned} \exists\ (x_K, \mathbf{A}_{K-1}) &\in \mathbf{A}_K[r_K, c_K], \\ &\vdots \\ \exists\ (x_{k+1}, \mathbf{A}) &\in \mathbf{A}_{k+1}[r_{k+1}, c_{k+1}] \end{aligned}$$

which says that $\mathbf{A}_k$ is used when computing $\mathit{Element}(\mathbf{A_K}, (r_K, \ldots, r_1), (c_K, \ldots, c_1))$. For instance, looking at Figure 7.6 we see that the first level-1 matrix is on path $[(1), (0)]$ and the third level-1 matrix is on paths $[(0), (1)]$ and $[(2), (1)]$. Matrices that share row (column) paths must have the same number of rows (columns). That is, if level-k matrix $\mathbf{A}_k$ is on path $[(r_K, \ldots, r_{k+1}), (c_K, \ldots, c_{k+1})]$ and matrix $\mathbf{A}'_k$ is on path $[(r_K, \ldots, r_{k+1}), (c'_K, \ldots, c'_{k+1})]$ then matrices $\mathbf{A}_k$ and $\mathbf{A}'_k$ must have the same number of rows. A similar argument can be made for the number of columns. Thus, if two matrices are on the same path then they must be the same size.

7.6.1 Kronecker products with matrix diagrams

A matrix diagram representing the Kronecker product of two matrices $\mathbf{A}$ and $\mathbf{B}$ is a simple 2-level matrix diagram with a single matrix per level. The level-1 matrix has sets whose elements correspond to the elements of $\mathbf{B}$, except that the zero elements of $\mathbf{B}$ are represented by empty sets in the matrix diagram. Similarly, the level-2 matrix has sets whose elements correspond to the non-zero elements of $\mathbf{A}$. The downward pointers associated with the level-2 elements all point to the level-1 matrix. This process can be extended to the Kronecker product of K matrices; this is shown in detail in Algorithm 7.1. Note that, in contrast to the decision diagram algorithms in the previous chapter, this algorithm is not recursive.

```
KronProd(A_K, ..., A_1)        • A_K, ..., A_1 are real matrices
 1: for k ← 1 to K do
 2:   for each row r of A_k do
 3:     for each column c of A_k do
 4:       if A_k[r,c] ≠ 0 then
 5:         if K > 1 then
 6:           B_k[r,c] ← {(A_k[r,c], B_{k-1})}
 7:         else
 8:           B_k[r,c] ← {A_k[r,c]}
 9:         end if
10:       else
11:         B_k[r,c] ← ∅
12:       end if
13:     end for
14:   end for
15:   B_k ←UniqueTableInsert(B_k)
16: end for
17: return B_K
```

Algorithm 7.1: Building a matrix diagram of a Kronecker product

Like the decision diagram algorithms, we use a table of unique matrices to keep the number of matrices in the matrix diagram small: line 15 of Algorithm 7.1 makes sure that duplicate matrices are not created. When the cost of line 15 is constant, the cost of building a matrix diagram representation of a Kronecker product is $O(\eta(\mathbf{A_K}) + \cdots + \eta(\mathbf{A_1}))$, assuming sparse storage is used for all matrices.

An example showing the matrix diagram representation of the Kronecker product of Figure 7.1 is shown in Figure 7.7. In the example, we have the Kronecker product of the three matrices $\mathbf{A}_3$, $\mathbf{A}_2$, and $\mathbf{A}_1$; thus we have a 3-level matrix diagram. Note that the empty sets in the matrix diagram correspond to the zero elements in $\mathbf{A}_3$ and $\mathbf{A}_1$.

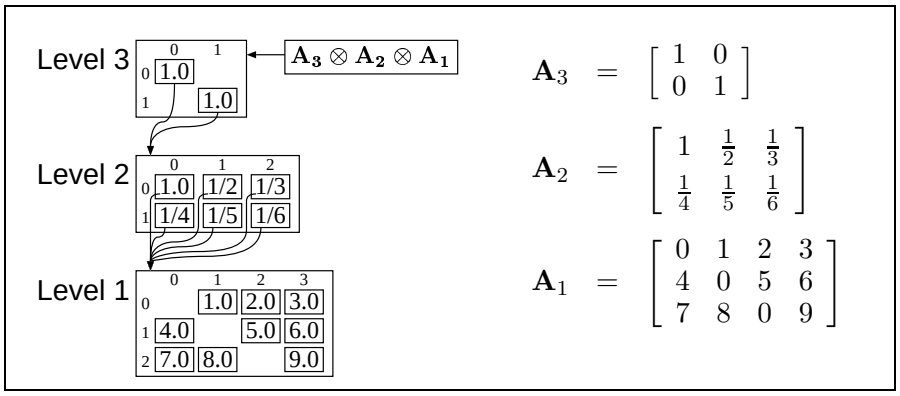


Figure 7.7: Matrix diagram of the Kronecker product of Figure 7.1

7.6.2 Addition of matrix diagrams

The addition of two K-level matrix diagrams representing matrices of the same size is done by taking the union of the elements of the level-K matrices. While this produces a proper matrix diagram, we can apply reductions in certain cases. The first special case arises when a set contains elements $(a_1, \mathbf{A})$ and $(a_2, \mathbf{A})$; in this case we can replace those elements with the single element $(a_1 + a_2, \mathbf{A})$. This reduction is equivalent to the transformation

$$a_1\mathbf{A} + a_2\mathbf{A} \Rightarrow (a_1 + a_2)\mathbf{A}, \tag{7.8}$$

where $\mathbf{A}$ is an ordinary real matrix and a_1 and a_2 are scalars. Note that the reduction corresponding to Equation 7.8 always reduces both CPU and memory requirements. The second special case arises when a set contains elements $(a, \mathbf{A}_1)$ and $(a, \mathbf{A}_2)$; in this case we can replace those elements with the single element $(a, \mathbf{A}_1 + \mathbf{A}_2)$, where $\mathbf{A}_1 + \mathbf{A}_2$ is the addition operator applied to the $(K-1)$-level matrix diagrams $\mathbf{A}_1$ and $\mathbf{A}_2$. This reduction

is equivalent to the transformation

$$a\mathbf{A}_1 + a\mathbf{A}_2 \Rightarrow a(\mathbf{A}_1 + \mathbf{A}_2), \tag{7.9}$$

where $\mathbf{A}_1$ and $\mathbf{A}_2$ are real matrices and a is a scalar. In contrast to the first case, the reduction corresponding to Equation 7.9 always reduces CPU requirements but will increase memory requirements if we must continue to store $\mathbf{A}_1$ and $\mathbf{A}_2$ because there are other downward pointers to them.

The matrix diagram addition operator where the reductions of Equation 7.8 and Equation 7.9 are performed whenever possible is shown in Algorithm 7.2. This algorithm, like most decision diagram algorithms, is recursive. As with other recursive decision diagram algorithms, we maintain a cache to eliminate duplication of work. If an operation has already been performed, it is detected immediately (lines 1-3) and the computed result is returned. Note that the recursion is necessary only to implement the reduction of Equation 7.9. Without the reductions, addition of matrix diagrams $\mathbf{A}_K$ and $\mathbf{B}_K$ has a cost of $O(|\mathbf{A}_K|+|\mathbf{B}_K|)$, where $|\mathbf{A}_K|$ is the sum of the set sizes over all elements of $\mathbf{A}_K$. The cost of implementing the reductions introduces a factor based on the time required to detect that the reductions can be performed. The second reduction may generate additional recursive calls to Add; at worst we generate a call to Add for every pair of matrices at the same level. An example for the addition operator is shown in Figure 7.8. The resulting matrix diagram is computed using Algorithm 7.2 with both reductions.

```
Add(k, A, B)        • A and B are level-k matrix diagrams
 1: if cache contains entry for (k, A, B) then
 2:    return cache entry result
 3: end if
 4: if k = 1 then
 5:    C ← A + B                    • Level 1 matrices are "normal" matrices
 6:    C ←UniqueTableInsert(C)
 7:    add [(k, A, B), C] to cache
 8:    return C
 9: end if
10: C ← A
11: for each row r of B do
12:    for each column c of B do
13:       for each (b, B') ∈ B[r, c] do
14:          if ∃(a, B') ∈ C[r, c] then                    • Try Equation 7.8
15:             C[r, c] ← (C[r, c] \ {(a, B')}) ∪ {(a + b, B')}
16:          else if ∃(b, A') ∈ C[r, c] then               • Try Equation 7.9
17:             C[r, c] ← (C[r, c] \ {(b, A')}) ∪ {(b, Add(k − 1, A', B'))}
18:          else
19:             C[r, c] ← C[r, c] ∪ {(b, B')}
20:          end if
21:       end for
22:    end for
23: end for
24: C ←UniqueTableInsert(C)
25: add [(k, A, B), C] to cache
26: return C
```

Algorithm 7.2: Adding two matrix diagrams

7.6.3 Submatrix selection with matrix diagrams

Another important operation for matrix diagrams is the Submatrix operator, which returns a matrix diagram containing only the selected rows and columns of the initial matrix diagram. That is, the matrix returned by Submatrix$(\mathbf{A}, \mathcal{R}, \mathcal{C})$ contains element $[(r_K, \ldots, r_1), (c_K, \ldots, c_1)]$ if $(r_K, \ldots, r_1) \in \mathcal{R}$ and $(c_K, \ldots, c_1) \in \mathcal{C}$. This operation can be efficiently performed on a K-level matrix diagram if K-variable MDDs are used to represent the char-

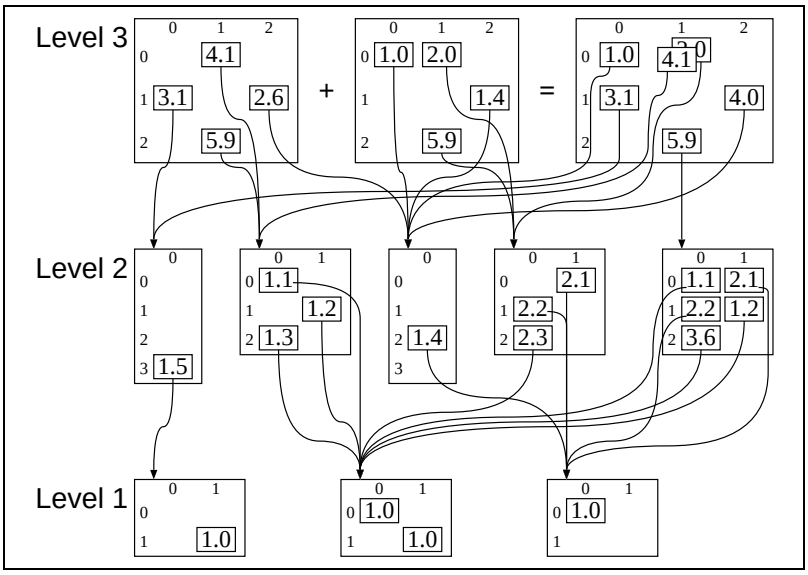


Figure 7.8: Example of matrix diagram addition

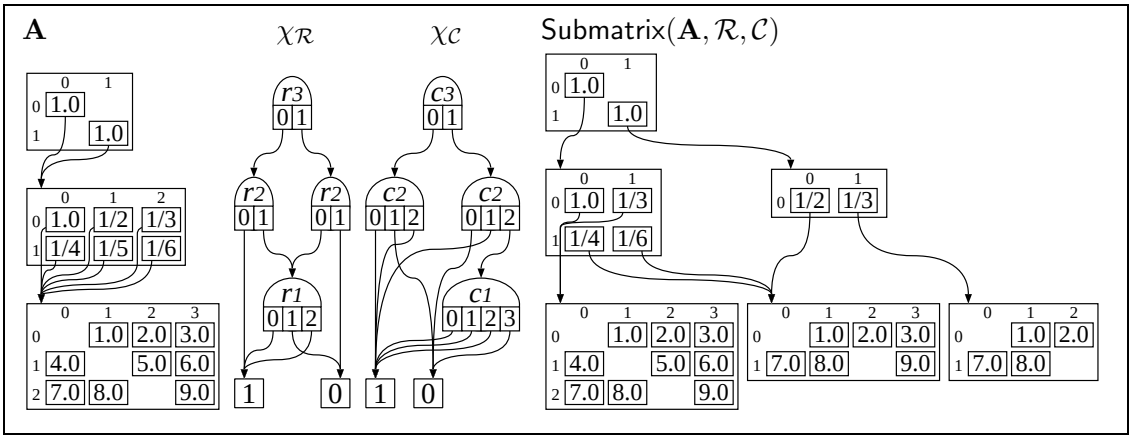


Figure 7.9: Example of a matrix diagram representing a submatrix

acteristic functions for sets $\mathcal{R}$ and $\mathcal{C}$. Algorithm 7.3 shows how to recursively compute a matrix diagram submatrix. Note that the level-k matrix $\mathbf{B}$ returned by Submatrix may have a different size than the level-k matrix $\mathbf{A}$. An example showing the matrix diagram computed by Submatrix is shown in Figure 7.9. As the example shows, a single matrix in the

```
Submatrix(k, A, R, C)        • A is a level-k matrix, R and C are MDDs
 1: if (R ≡ 1) and (C ≡ 1) then
 2:    return A
 3: end if
 4: if cache contains entry for (k, A, R, C) then
 5:    return cache entry result
 6: end if
 7: b_row ← 0
 8: for each row i of A do
 9:    if R_{x_k=i} ≢ 0 then
10:       b_col ← 0
11:       for each column j of A do
12:          if C_{x_k=j} ≢ 0 then
13:             if k > 1 then
14:                for each (x, D) ∈ A[i, j] do
15:                   D' ← Submatrix(k − 1, D, R_{x_k=i}, C_{x_k=j})
16:                   if D' ≠ 0 then
17:                      B[b_row, b_col] ← B[b_row, b_col] ∪ {(x, D')}
18:                   end if
19:                end for
20:             else                                  • No downward pointers
21:                B[b_row, b_col] ← A[i, j]
22:             end if
23:             b_col ← b_col + 1
24:          end if
25:       end for
26:       b_row ← b_row + 1
27:    end if
28: end for
29: B ←UniqueTableInsert(B)
30: add [(k, A, R, C), B] to cache
31: return B
```

Algorithm 7.3: Computing a submatrix using matrix diagrams

original matrix diagram may produce multiple matrices in the matrix diagram representing a submatrix. This is due to differences between the shared nodes in the matrix diagram and the MDDs for row and column selection. For instance, in the example we see that the level-3 node of $\chi_{\mathcal{R}}$ has different downward pointers for $r_3 = 0$ and $r_3 = 1$; thus different

rows are selected based on the value of r_3. As we expect, in the submatrix representation, downward pointers in the level-3 matrix are different for different rows.

When we select a submatrix, we must decide if we should renumber the rows and columns. For instance, if we originally have a 3×3 matrix and we remove column 1, we must decide if column 2 should remain column 2, or be renumbered as column 1. In the example of Figure 7.9, the rows and columns have all been renumbered. When rows and columns are renumbered, the potential for "merging" increases. For instance, selecting rows $\{a, b\}$ and columns $\{a, b\}$ of $\mathbf{I}_8$ produces $\mathbf{I}_2$ for any distinct pair (a, b), if the rows and columns are renumbered. However, when renumbering occurs, we must be able to convert from the original indexing to the renumbered indexing. For instance, in the 3×3 matrix, elements that were indexed by column 2 must now be indexed by column 1 if the columns are renumbered. This conversion is possible using the MDDs for the selected rows and columns. An example of this will be given later.

7.6.4 Representing R with matrix diagrams

With the above operations, we can compute a matrix diagram representation for the transition rate matrix $\mathbf{R}$. First, we build the Kronecker matrices $\mathbf{W}_k^e$. Then, we build a matrix diagram representation for $\hat{\mathbf{R}}$ based on Equation 7.4, using Algorithm 7.1 to do the Kronecker products and Algorithm 7.2 to do the sums. Finally, we compute $\mathbf{R} = \hat{\mathbf{R}}[\mathcal{S}, \mathcal{S}]$ using Algorithm 7.3 to compute $\mathsf{Submatrix}(K, \hat{\mathbf{R}}, \mathcal{S}, \mathcal{S})$.

The final matrix diagram representation of $\mathbf{R}$ for the running Petri net example (from Figure 4.5) is shown in Figure 7.10. In the figure, the rows and columns were all renumbered after the submatrix operation. Since the state space for this model contains 486 states, the

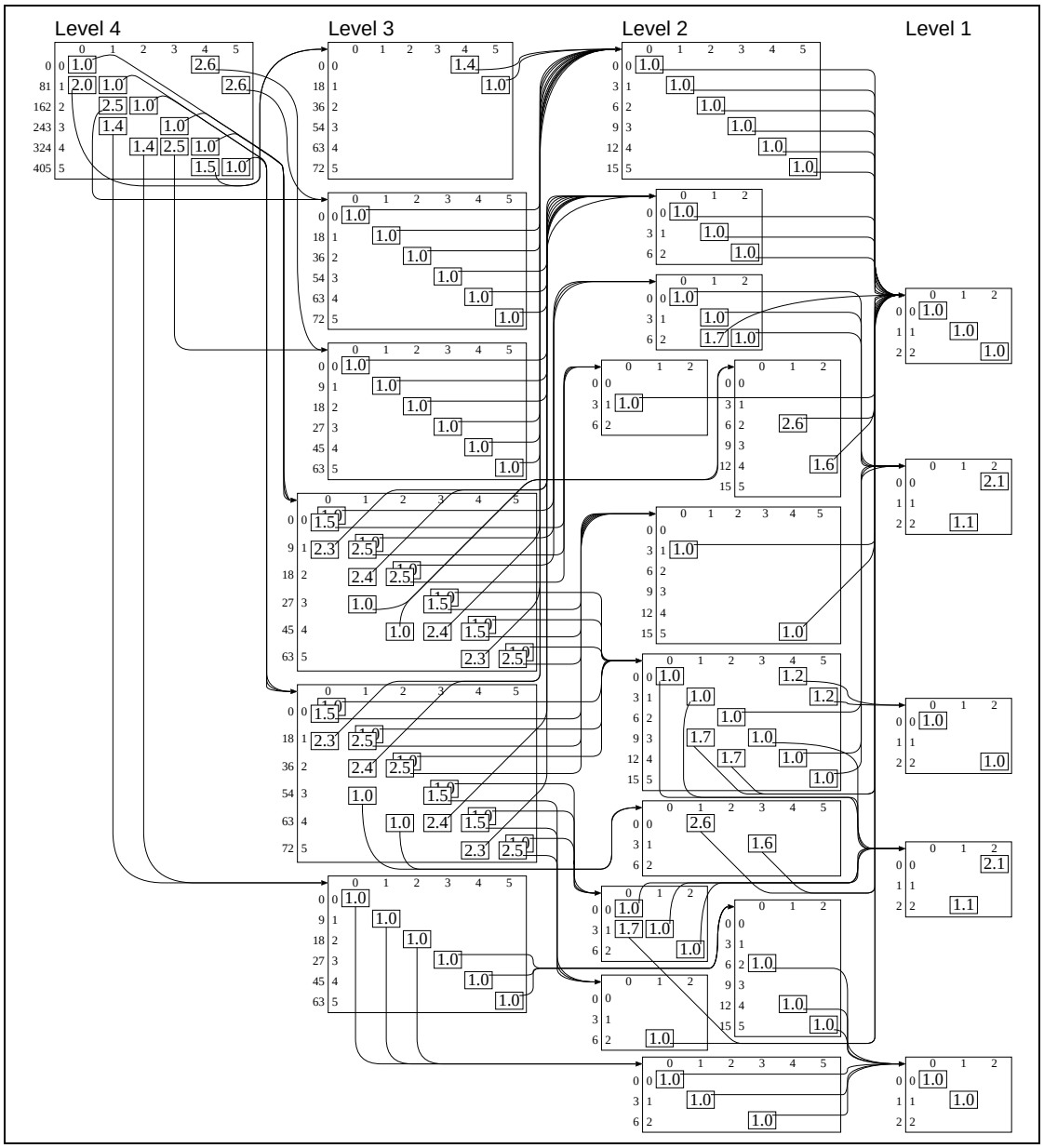


Figure 7.10: Matrix diagram for the structured model of Figure 4.5

transition rate matrix $\mathbf{R}$ represented by the matrix diagram in Figure 4.5 is of size 486×486. The numbers to the left of the matrices are the *row offsets*. These are analogous to the offsets used in decision diagrams: the actual index of a row is computed by summing the

row offsets. As with decision diagram offsets, the matrix diagram row offsets do not need to be explicitly stored at the last level.

As we did with the Kronecker representation, we can compute the rate of transition between (potential) states $(1, 1, 2, 4)$ and $(0, 5, 2, 4)$. However, if the rows and columns were renumbered when the submatrix was selected, we must convert the *potential* states into their *actual* equivalents. This can be viewed as the difference between the indexing of the potential matrix $\hat{\mathbf{R}}$ and the actual matrix $\mathbf{R}$ represented by the matrix diagram. The conversion is done by replacing each substate with its "reachable substate index", which can be determined from Figure 7.5: substate 1 is in position 1 of the substate array at level 4, substate 1 is in position 1 of the substate array at level 3, substate 2 is in position 2 of the substate array at level 2, and substate 4 is in position 2 of the substate array at level 1. Thus potential state $(1, 1, 2, 4)$ corresponds to actual state $[1, 1, 2, 2]$, and similarly we can determine that potential state $(0, 5, 2, 4)$ corresponds to actual state $[0, 5, 2, 2]$. Looking at the matrix diagram in Figure 7.10, we see that the transition from $[1, 1, 2, 2]$ to $[0, 5, 2, 2]$ is $(2.0)(1.0)(1.0)(1.0)$, exactly the same as the Kronecker case. However, recall that the rate from $(1, 1, x, y)$ to $(0, 5, x, y)$ is 2.0 according to the Kronecker expression, even for unreachable states $(1, 1, x, y)$ and $(0, 5, x, y)$. In the matrix diagram, there are entries only for the reachable states.

7.6.5 Matrix diagram column access

A critical requirement for numerical solution using Gauss-Seidel is efficient access to a given column of the transition rate matrix $\mathbf{R}$. We can efficiently access a column of a matrix diagram if the data structures used to store the matrices at each level allow for

```
GetColumn(k, A, (c_k, ..., c_1))
 • A is a level-k matrix
 • c_k, ..., c_1 are column indices to select at this level and below
 1: if k = 1 then                                  • Bottom level, return the column
 2:    return A[•, c_1]
 3: end if
 4: column ← 0                                     • column is a sparse vector
 5: for each non-empty row r_k in column c_k of A do
 6:    for each element (x, D) ∈ A[r_k, c_k] do
 7:       down ← GetColumn(k − 1, D, (c_{k−1}, ..., c_1))
 8:       down ← x · down
 9:       shift indices of down by the offset of row r_k
10:       column ← column + down
11:    end for
12: end for
13: return column
```

Algorithm 7.4: Obtaining a matrix diagram column

efficient access to the non-zero elements of a given column. Algorithm 7.4 shows how to recursively obtain a specific column of a matrix that is represented as a matrix diagram. If the specified matrix is at the bottom level (level 1) of the matrix diagram, we can simply return the selected column (line 2). Otherwise, for each element of the selected column of the level-k matrix, we follow the downward pointer and obtain the appropriate column of this level-$(k-1)$ matrix (line 7). The downward column is then scaled by the value of the current element (line 8) and shifted by the row offset of the current row (line 9). The scaling computes the product of Equation 7.7, and the shifting computes the row indices for the elements. The sum of these adjusted downward columns gives the selected column.

Continuing our running example, we can compute the column corresponding to actual state $[0, 5, 2, 2]$ as shown in Figure 7.12, where the "interesting" portion of the matrix diagram is shown in Figure 7.11. Matrices are indicated by their ordinal number, counting

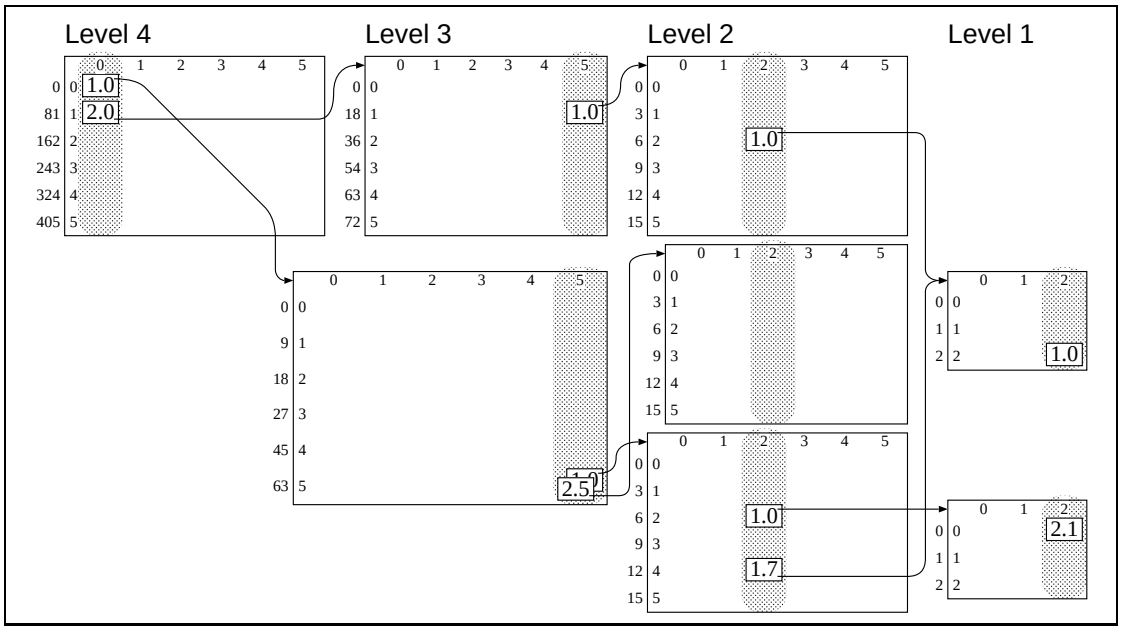


Figure 7.11: Interesting portion of the matrix diagram in Figure 7.10

from the top down; for instance, GetColumn$(2, \#2, \ldots)$ refers to the middle matrix at level 2. The column is computed by calling GetColumn$(4, \#1, [0, 5, 2, 2])$. We see that the column contains three entries: 2.1 at row 69, 1.7 at row 77, and 2.0 at row 107. These represent the transitions to potential state $(0, 5, 2, 4)$ from potential states $(0, 5, 2, 0)$, $(0, 5, 4, 4)$ and $(1, 1, 2, 4)$ via events Td_1, Ta_2, and $Tsynch_{34}$, respectively.

If we were to compute the column corresponding to actual state $[4, 5, 2, 2]$, we would find ourselves repeating much of the computation that we did for column $[0, 5, 2, 2]$; in particular, the calls to GetColumn$(3, \#1, [5, 2, 2])$ and GetColumn$(3, \#2, [5, 2, 2])$ from Figure 7.12 will again be generated. Since these intermediate results are the same as for our previous column access, we can avoid duplicate computation by using a cache of recent operations as we did for other MDD and matrix diagram algorithms. The results of a GetColumn computation can then be re-used within a single column access and over multiple column accesses.

Level 4

$$\mathsf{GetColumn}(4, \#1, [0,5,2,2])$$
$$= \mathsf{GetColumn}(3, \#2, [5,2,2]) \left\langle \begin{matrix}\text{scale } 1.0\\ \text{shift } 0\end{matrix} \right\rangle + \mathsf{GetColumn}(3, \#1, [5,2,2]) \left\langle \begin{matrix}\text{scale } 2.0\\ \text{shift } 81\end{matrix} \right\rangle$$
$$= \begin{bmatrix}\text{value} & 2.1 & 1.7 & 2.0\\ \text{row} & 69 & 77 & 107\end{bmatrix}$$

Level 3

$$\mathsf{GetColumn}(3, \#1, [5,2,2])$$
$$= \mathsf{GetColumn}(2, \#1, [2,2]) \left\langle \begin{matrix}\text{scale } 1.0\\ \text{shift } 18\end{matrix} \right\rangle$$
$$= \begin{bmatrix}\text{value} & 1.0\\ \text{row} & 26\end{bmatrix}$$
$$\mathsf{GetColumn}(3, \#2, [5,2,2])$$
$$= \mathsf{GetColumn}(2, \#2, [2,2]) \left\langle \begin{matrix}\text{scale } 2.5\\ \text{shift } 63\end{matrix} \right\rangle + \mathsf{GetColumn}(2, \#3, [2,2]) \left\langle \begin{matrix}\text{scale } 1.0\\ \text{shift } 63\end{matrix} \right\rangle$$
$$= \begin{bmatrix}\text{value} & 2.1 & 1.7\\ \text{row} & 69 & 77\end{bmatrix}$$

Level 2

$$\mathsf{GetColumn}(2, \#1, [2,2])$$
$$= \mathsf{GetColumn}(1, \#1, [2]) \left\langle \begin{matrix}\text{scale } 1.0\\ \text{shift } 6\end{matrix} \right\rangle$$
$$= \begin{bmatrix}\text{value} & 1.0\\ \text{row} & 8\end{bmatrix}$$
$$\mathsf{GetColumn}(2, \#2, [2,2]) = \mathbf{0}$$
$$\mathsf{GetColumn}(2, \#3, [2,2])$$
$$= \mathsf{GetColumn}(1, \#2, [2]) \left\langle \begin{matrix}\text{scale } 1.0\\ \text{shift } 6\end{matrix} \right\rangle + \mathsf{GetColumn}(1, \#1, [2]) \left\langle \begin{matrix}\text{scale } 1.7\\ \text{shift } 12\end{matrix} \right\rangle$$
$$= \begin{bmatrix}\text{value} & 2.1 & 1.7\\ \text{row} & 6 & 14\end{bmatrix}$$

Level 1

$$\mathsf{GetColumn}(1, \#1, [2]) = \begin{bmatrix}\text{value} & 1.0\\ \text{row} & 2\end{bmatrix}$$
$$\mathsf{GetColumn}(1, \#2, [2]) = \begin{bmatrix}\text{value} & 2.1\\ \text{row} & 0\end{bmatrix}$$

Figure 7.12: A trace of Algorithm 7.4

Of course, we do not want to store columns in the cache forever; doing so would be equivalent to storing the entire matrix $\mathbf{R}$ once we have accessed every column. Instead, we will implement our cache by saving only the most recently generated column for each matrix.

Thus, for each matrix in the matrix diagram, we require a single vector. Fortunately, these vectors are extremely sparse in practice (as can be seen in the example), and we only need to store their non-zero elements. Bounds for the sizes of these vectors can be computed *a priori*, by determining the maximum number of non-zero elements possible for each column in a bottom-up fashion. Then, we allocate a vector **column** of the appropriate size for each matrix before starting the numerical iterations, and use **column** whenever a GetColumn computation is performed on that matrix.

We *clear* the cache at level k by setting all the **column** vectors at level k to a null value. This must be done whenever the specified column at level k or below changes. For instance, if we just obtained column $[0, 5, 2, 2]$, and now we wish to obtain column $[0, 4, 2, 2]$, we must clear the cache at levels 4 and 3. Thus, to maximize our possible cache hits, after visiting the reachable column for state $[c_K, \ldots, c_1]$, we should visit the reachable column $[c'_K, \ldots, c'_k, c_{k-1}, \ldots, c_1]$ for k as large as possible. This is equivalent to the state that follows $[c_K, \ldots, c_1]$ in lexicographic order *when the string is read in reverse*. To implement this second "upside-down" order, we generate a second, upside-down copy of the state space which is stored in a decision diagram $\mathcal{U}$. For any state $(s_K, \ldots, s_1)$ belonging to $\mathcal{S}$, state $(s_1, \ldots, s_K)$ belongs to $\mathcal{U}$ and vice-versa.

Thus, given a reachable potential state $(c_K, \ldots, c_1)$ whose column we just computed, we compute the "next" column by performing the following steps.

1. Determine the potential state $(c'_1, \ldots, c'_K)$ whose actual index is one higher than $(c_1, \ldots, c_K)$ according to $\mathcal{U}$.

2. Convert the potential state $(c'_K, \ldots, c'_1)$ into the actual state $[a'_K, \ldots, a'_1]$ according to

$\mathcal{S}$.

3. Compute the index Ψ of $[a'_K, \ldots, a'_1]$ according to $\mathcal{S}$.

4. Find the smallest k such that $a'_k \neq a_k$, where $[a_K, \ldots, a_1]$ is the previous actual state. Clear the cache at levels K through k.

5. Obtain column Ψ by calling $\mathsf{GetColumn}(K, R, [a'_K, \ldots, a'_1])$, where R is the top matrix of the matrix diagram representation for $\mathbf{R}$.

Recall that step 1 has a cost of $O(1)$ on average and $O(K)$ in the worst case. Step 2 is required only if the rows and columns are renumbered when the submatrix is selected. However, step 3 is always required, and steps 2 and 3 can be performed simultaneously with a fixed cost of $O(K)$. Thus, in our case, renumbering the rows and columns does not introduce additional overheads, and can potentially reduce the size of the resulting data structure. The cost of clearing the cache has a cost of $O(K)$ to determine which levels to clear, plus the cost of visiting the matrices at levels K through k. Usually, only the cache at level K needs to be cleared (which has a cost of $O(1)$), but in the worst case the cache at every level needs to be cleared, which requires visiting every matrix in the matrix diagram.

Of course, for numerical solution we must be able to multiply a row vector by a specific matrix column. In contrast to the Kronecker techniques, which essentially perform the multiplication while computing the matrix column, our technique requires the computation of a column as a separate step. Once the column has been obtained, we then perform the dot product of the row vector by the selected column. Since the selected column $\mathbf{c}$ is represented using sparse storage, the dot product requires only $O(\eta(\mathbf{c}))$ operations.

7.7 Experimental results

We test our techniques on the Kanban and FMS models described in Appendix B. The models were run on a 400Mhz Pentium-II workstation with 384Mb of main memory, under the Linux operating system. We compute the stationary probability vector using Gauss-Seidel or Jacobi, stopping when the relative error between subsequent solution vectors is less than 10^{-5}. The uniform probability vector is used for the initial solution vector.

Holding times are computed as needed, except for the explicit storage case, which uses a full vector to store them. In the Kronecker case, we use the Kronecker expression in Equation 7.6 for the row sums to compute the holding times as described in [97]. For the matrix diagram technique, we use a second matrix diagram to represent the row sums, which is built from Equation 7.6. Alternatively, we could use a full vector to store the holding times (as in the explicit storage case). This requires an additional vector of size $|\mathcal{S}|$, but reduces the CPU time by 10% to 20% for the models we checked.

The sizes of the underlying CTMCs for various numbers of jobs in the Kanban and FMS systems are shown in Table 7.2, along with the amount of memory required to store the transition rate matrix $\mathbf{R}$ using various techniques. Note that the memory requirements for the matrix diagram representation are greater than for the Kronecker representation, as expected. Of course, these requirements are only a small fraction of the memory required for the solution vector, which is the memory bottleneck since it requires one floating-point number per reachable state. To illustrate the memory savings of matrix diagram and Kronecker representations with respect to explicit (sparse) matrix storage, those memory requirements are also listed in the table. The memory requirements for explicit matrix

Model	# Jobs	# states $\lvert\mathcal{S}\rvert$	# arcs $\eta(\mathbf{R})$	Memory required (bytes) to represent $\mathbf{R}$ Matrix diagram	Kronecker	Explicit
Kanban	1	160	616	2,119	1,382	4,448
	2	4,600	28,120	4,169	2,246	205,608
	3	58,400	446,400	7,599	3,746	3,145,688
	4	454,475	3,979,850	13,518	6,096	35,474,688
	5	2,546,432	24,460,016	21,667	9,486	216,051,672
	6	11,261,376	115,708,992	32,702	14,106	1,015,762,944†
	7	41,644,800	450,455,040	46,678	20,388	3,936,798,720†
FMS	1	120	345	4,082	4,955	2,773
	2	3,444	16,311	14,369	9,216	125,506
	3	48,590	297,382	45,775	19,346	2,173,100
	4	438,600	3,166,985	128,444	41,757	28,844,768
	5	2,895,018	23,527,185	302,327	88,210	211,377,712
	6	15,126,440	134,459,031	643,715	170,152	1,196,683,768†

† These values are projections, computed as $8\eta(\mathbf{R}) + 8|\mathcal{S}|$.

Table 7.2: CTMC sizes and memory requirements for Kanban and FMS

storage include the full vector of holding times. For the larger cases (i.e., $N = 6$ and $N = 7$ for Kanban, $N = 6$ for FMS), the explicit storage requirements exceeded the available memory on our machine; in the table we show the amount of memory required for a sparse representation of $\mathbf{R}$ by columns and a vector of holding times. Specifically, we assume that explicit storage requires a four-byte integer and a four-byte, single-precision real for each non-zero element of $\mathbf{R}$, a four-byte integer for each column of $\mathbf{R}$, and a single-precision real for each diagonal element of $\mathbf{R}$.

The average iteration times for matrix diagram, Kronecker, and explicit techniques are shown in Table 7.3. The average iteration time is computed as the total time required by the iterative technique divided by the number of iterations. For the Kronecker case, we use the multi-level data structure for $\mathcal{S}$ (with the logarithmic overhead) as described in [15] and

Model		Matrix diagram Gauss-Seidel		Kronecker Gauss-Seidel		Kronecker Jacobi		Explicit Gauss-Seidel	
	# Jobs	Iters	sec/iter	Iters	sec/iter	Iters	sec/iter	Iters	sec/iter
Kanban	1	19	0.004	26	0.005	84	0.002	26	0.0004
	2	40	0.104	55	0.170	134	0.087	55	0.0169
	3	67	1.389	101	2.660	240	1.299	97	0.2568
	4	99	11.564	161	24.993	370	11.540	149	2.2182
	5	139	68.621	265	158.565	527	71.474	214	13.3769
	6	185	313.498	334	762.053	712	344.677	—	—
	7	238	1,207.440	445	3,078.340	—	—	—	—
FMS	1	62	0.004	63	0.008	217	0.004	61	0.0003
	2	132	0.103	137	0.280	506	0.135	126	0.0113
	3	203	1.537	227	4.679	802	2.137	191	0.1936
	4	273	14.431	311	53.342	1,100	21.745	256	1.9468
	5	500	100.237	500	394.695	1,500	155.264	500	13.7506

Table 7.3: Iteration times for Kanban and FMS

[27], and we use the best multiplication algorithms described in [15]. Note that the number of Gauss-Seidel iterations differs for each of the three techniques; this is because the states are visited in different orders by the three techniques, and Gauss-Seidel is sensitive to this ordering. In the explicit case, states are visited in the order in which they are discovered; in the Kronecker case, they are visited in lexicographical order where the substates are ordered in discovery order; and in the matrix diagram case, they are visited according to the "upside-down" lexicographical order.

As expected, the number of iterations required by Jacobi is much higher than for Gauss-Seidel, but (in the Kronecker case) the per-iteration cost for Jacobi is much less than for Gauss-Seidel. For the Kanban model, we use Jacobi with an under-relaxation parameter of $\omega = 0.9$. We see that the per-iteration cost of matrix diagrams using Gauss-Seidel is roughly equal to the per-iteration cost of Kronecker using Jacobi. Gauss-Seidel is the more desirable of the two iterative techniques, since the number of iterations can be much smaller than

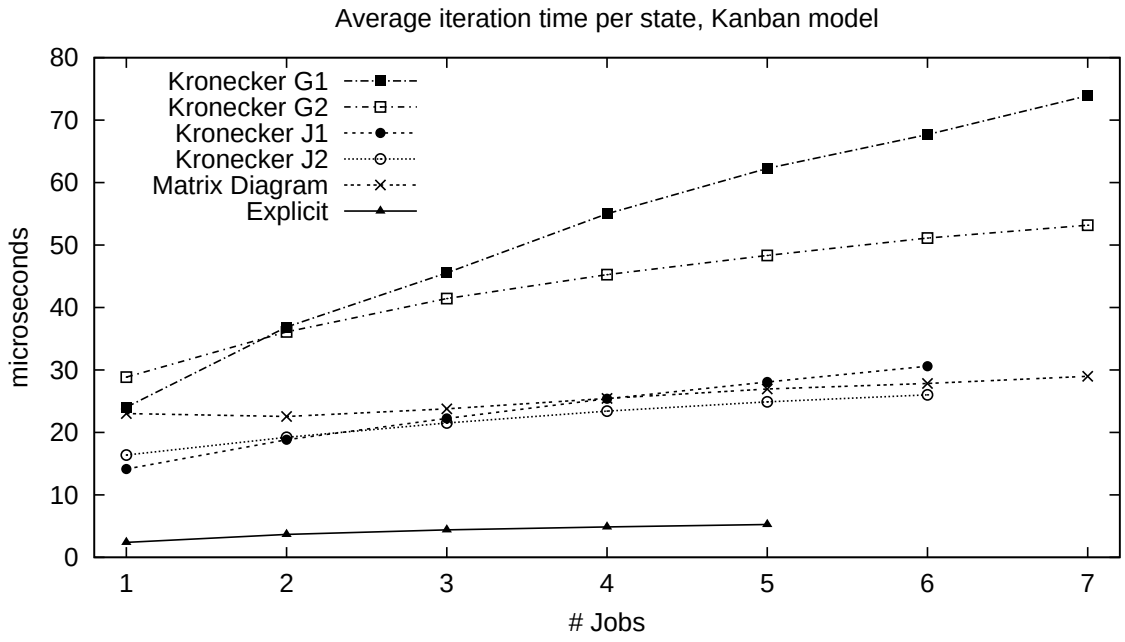


Figure 7.13: Column multiplication times, Kanban model

Jacobi and since Jacobi requires an additional full-size, double-precision accumulator vector. The missing data in the table for Jacobi is due to these excessive memory requirements.

We can also use our new MDD structure with Kronecker techniques to eliminate the logarithmic overhead. A comparison of various techniques is shown in Figure 7.13 for the Kanban model, and in Figure 7.14 for the FMS model. In the plots, we compare the following.

Kronecker G1: We use a Kronecker representation using the multi-level structure with logarithmic overhead, with Gauss-Seidel as our iterative technique.

Kronecker G2: We use a Kronecker representation using our new MDD-based structure without logarithmic overhead, with Gauss-Seidel as our iterative technique.

Kronecker J1: We use a Kronecker representation using the multi-level structure with

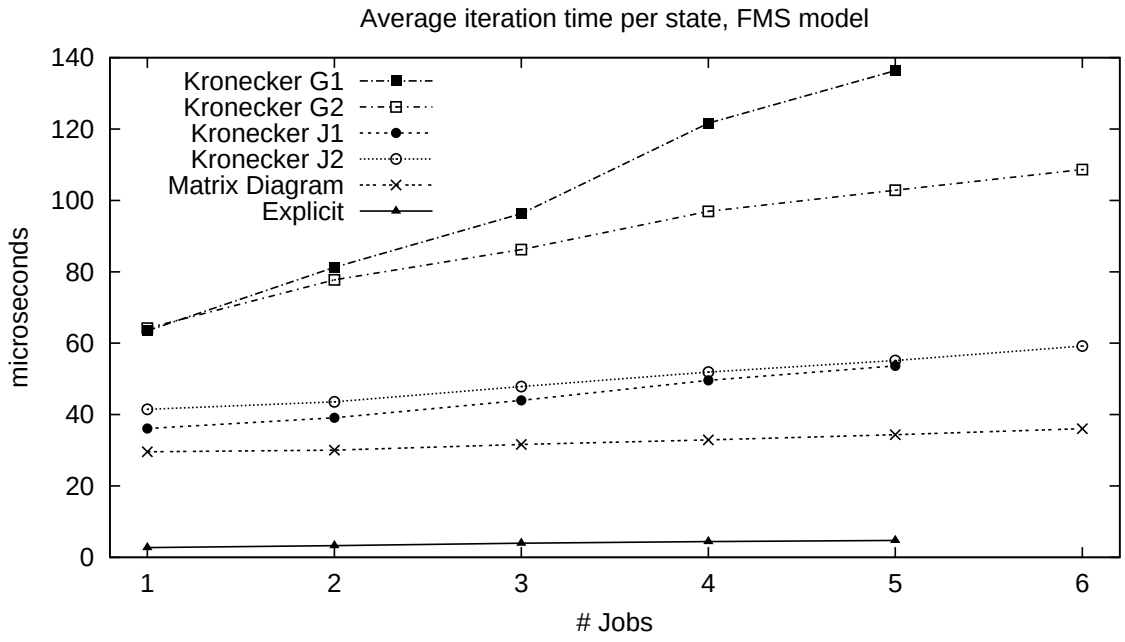


Figure 7.14: Column multiplication times, FMS model

logarithmic overhead, with Jacobi as our iterative technique.

Kronecker J2: We use a Kronecker representation using our new MDD-based structure without logarithmic overhead, with Jacobi as our iterative technique.

Matrix Diagram: We use a matrix diagram representation with Gauss-Seidel. There is no logarithmic overhead.

Explicit: The transition rate matrix is stored using explicit sparse storage by columns, and we use Gauss-Seidel as our iterative technique.

The plots compare the average iteration time per state, which is computed as the average iteration time divided by the number of reachable states. This quantity can be interpreted as the average amount of time required to compute the new value of one entry in the solution vector. We see that our new data structure for $\mathcal{S}$ that eliminates the logarithmic overhead

causes significant improvements to the Kronecker techniques for large models, especially when Gauss-Seidel is used. For smaller models the new data structure can actually increase the overhead; this is because a state index is computed by summing the offsets in the new data structure, but is explicitly stored in the multi-level data structure. Thus, for smaller models, the cost of summing the offsets can be higher than the cost of the binary searches.

We also see that our matrix diagram technique greatly outperforms the Kronecker techniques for Gauss-Seidel. This great difference is due to the reduction in the number of floating-point multiplications from using the cache. The matrix diagram technique using Gauss-Seidel requires about as much time per iteration as the Kronecker techniques using Jacobi in the Kanban model, and requires quite a bit less time in the FMS model.

7.8 Conclusion

Using a Kronecker expression, we can represent extremely large matrices in a compact form. However, the cost of this compact representation is a substantial CPU overhead introduced in numerical solution. This overhead is mainly caused by floating-point multiplications in the Kronecker expression and the need to convert potential indices to actual indices. The overheads are especially expensive for iterative solvers that require access to matrix columns, since the columns can contain "spurious" entries and current state-of-the-art column multiplication algorithms have higher cost.

Previous techniques made use of data structures that require binary searches when converting potential indices into actual indices, thus introducing a logarithmic overhead. We present a data structure that does not require binary searches. By simply changing

data structures for the storage of the state space, we can reduce the overhead due to the index conversion. For column multiplication algorithms, this reduction is substantial: up to 30% for the models we tested.

Column multiplication can be additionally improved by using our new data structure, *matrix diagrams*. The matrix diagram representation requires more memory than the Kronecker representation, but this amount of memory is an insignificant fraction of the memory required by the solution vector. Since the matrix diagram representation is exact, the problem of "spurious" column entries is eliminated. We show how a cache can be used to reduce the number of floating-point multiplications required when obtaining a matrix column, and how to visit columns such that the cache is used as often as possible.

Our results show a speedup factor of two or greater in the solution times with respect to the fastest algorithms previously proposed. Since this difference is due to the elimination of a logarithmic overhead factor and a reduction in the number of floating-point multiplications, these differences are going to become even more relevant as improvements in hardware allow us to tackle larger models.

Chapter 8

A Stationary Approximation

Using data structures described in previous chapters, we can represent the state space $\mathcal{S}$ and the transition rate matrix $\mathbf{R}$ of the underlying CTMC extremely efficiently. The remaining memory bottleneck for an exact analysis of an extremely large CTMC is the stationary probability vector $\boldsymbol{\pi}$. For certain types of models, exact analysis is possible by studying portions of the model in isolation, and then combining these results to obtain the stationary measures of interest about the model. Excellent examples of this type of approach are product-form models [7, 92] and techniques using exact aggregation [102]. Unfortunately, these results are exact only for a restricted class of models. Obtaining exact stationary probabilities for a general class of models, without explicitly storing the entire vector $\boldsymbol{\pi}$, appears to be a very difficult problem.

In this chapter we present a novel technique for *approximating* $\boldsymbol{\pi}$. Some techniques are available in which approximations are done at the matrix level, where structural assumptions are made about the transition rate matrix $\mathbf{R}$ which allow for computation of approximations or bounds for the stationary probability vector [34, 73]. A somewhat more general (and popular) approach is to instead do the approximation at the model level, in which structural properties of the model are used to perform a decomposition. Each submodel of

the decomposition is analyzed in isolation. Results from one submodel may be used when analyzing another submodel, if there are interactions between the submodels. Fixed-point iterations are used to break cyclic dependencies among the submodels. Techniques of this type have been successfully adopted in [20, 30, 50, 51, 53, 93, 98, 100, 105, 107]. Proofs that the fixed-point iterations do indeed converge are usually difficult to obtain [64].

Our approximation technique [69] is unique, in the sense that exact knowledge of $\mathcal{S}$ and $\mathbf{R}$ is used. Our technique uses structural properties of $\mathcal{S}$ to determine classes of states of the underlying CTMC. We then perform a series of K aggregations on these classes of states. Aggregation k is determined based on $\mathcal{S}$, exact knowledge of $\mathbf{R}$, the probabilities computed from the other aggregations, and a single simplifying assumption. We show that the assumption holds and thus the results are exact if the model has a product-form solution; however, in the general case, the results are approximate.

The remainder of this chapter is organized as follows. Section 8.1 gives a brief overview of exact aggregation of a CTMC. Section 8.2 describes our approximate aggregation technique; in particular, we show how the states are grouped into classes and how the rates between classes are computed. Section 8.3 presents our algorithm for performing the approximate aggregation. In Section 8.4, we demonstrate that our aggregation is exact if the model possesses a product-form solution. Section 8.5 gives some experimental results in which our approximation is compared with exact analysis for different types of models. Finally, Section 8.6 presents concluding remarks.

8.1 Exact aggregation

Given a CTMC with state space $\mathcal{S}$, transition rate matrix $\mathbf{R}$, and stationary probability vector $\boldsymbol{\pi}$, we can define an aggregated CTMC by partitioning $\mathcal{S}$ into C disjoint classes $\mathcal{C}_1, \ldots, \mathcal{C}_C$. The aggregated CTMC treats an entire class as a single state, and thus has state space $\{1, \ldots, C\}$, and has transition rate matrix $\mathbf{R}_{aggr}$ given by

$$\mathbf{R}_{aggr}[i,j] = \sum_{\mathbf{i} \in \mathcal{C}_i} \boldsymbol{\pi}[\mathbf{i}|\mathcal{C}_i] \cdot \sum_{\mathbf{j} \in \mathcal{C}_j} \mathbf{R}[\mathbf{i},\mathbf{j}], \tag{8.1}$$

where $\boldsymbol{\pi}[\mathbf{i}|\mathcal{C}_i]$ is the conditional probability that the original CTMC is in state $\mathbf{i}$ given that it is one of the states of $\mathcal{C}_i$:

$$\boldsymbol{\pi}[\mathbf{i}|\mathcal{C}_i] = \boldsymbol{\pi}[\mathbf{i}] \cdot \left(\sum_{\mathbf{j} \in \mathcal{C}_i} \boldsymbol{\pi}[\mathbf{j}] \right)^{-1}. \tag{8.2}$$

Note that $\mathbf{R}_{aggr}[i,j]$ represents the stationary rate at which the original CTMC goes from a state in class $\mathcal{C}_i$ to a state in class $\mathcal{C}_j$, given that it is in a state in class $\mathcal{C}_i$. If the original CTMC is ergodic, so is the aggregated CTMC, and its stationary probability vector $\boldsymbol{\pi}_{aggr}$ satisfies $\boldsymbol{\pi}_{aggr} = \sum_{\mathbf{i} \in \mathcal{C}_i} \boldsymbol{\pi}[\mathbf{i}]$.

An example showing an aggregation of a simple CTMC is given in Figure 8.1. In the example, the state space $\mathcal{S} = \{(1010), (1001), (0101), (0110), (1100), (0011)\}$ is partitioned into classes $\mathcal{C}_1 = \{(1010), (1001)\}$, $\mathcal{C}_2 = \{(0101), (0110)\}$, and $\mathcal{C}_3 = \{(1100), (0011)\}$. The aggregated CTMC is shown on the right, where the transition rates are computed according to Equation 8.1. Clearly, both the original and aggregated CTMCs are ergodic, since every state can reach every other state. Note that the rate of transition between $\mathcal{C}_1$ and $\mathcal{C}_2$ is $\lambda_1 \cdot \Pr\{(1010)|\mathcal{C}_1\} + \lambda_1 \cdot \Pr\{(1001)|\mathcal{C}_1\}$, which reduces to simply λ_1 due to the law of total probability.

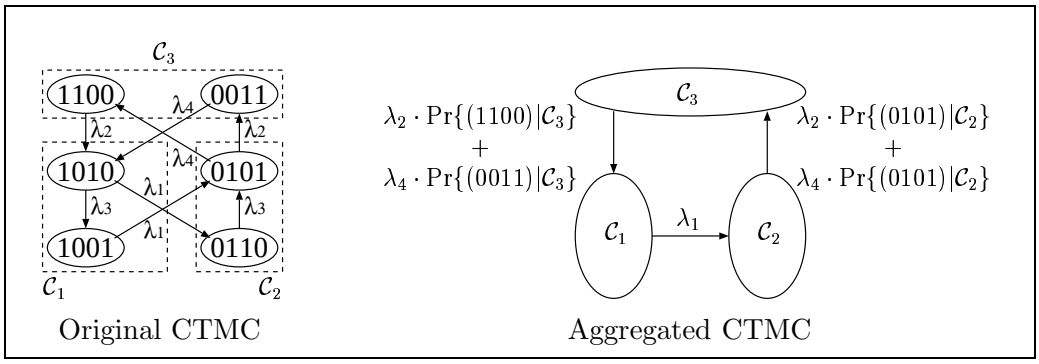


Figure 8.1: A simple aggregation example

While exact aggregation allows us in principle to greatly reduce the size of the state space to be considered, its use is rarely practical, since the computation of the aggregate rates in Equation 8.1 can be correctly performed in general only if the stationary probability vector $\boldsymbol{\pi}$ of the original CTMC is known. In practice, we often resort to approximate aggregation, where *estimates* of the conditional probabilities $\boldsymbol{\pi}[\mathbf{i}|\mathcal{C}_i]$ are used instead of the exact expression of Equation 8.2.

8.2 Approximate aggregation using decision diagrams

Any aggregation technique must specify how to partition the state space $\mathcal{S}$ and compute or approximate the conditional probabilities. Our technique uses complete knowledge of the underlying CTMC to perform approximate aggregation. We use decision diagrams to represent the state space $\mathcal{S}$ and to define our partition, and we use a Kronecker representation for the transition rate matrix $\mathbf{R}$. Thus, our technique works for *Kronecker-product-form* models.

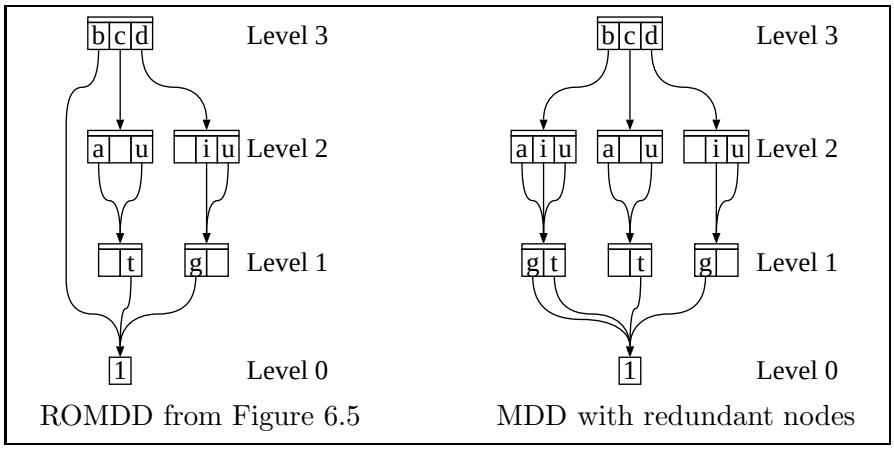
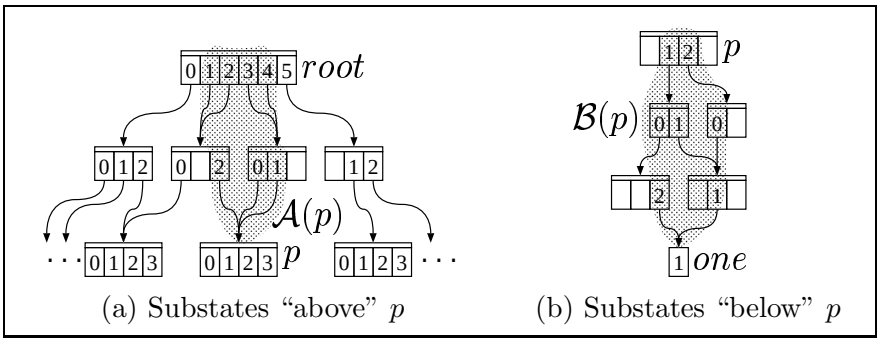


Figure 8.2: Adding redundant nodes to an ROMDD

8.2.1 Our decision diagram structure

For our aggregation technique, we use a special version of decision diagrams where downward pointers are allowed to go between adjacent levels only, and the *root* node representing the set $\mathcal{S}$ must be a level-K node. This means that we may have to introduce "redundant" nodes, in which all the downward pointers have the same non-null value. It is important to note that decision diagrams of this type are still a canonical representation of integer functions. An example showing an MDD containing redundant nodes so that pointers are between adjacent levels only is shown in Figure 8.2.

The concept of reachable *substate* is fundamental to our discussion. We denote the set of all level-k nodes in the MDD representing $\mathcal{S}$ as $\mathcal{L}[k]$, and we denote the i^{th} downward pointer of a node p as $p[i]$. Formally, if a node $p \in \mathcal{L}[k]$ represents integer function f, then $p[i]$ is the node representing $f_{x_k=i}$. Given an MDD node $p \in \mathcal{L}[k]$, we can recursively define



(a) Substates "above" p (b) Substates "below" p

Figure 8.3: Example of sets $\mathcal{A}(p)$ and $\mathcal{B}(p)$

the set $\mathcal{B}(p)$ of substates encoded by (or the substates "below") node p as

$$\mathcal{B}(p) = \begin{cases} \bigcup_{i \in \mathcal{S}_k} \{i\} \times \mathcal{B}(p[i]) & \text{if } k > 1 \\ \{i \in \mathcal{S}_1 : p[i] \neq \mathsf{null}\} & \text{if } k = 1 \end{cases},$$

where $\mathcal{B}(\mathsf{null})$ is the empty set by definition. Note that the reachability set $\mathcal{S}$ is encoded by the *root* node: $\mathcal{S} = \mathcal{B}(root)$.

We say a substate $\alpha = (i_K, \ldots, i_k)$ is reachable if there is at least one state in $\mathcal{S}$ beginning with α (i.e., there exists a reachable state $(i_K, \ldots, i_1) \in \mathcal{S}$). Such a substate uniquely identifies a level-k node in the decision diagram representing $\mathcal{S}$:

$$node(\alpha) = \begin{cases} root & \text{if } |\alpha| = 0 \\ node(i_K, \ldots, i_{k+1})[i_k] & \text{if } \alpha = (i_K, \ldots, i_k) \end{cases}.$$

We can then define the set $\mathcal{A}(p)$ of substates that lead to (or the substates "above") node p as $\mathcal{A}(p) = \{\alpha : node(\alpha) = p\}$. In particular, all the reachable states lead to the special node *one* (the terminal node 1 at level 0); thus we have $\mathcal{A}(one) = \mathcal{S}$. Figure 8.3 presents an example of sets $\mathcal{A}(p)$ and $\mathcal{B}(p)$. In Figure 8.3(a) we have $\mathcal{A}(p) = \{12, 22, 30, 31, 40, 41\}$, and in Figure 8.3(b) we have $\mathcal{B}(p) = \{102, 111, 201\}$.

8.2.2 A partition based on decision diagrams

We will now apply aggregation based on a partition of $\mathcal{S}$ that is guided by the decision diagram structure for $\mathcal{S}$. We will build K aggregated CTMCs, one for each level of the decision diagram. The state space $\mathcal{M}_k$ for the level-k CTMC contains states of the form (p, i_k), where p is a level-k node in the decision diagram for $\mathcal{S}$ and i_k is a reachable local state for that node:

$$\mathcal{M}_k = \{(p, i_k) : p \in \mathcal{L}[k] \wedge p[i_k] \neq \mathsf{null}\}.$$

The set $\mathcal{M}_k$ induces the partition $\mathcal{S} = \bigcup_{(p,i_k)\in\mathcal{M}_k} \mathcal{C}(p, i_k)$, where

$$\mathcal{C}(p, i_k) = \mathcal{A}(p) \times \{i_k\} \times \mathcal{B}(p[i_k]).$$

Intuitively, class $\mathcal{C}(p, i_k)$ contains the states with i_k as the local state for submodel k whose path in the decision diagram from node *root* to node *one* includes node p. For example, using the decision diagrams in Figure 8.3, class $\mathcal{C}(p, 1)$ contains states {12102, 12111, 22102, 22111, 30102, 30111, 31102, 31111, 40102, 40111, 41102, 41111} and $\mathcal{C}(p, 2)$ = {12201, 22201, 30201, 31201, 40201, 41201}.

For an exact aggregation, we derive the rate from (p, i_k) to (q, j_k) in the level-k CTMC

due to event e by combining the Kronecker representation (Equation 7.3) and Equation 8.1:

$$\begin{aligned}
\mathbf{R}^e_{\mathcal{M}_k}[(p,i_k),(q,j_k)] &= \sum_{\mathbf{i}\in\mathcal{C}(p,i_k)} \boldsymbol{\pi}[\mathbf{i}|\mathcal{C}(p,i_k)] \cdot \sum_{\mathbf{j}\in\mathcal{C}(q,j_k)} \mathbf{R}^e[\mathbf{i},\mathbf{j}] \\
&\quad \left(\text{let} \begin{array}{l} (\alpha,i_k,\beta) = (i_K,\ldots,i_1) = \mathbf{i}\ , \\ (\alpha',j_k,\beta') = (j_K,\ldots,j_1) = \mathbf{j} \end{array}\right) \\
&= \sum_{\alpha\in\mathcal{A}(p),\ \beta\in\mathcal{B}(p[i_k])} \boldsymbol{\pi}[(\alpha,i_k,\beta)|\mathcal{C}(p,i_k)] \cdot \sum_{\alpha'\in\mathcal{A}(q),\ \beta'\in\mathcal{B}(q[j_k])} \mathbf{W}^e_K[i_K,j_K]\cdots\mathbf{W}^e_1[i_1,j_1] \\
&= \sum_{\alpha\in\mathcal{A}(p)} \Pr\{\alpha,i_k|p,i_k\} \cdot \sum_{\alpha'\in\mathcal{A}(q)} \mathbf{W}^e_K[i_K,j_K]\cdots\mathbf{W}^e_k[i_k,j_k] \cdot \\
&\quad \sum_{\beta\in\mathcal{B}(p[i_k])} \Pr\{\beta|\alpha,i_k\} \cdot \sum_{\beta'\in\mathcal{B}(q[j_k])} \mathbf{W}^e_{k-1}[i_{k-1},j_{k-1}]\cdots\mathbf{W}^e_1[i_1,j_1] \\
&= \sum_{\alpha\in\mathcal{A}(p)} \Pr\{\alpha|p,i_k\} \cdot \sum_{\alpha'\in\mathcal{A}(q)} \mathbf{W}^e_K[i_K,j_K]\cdots\mathbf{W}^e_k[i_k,j_k] \cdot \\
&\quad \sum_{\beta\in\mathcal{B}(p[i_k])} \Pr\{\beta|\alpha,i_k\} \cdot \mathit{Rate}^e_{k-1}(i_{k-1})\cdots\mathit{Rate}^e_1(i_1). \qquad (8.3)
\end{aligned}$$

Note that we must be able to compute the quantities $\Pr\{\alpha|p,i_k\}$ and $\Pr\{\beta|\alpha,i_k\}$ if we are to perform exact aggregation. It can be shown that if the original CTMC is ergodic, then the level-k CTMC is also ergodic. We denote the stationary probability vector of the level-k CTMC as $\boldsymbol{\pi}_k$.

8.2.3 A simple case of exact aggregation

We will now discuss a special case where exact aggregation naturally arises. Consider a decision diagram that is actually a tree, because we forbid two arcs from pointing to the same node. Such a decision diagram may contain replicated nodes, and is equivalent to the multi-level structures presented in Chapter 5. Since the decision diagram is a tree, there is exactly one path to each node p (i.e., $|\mathcal{A}(p)| = 1$). In this case, we have

$$\Pr\{\alpha|p,i_k\} = 1 \qquad \text{and} \qquad \Pr\{\beta|\alpha,i_k\} = \Pr\{\beta|p,i_k\}$$

and so we can rewrite Equation 8.3 for $\mathbf{R}^e_{\mathcal{M}_k}$ as

$$\mathbf{R}^e_{\mathcal{M}_k}[(p, i_k), (q, j_k)] = \mathbf{A}^e_k[p, q] \cdot \mathbf{W}^e_k[i_k, j_k] \cdot \mathbf{b}^e_{k-1}[p[i_k]]$$

where $\mathbf{A}^e_k[p, q]$, defined as

$$\mathbf{A}^e_k[p, q] = \mathbf{W}^e_K[i_K, j_K] \cdots \mathbf{W}^e_{k+1}[i_{k+1}, j_{k+1}],$$

expresses the contribution to the rate by capturing the rate of transition from node p to node q due to event e "from above", and

$$\mathbf{b}^e_k[p[i_k]] = \sum_{\beta \in \mathcal{B}(p[i_k])} \Pr\{\beta | p, i_k\} \cdot Rate^e_{k-1}(i_{k-1}) \cdots Rate^e_1(i_1)$$

expresses the contribution "from below". Defining the terminal cases $\mathbf{A}^e_K[root, root] = 1$ and $\mathbf{b}^e_0[one] = 1$, we can express $\mathbf{A}$ and $\mathbf{b}$ recursively: if $p = p'[i_{k+1}]$ and $q = q'[j_{k+1}]$,

$$\mathbf{A}^e_k[p, q] = \mathbf{A}^e_{k+1}[p', q'] \cdot \mathbf{W}^e_{k+1}[i_{k+1}, j_{k+1}] \tag{8.4}$$

$$\mathbf{b}^e_k[p] = \sum_{i_k : p[i_k] \neq \mathsf{null}} \Pr\{i_k | p\} \cdot Rate^e_k(i_k) \cdot \mathbf{b}^e_{k-1}[p[i_k]]. \tag{8.5}$$

Note that each class in the partition at level 1 contains one state of the original CTMC: $|\mathcal{M}_1| = |\mathcal{S}|$. Thus, at level 1, the "aggregated" CTMC and the exact CTMC coincide, and we have $\boldsymbol{\pi}_1 = \boldsymbol{\pi}$. The recursive computation of $\mathbf{b}$ can then use the correct conditional probabilities $\Pr\{i_k | p\}$ at every level. In other words, every level-k CTMC is an exact aggregation of the original CTMC. Of course, this is an illustrative case only, since there is no point in defining aggregated CTMCs once the exact CTMC has been solved.

8.2.4 Our approximation

Armed with the intuition from the previous section, we are now ready to consider the case where the decision diagram for $\mathcal{S}$ is reduced, and multiple paths can reach node p (i.e., $|\mathcal{A}(p)| \geq 1$). We will now introduce the source of our approximation, and from that we will derive the expression for the (approximately) aggregated CTMCs. Our goal is to obtain stationary probability vectors for the aggregated CTMCs at each level, and then use these vectors to approximate the stationary probability vector for the original CTMC. To do so, we assume that

$$\forall \alpha \in \mathcal{A}(p), \quad \Pr\{i_k|\alpha\} = \Pr\{i_k|p\}, \tag{8.6}$$

or, equivalently,

$$\forall \alpha, \alpha' \in \mathcal{A}(p), \quad \Pr\{i_k|\alpha\} = \Pr\left\{i_k|\alpha'\right\},$$

which implies that node p captures all the information about the path α, from *root* to p, required to correctly compute the rate of an event e.

Of course, this is a strong assumption, and the quality of the approximation will ultimately depend on how closely it holds. However, this assumption has excellent structural justification: the set of substates (i_k, β) that can complete a given α to form a reachable state $(\alpha, i_k, \beta) \in \mathcal{S}$ is $\mathcal{B}(p)$, which is of course exactly the same for any $\alpha \in \mathcal{A}(p)$. Since zero probability is associated with unreachable states only, this implies that, given paths $\alpha, \alpha' \in \mathcal{A}(p)$, we have $\Pr\{i_k|\alpha\} = 0 \Leftrightarrow \Pr\{i_k|\alpha'\} = 0$. In other words, Equation 8.6 holds for i_k such that $\Pr\{i_k|\alpha\}$ is zero. This is one of the strengths of our approximation algorithm: we correctly assign zero probability to "spurious" states in $\hat{\mathcal{S}} \setminus \mathcal{S}$, since we have an exact encoding of the state space $\mathcal{S}$. What is not true in general, of course, is that

$\Pr\{i_k|\alpha\} = \Pr\{i_k|\alpha'\}$ when they are not zero.

Using the assumption of Equation 8.6, the stationary probability of state $(i_K, \ldots, i_1)$ of the original CTMC is then

$$\begin{aligned}
\Pr\{i_K, \ldots, i_1\} &= \Pr\{i_1|i_K, \ldots, i_2\} \cdot \Pr\{i_K, \ldots, i_2\} \\
&= \Pr\{i_1|p_1\} \cdot \Pr\{i_K, \ldots, i_2\} = \prod_{k=1}^{K} \Pr\{i_k|p_k\} \\
&= \prod_{k=1}^{K} \frac{\boldsymbol{\pi}_k[(p_k, i_k)]}{\Pr\{p_k\}}, \qquad (8.7)
\end{aligned}$$

where $p_k = node(i_K, \ldots, i_{k+1})$, $\boldsymbol{\pi}_k[(p_k, i_k)]$ is the stationary probability of state (p_k, i_k) in the level-k CTMC, and $\Pr\{p_k\}$ is the probability associated with node p_k, with $\Pr\{p_K\} = \Pr\{root\} = 1$. Equation 8.6 also allows us to rewrite $\Pr\{\alpha|p, i_k\}$ as

$$\Pr\{\alpha|p, i_k\} = \frac{\Pr\{\alpha, p, i_k\}}{\Pr\{p, i_k\}} = \frac{\Pr\{p, i_k|\alpha\} \cdot \Pr\{\alpha\}}{\Pr\{i_k|p\} \cdot \Pr\{p\}} = \frac{\Pr\{i_k|\alpha\} \cdot \Pr\{\alpha\}}{\Pr\{i_k|p\} \cdot \Pr\{p\}} = \frac{\Pr\{\alpha\}}{\Pr\{p\}} \qquad (8.8)$$

and similarly allows us to recursively eliminate the dependency of β on α in $\Pr\{\beta|\alpha, i_k\}$. By letting β equal $(i_{k-1}, \ldots, i_1) = (i_{k-1}, \gamma)$, we have

$$\begin{aligned}
\Pr\{\beta|\alpha, i_k\} &= \Pr\{i_{k-1}, \gamma|\alpha, i_k\} = \Pr\{i_{k-1}|\alpha, i_k\} \cdot \Pr\{\gamma|\alpha, i_k, i_{k-1}\} \\
&= \Pr\{i_{k-1}|p, i_k\} \cdot \Pr\{\gamma|\alpha, i_k, i_{k-1}\} \\
&= \Pr\{i_{k-1}|p, i_k\} \cdots \Pr\{i_1|p, i_k, \ldots, i_2\} \\
&= \Pr\{i_{k-1}, \ldots, i_1|p, i_k\} \\
&= \Pr\{\beta|p, i_k\}. \qquad (8.9)
\end{aligned}$$

We can then substitute Equation 8.8 and Equation 8.9 into Equation 8.3, giving

$$\begin{aligned}\mathbf{R}^e_{\mathcal{M}_k}[(p,i_k),(q,j_k)] &= \left(\sum_{\alpha\in\mathcal{A}(p)}\frac{\Pr\{\alpha\}}{\Pr\{p\}}\cdot\sum_{\alpha'\in\mathcal{A}(q)}\mathbf{W}^e_K[i_K,j_K]\cdots\mathbf{W}^e_{k+1}[i_{k+1},j_{k+1}]\right)\cdot \\ &\quad \mathbf{W}^e_k[i_k,j_k]\cdot \\ &\quad \left(\sum_{\beta\in\mathcal{B}(p[i_k])}\Pr\{\beta|p,i_k\}\cdot Rate^e_{k-1}(i_{k-1})\cdots Rate^e_1(i_1)\right) \\ &= \mathbf{A}^e_k[p,q]\cdot\mathbf{W}^e_k[i_k,j_k]\cdot\mathbf{b}^e_{k-1}[p[i_k]] \qquad (8.10)\end{aligned}$$

where $\mathbf{A}$ can be written as the recurrence

$$\begin{aligned}\mathbf{A}^e_k[p,q] &= \sum_{(i_K,\ldots,i_{k+1})\in\mathcal{A}(p)}\frac{\Pr\{i_K,\ldots,i_{k+1}\}}{\Pr\{p\}}\cdot\sum_{(j_K,\ldots,j_{k+1})\in\mathcal{A}(q)}\mathbf{W}^e_K[i_K,j_K]\cdots\mathbf{W}^e_{k+1}[i_{k+1},j_{k+1}] \\ &= \sum_{p',i_{k+1}:p'[i_{k+1}]=p}\frac{\Pr\{p',i_{k+1}\}}{\Pr\{p\}}\cdot\sum_{q',j_{k+1}:q'[j_{k+1}]=q}\mathbf{A}^e_{k+1}[p',q']\cdot\mathbf{W}^e_{k+1}[i_{k+1},j_{k+1}] \qquad (8.11)\end{aligned}$$

which can be proved by letting $\alpha = (i_K,\ldots,i_{k+1}) = (\gamma,i_{k+1})$ and $p' = node(\gamma)$, and observing that

$$\frac{\Pr\{\alpha\}}{\Pr\{p\}} = \frac{\Pr\{i_{k+1}|\gamma\}\cdot\Pr\{\gamma\}}{\Pr\{p\}} = \frac{\Pr\{i_{k+1}|p'\}\cdot\Pr\{\gamma\}}{\Pr\{p\}} = \frac{\Pr\{p',i_{k+1}\}}{\Pr\{p\}}\cdot\frac{\Pr\{\gamma\}}{\Pr\{p'\}}.$$

Analogously, $\mathbf{b}$ can be written as the recurrence

$$\begin{aligned}\mathbf{b}^e_k[p] &= \sum_{(i_k,\ldots,i_1)\in\mathcal{B}(p)}\Pr\{i_k,\ldots,i_1|p\}\cdot Rate^e_k(i_k)\cdots Rate^e_1(i_1) \\ &= \sum_{i_k:p[i_k]\neq\mathsf{null}}\frac{\Pr\{p,i_k\}}{\Pr\{p\}}\cdot Rate^e_k(i_k)\cdot\mathbf{b}^e_{k-1}[p[i_k]]. \qquad (8.12)\end{aligned}$$

The terminating conditions of the recursions are $\mathbf{A}^e_K[root,root]=1$ and $\mathbf{b}^e_0[one]=1$, as for the exact aggregation case. Note that Equation 8.11 and Equation 8.12 simplify to Equation 8.4 and Equation 8.5 if the decision diagram is a tree.

Thus, to perform our approximate aggregation in Equation 8.10 using Equation 8.11 and Equation 8.12, we need to compute probabilities of the type $\Pr\{p\}$ and $\Pr\{p, i_k\}$, for each node p at each level k and each local state i_k such that $p[i_k] \neq \mathsf{null}$. In our algorithm, we use fixed-point iterations to compute each of the level-k probability vectors $\boldsymbol{\pi}_k$. Thus, $\Pr\{p, i_k\}$ is simply $\boldsymbol{\pi}_k[(p, i_k)]$, the entry of the (current guess of the) level-k stationary probability vector for state (p, i_k). To compute $\Pr\{p\}$, we use the expression

$$\Pr\{p\} = \sum_{i_k : p[i_k] \neq \mathsf{null}} \boldsymbol{\pi}_k[(p, i_k)]$$

and perform the sum for each node p after computing the level-k stationary probability vector. We could also use the expression

$$\Pr\{p\} = \sum_{\alpha \in \mathcal{A}(p)} \Pr\{\alpha\}$$

but the former is preferable because it is a simple sum, while the latter requires us to recursively compute $\Pr\{\alpha\}$. The number of paths $\alpha \in \mathcal{A}(p)$ can grow exponentially as we descend the decision diagram, but the recursive computation can make use of previous results using a cache, thus its cost is proportional to the number of *nodes* above p, not the number of *paths* leading to p. In any event, it is comforting to know that, upon convergence of our fixed-point iterations, the values obtained with either expression coincide.

8.2.5 An example

We will now consider the aggregated CTMCs for the running example from the previous chapter: the Petri net model of Figure 4.5. The decision diagram representing the reachable

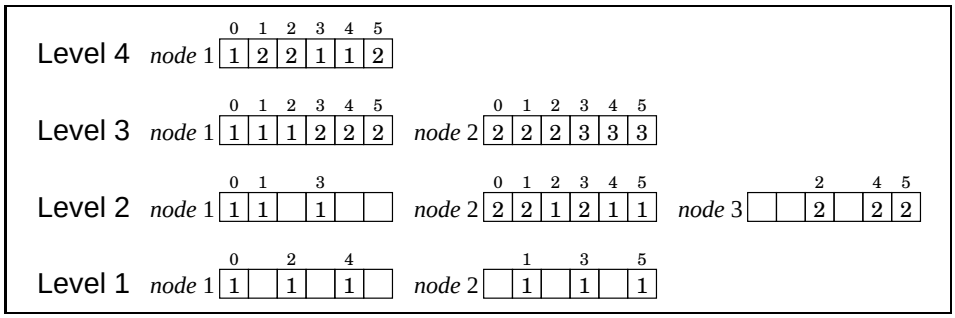


Figure 8.4: Decision diagram with node labels

states is shown in Figure 8.4. In the figure (which is equivalent to Figure 7.5), nodes are indexed by levels. The *root* node corresponds to node 1 at level 4. The nodes are depicted as arrays of indices to nodes at the next level. For clarity, the null pointers corresponding to unreachable states are drawn as blank space. States are represented as usual in the decision diagram. For instance, we see that state $(0, 0, 0, 0)$ is reachable, since $root[0][0][0][0] = one$ ($root[0]$ is node 1 at level 3, $root[0][0]$ is node 1 at level 2, $root[0][0][0]$ is node 1 at level 1), and we see that state $(0, 1, 2, 3)$ is not reachable, since $root[0][1][2] = \mathsf{null}$.

The non-blank entries in the node arrays correspond to the states of the aggregated CTMCs. For instance, the level-1 CTMC has states $\mathcal{M}_1 = \{(\#1, 0)$, (#1,2), (#1,4), (#2,1), (#2,3), $(\#2, 5)\}$. The structure of the level-4 CTMC is shown in Figure 8.5. In the figure, we depict the events that cause the transitions; the rates are computed using Equation 8.10. Note that the transitions correspond to those listed in Table 7.1; for instance, in the table we see that local state 2 goes to local state 1 via event Tc_4, and in Figure 8.5 we have a transition from state $(\#1, 2)$ to state $(\#1, 1)$ via event Tc_4. The figure shows the "interesting" transitions only. That is, the events that do not affect level 4 are omitted from the figure. For instance, there should be an arc from state $(\#1, 0)$ to itself due to event Tc_3.

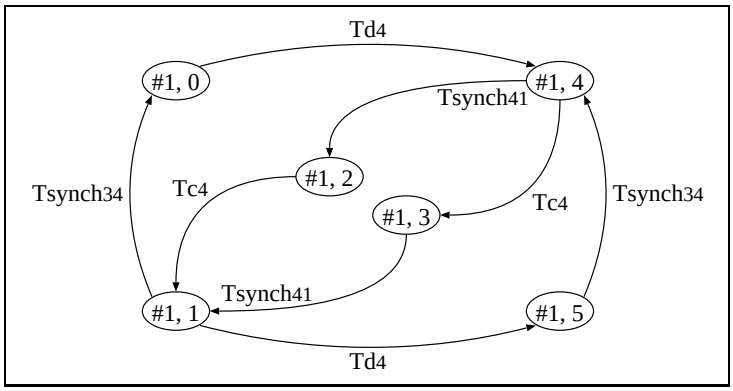


Figure 8.5: Level 4 CTMC

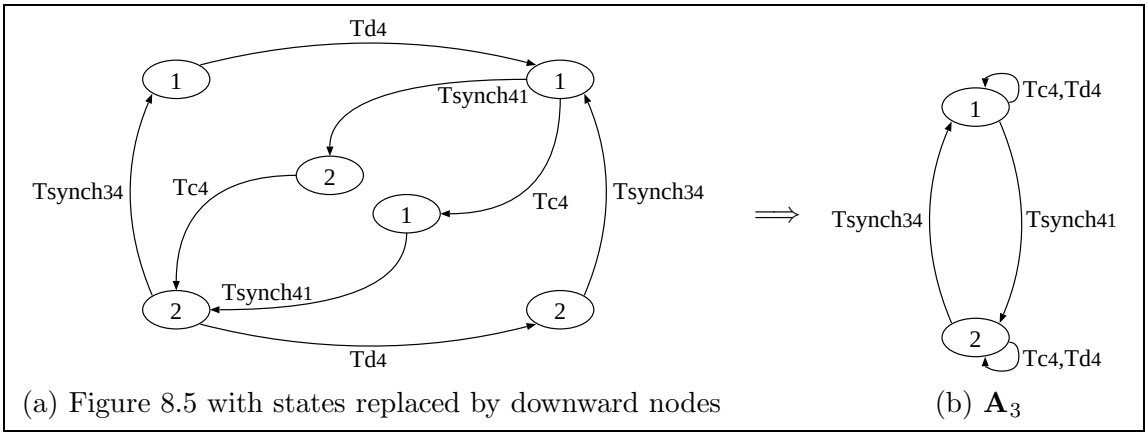


Figure 8.6: Computing **A** matrices at level 3

In fact, there should be self-arcs for every state due to events that do not affect level 4.

Computation of the **A** matrices is illustrated conceptually in Figure 8.6. The **A** matrices capture the transitions between nodes. If there is a transition from state (p, i_k) to (q, j_k) in the level-k CTMC, then there is a transition between nodes $p[i_k]$ and $q[j_k]$ in the **A** matrix for level $k-1$. In Figure 8.6(a), we replace the states from Figure 8.5 with their associated downward pointers according to Figure 8.4: (#1, 0) becomes #1[0] = 1, (#1, 1) becomes 2, (#1, 2) becomes 2, (#1, 3) becomes 1, (#1, 4) becomes 1, and (#1, 5)

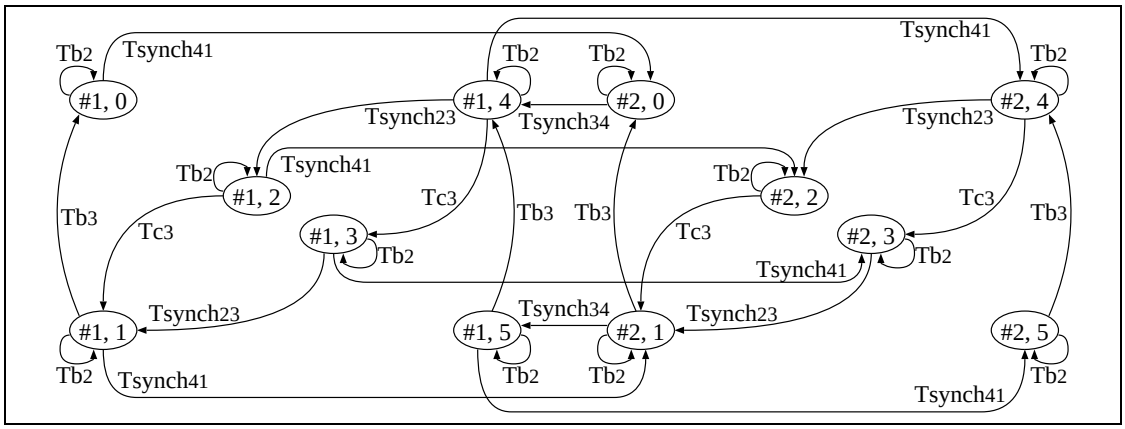


Figure 8.7: Level 3 CTMC

becomes 2. Equivalent states are then merged, producing the transition diagram of Figure 8.6(b). This diagram illustrates the non-zero entries in the **A** matrices; for instance, we see that $\mathbf{A}_3^{Tsynch_{41}}[1,2]$ is non-zero. The exact value of $\mathbf{A}_3^{Tsynch_{41}}[1,2]$ is computed using Equation 8.11.

The structure of the level-3 CTMC is shown in Figure 8.7. The state component of the transitions is determined according to Table 7.1, while the node component of the transitions is determined according to the **A** matrix structure of Figure 8.6. For instance, according to Table 7.1, event $Tsynch_{34} = Td_3$ causes a transition from local state 0 to local state 4, and from local state 1 to local state 5. Looking at Figure 8.6, we see that event $Tsynch_{34}$ causes a transition from node #2 to node #1. Thus, in the level-3 CTMC, we have transitions from state $(\#2, 0)$ to state $(\#1, 4)$ and from state $(\#2, 1)$ to state $(\#1, 5)$. Events that do not affect level 4 will produce self-arcs in the corresponding **A** matrix for level 3. For instance, event Tc_3 does not affect level 4, so the transitions due to Tc_3 in the level 3 CTMC will only change the local state component while the node component remains unchanged,

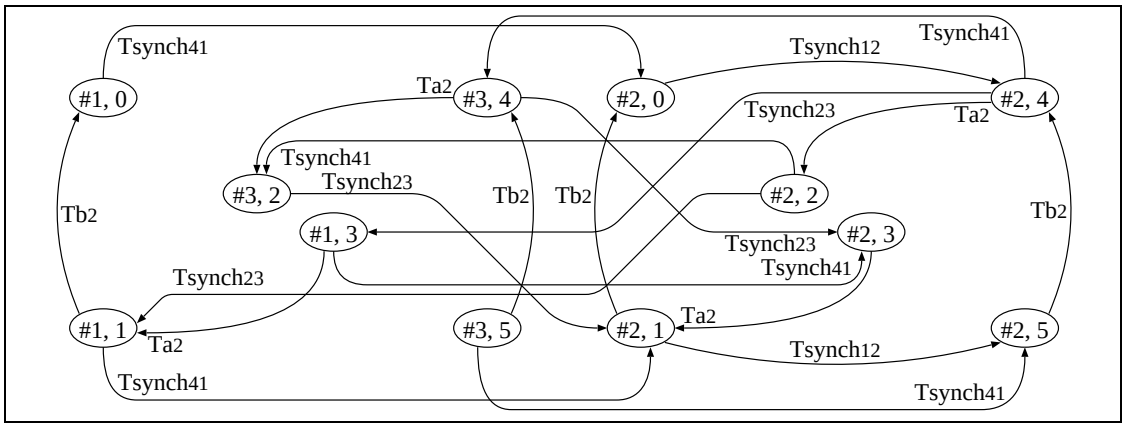


Figure 8.8: Level 2 CTMC

producing transitions such as from state $(\#1, 2)$ to state $(\#1, 1)$. The opposite effect (i.e., only the node component changes) can also occur: since event $Tsynch_{41}$ affects level 4 but not level 3, the event causes a node change but not a local state change, producing transitions such as from state $(\#1, 2)$ to state $(\#2, 2)$. We can also have "transitions" which do not change state at all, such as those due to event Tb_2. These are ignored when producing the level 3 CTMC, since they would generate self-arcs, but can be important for structure or rate information. In this particular case, the self-arcs due to event Tb_2 capture rate-dependency information, since the rate of Tb_2 depends on the local state of submodel 3, as can be seen from matrix $\mathbf{W}_3^{Tb_2}$ in Figure 7.4.

The structure of the level 2 and level 1 CTMCs can be determined in a similar fashion. For completeness, these are shown in Figure 8.8 and Figure 8.9.

8.2.6 Exploiting event locality

In the example of the previous section, we saw that we could ignore the "uninteresting" events. Specifically, if we are building the level-k CTMC, we can handle the events that do

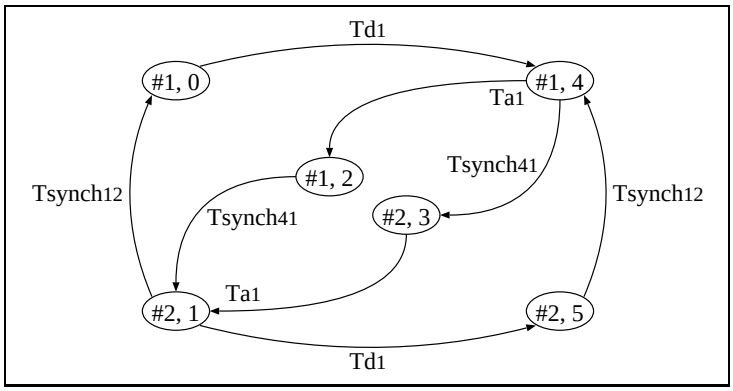


Figure 8.9: Level 1 CTMC

not affect level k in a special, more simplified way. In this section, we will formalize these simplifications.

Considering that both the enabling and the effect of an event e are restricted to the levels between $First(e)$ and $Last(e)$, we can further simplify the recurrence expressions of Equation 8.11 and Equation 8.12. For $k > First(e)$, event e has not affected any submodel yet, and $\mathbf{W}_k^e$ is the identity matrix. This gives us

$$\forall k \geq First(e), \quad \mathbf{A}_k^e[p,q] = \begin{cases} \sum_{\alpha \in \mathcal{A}(p)} \frac{\Pr\{\alpha\}}{\Pr\{p\}} = 1 & \text{if } p = q, \\ 0 & \text{otherwise} \end{cases} .$$

This means that e can be ignored for $k > First(e)$, and e changes only the local state component of the state (p, i_k) at level $k = First(e)$, but not the node component:

$$\mathbf{R}^e_{\mathcal{M}_k}[(p,i_k),(p,j_k)] = \mathbf{W}_k^e[i_k,j_k] \cdot \mathbf{b}_{k-1}^e[p[i_k]] \tag{8.13}$$

and

$$\mathbf{R}^e_{\mathcal{M}_k}[(p,i_k),(q,j_k)] = 0 \text{ for } p \neq q.$$

Analogously, since $\mathbf{W}_k^e$ is also an identity for $k < Last(e)$, which implies that $Rate_k^e(i_k)$ is

one for any $i_k \in \mathcal{S}_k$, we have

$$\forall k < \mathit{Last}(e), \quad \mathbf{b}_k^e[p] = \sum_{\beta \in \mathcal{B}(p)} \Pr\{\beta|p\} = 1.$$

This means that for $k \leq \mathit{Last}(e)$ we have

$$\mathbf{R}^e_{\mathcal{M}_k}[(p, i_k), (q, j_k)] = \mathbf{A}_k^e[p, q] \cdot \mathbf{W}_k^e[i_k, j_k].$$

When $k < \mathit{Last}(e)$, $\mathbf{W}_k^e$ is the identity matrix; this gives us

$$\mathbf{R}^e_{\mathcal{M}_k}[(p, i_k), (q, i_k)] = \mathbf{A}_k^e[p, q] \tag{8.14}$$

and

$$\mathbf{R}^e_{\mathcal{M}_k}[(p, i_k), (q, j_k)] = 0 \text{ for } i_k \neq j_k.$$

Thus when $k < \mathit{Last}(e)$, event e can change the node component but not the local state component.

Note that if e is a local event for level k, we have $\mathit{Last}(e) = k = \mathit{First}(e)$, and we obtain that the only transitions caused by e are

$$\mathbf{R}^e_{\mathcal{M}_k}[(p, i_k), (p, j_k)] = \mathbf{W}_k^e[i_k, j_k]. \tag{8.15}$$

As these rates are exact (since they do not involve the stationary probabilities), a simple rule of thumb in the hope of achieving a good approximate aggregation is to partition the model in such a way that minimizes the number of synchronizing events. Of course, this is just a heuristic; we will see that, for models exhibiting a product-form solution, our algorithm computes exact results even if *every* event is synchronizing.

We now summarize, giving an intuitive explanation for the effect of locality on the transition rates in $\mathbf{R}^e_{\mathcal{M}_k}$. Consider first the effect of a local event for level k. Since e is independent of all other levels, we know that

$$Next^e((\alpha, i_k, \beta)) = \alpha \times Next^e_k(i_k) \times \beta$$

and the rate of e is $\mathbf{W}^e_k[i_k, j_k]$ for each $j_k \in Next^e_k(i_k)$. Thus, for each state in $\mathcal{C}(p, i_k)$, we know that event e is enabled, and leads to states in $\mathcal{C}(p, j_k)$ for $j_k \in Next^e_k(i_k)$. In the level-k CTMC, this is reflected by transitions from state (p, i_k) to state (p, j_k) with rate $\mathbf{W}^e_k[i_k, j_k]$, according to Equation 8.15. It is also possible for e to affect CTMCs at lower levels. For instance, in the level-$(k-1)$ CTMC, event e causes a transition from $(p[i_k], i_{k-1})$ to $(p[j_k], i_{k-1})$. The total rate of this transition is given by $\mathbf{A}^e_k[p[i_k], p[j_k]]$, which is the average transition rate from states in $\mathcal{C}(p, i_k)$ to states in $\mathcal{C}(p, j_k)$ due to event e. This local event e will continue to affect lower levels until all the arcs corresponding to the event are self-arcs. In particular, if $p[i_k] = p[j_k]$, then the corresponding arcs at the next level will all be self-arcs, and e will not affect any other levels.

For a synchronizing event e, there are four distinct cases.

- $k > First(e)$. In this trivial case, e does not affect level k, nor does it affect any levels above it. Any transition due to e results in a self-arc in the level-k CTMC, so we can simply ignore the effects of e at this level.

- $k = First(e)$. We know that $Next^e((\alpha, i_k, \beta)) = \alpha \times Next^e_k(i_k) \times \beta'$ for some β'. As the α portion of the state does not change, the corresponding transition due to e in the level-k CTMC will be from state (p, i_k) to state (p, j_k) for $j_k \in Next^e_k(i_k)$. Unlike a

local event, though, e might be enabled in all, some, or none of the states in $\mathcal{C}(p, i_k)$. Thanks to our exact encoding of the state space $\mathcal{S}$, we can detect when e is not enabled in any state in $\mathcal{C}(p, i_k)$; if this is the case, we do not put the corresponding arc in the level-k CTMC. Otherwise, if e is enabled in some or all of the states in $\mathcal{C}(p, i_k)$, we place arcs between state (p, i_k) and all the states (p, j_k) for $j_k \in \mathit{Next}_k^e(i_k)$ such that $(p, j_k) \in \mathcal{M}_k$. The rate of each arc is given by Equation 8.13: the product of the contribution at level k and the contribution "from below".

- $\mathit{First}(e) > k \geq \mathit{Last}(e)$. We have $\mathit{Next}^e((\alpha, i_k, \beta)) = \alpha' \times \mathit{Next}_k^e(i_k) \times \beta'$ for some α', β'; this is the most general case. If e is enabled in at least one state in $\mathcal{C}(p, i_k)$, the level-k CTMC has arcs from state (p, i_k) to states (q, j_k) for $j_k \in \mathit{Next}_k^e(i_k)$ such that $(q, j_k) \in \mathcal{M}_k$, where $p = \mathit{node}(\alpha)$ and $q = \mathit{node}(\alpha')$. The rate of each arc is given by Equation 8.10: the product of the contribution "from above", from level k, and "from below". If $k = \mathit{Last}(e)$, then we know that $\mathbf{b}_{k-1}^e[p[i_k]]$ is one, since the $\mathbf{W}$ matrices are all identities at the levels below.

- $k > \mathit{Last}(e)$. This is similar to the case when an event local to level k affects levels below level k. In this case, we know that $\mathit{Next}^e((\alpha, i_k, \beta)) = \alpha' \times \{i_k\} \times \beta$ for some α. Since node $q = \mathit{node}(\alpha')$ might be different from node p, event e may still cause a change of state in the level-k CTMC. If $q \neq p$, the level-k CTMC contains arcs from state (p, i_k) to state (q, i_k) for all $(q, i_k) \in \mathcal{M}_k$. The rates of these arcs are given by Equation 8.14, which is simply the contribution "from above", since there is no contribution at this level or at levels below.

ComputeAs(k)

if $k = K$ **then** • There is only the root node
  for each $e \in \mathcal{E}$ **do**
    $\mathbf{A}_k^e \leftarrow \mathbf{I}_1$
  end for
  return
end if
for each $e \in \mathcal{E}$ **do**
  $\mathbf{A}_k^e \leftarrow \mathbf{0}$ • Clear the old $\mathbf{A}$ matrices
end for
for each $p \in \mathcal{L}[k+1]$ **do** • Nodes p in level $k+1$
  for each i_{k+1} such that $p[i_{k+1}] \neq$null **do**
    $adjust \leftarrow \boldsymbol{\pi}_{k+1}[(p, i_{k+1})] / \Pr\{p[i_{k+1}]\}$
    for each $e \in \mathcal{E}$ **do**
      for each $j_{k+1} \in Next_{k+1}^e(i_{k+1})$ **do**
        for each $q \in \mathcal{L}[k+1]$ such that $\mathbf{A}_{k+1}^e[p, q] \neq 0$ **do**
          if $q[j_{k+1}] \neq$ null **then**
            $rate \leftarrow \mathbf{A}_{k+1}^e[p, q] \cdot \mathbf{W}_{k+1}^e[i_k, j_k]$
            $\mathbf{A}_k^e[p[i_k], q[j_k]] \leftarrow \mathbf{A}_k^e[p[i_k], q[j_k]] + rate \cdot adjust$
          end if
        end for
      end for
    end for
  end for
end for

Algorithm 8.1: Computing the $\mathbf{A}$ matrices

8.3 Algorithmic details

We wish to compute the stationary probability vectors for each of the level-k CTMCs. To compute the rates of the level-k CTMC, we must compute the $\mathbf{A}$ matrices for level k, which can be done using Algorithm 8.1. We must also compute the $\mathbf{b}$ vectors for level $k-1$, which can be done using Algorithm 8.2. However, the $\mathbf{A}$ matrices and $\mathbf{b}$ vectors require the stationary probability vectors for the CTMCs at other levels. Thus, in general, the stationary probability vector $\boldsymbol{\pi}_k$ will depend on vectors $\boldsymbol{\pi}_K, \ldots, \boldsymbol{\pi}_{k+1}, \boldsymbol{\pi}_{k-1}, \ldots, \boldsymbol{\pi}_1$. To

```
ComputeBs(k)
 1: for each p ∈ L[k] do
 2:   for each e ∈ E do
 3:     b_k^e[p] ← 0                          • Clear the b vector for node p
 4:   end for
 5:   for each i_k such that p[i_k] ≠ null do
 6:     adjust ← π_{k+1}[(p, i_{k+1})] / Pr{p}
 7:     for each e ∈ E such that Next_k^e(i_k) ≠ ∅ do
 8:       if k > 1 then
 9:         down ← b_{k-1}^e[p[i_k]]
10:       else
11:         down ← 1                          • Terminal case
12:       end if
13:       b_k^e[p] ← b_k^e[p] + adjust · down · Rate_k^e[i_k]
14:     end for
15:   end for
16: end for
```

Algorithm 8.2: Computing the $\mathbf{B}$ matrices

compute these vectors, we use the fixed-point computation of Algorithm 8.3.

8.3.1 The fixed-point computation

The overall fixed-point iteration is carried out by procedure Solve. At each iteration, Solve calls SolveLevel(k) for each level k after having updated $\mathbf{A}$ and $\mathbf{b}$. To compute $\boldsymbol{\pi}_k$, SolveLevel(k) first computes the rates of the level-k CTMC using Equation 8.10 by calling ComputeMC(k) (not shown). Then, an iterative technique is used (we use Gauss-Seidel with under-relaxation) to compute the stationary probability vector $\boldsymbol{\pi}_k$ of the level-k CTMC, given the current value of its transition rate matrix $\mathbf{R}_{\mathcal{M}_k}$. For efficiency, we use the final iterate $\boldsymbol{\pi}_k$ of the previous fixed-point iteration as the next initial guess in the Gauss-Seidel solution in the next fixed-point iteration. By doing so, the number of Gauss-Seidel iterations required to solve the level-k CTMC tends to decrease from one fixed-point iteration

```
SolveLevel(k)
 1: ComputeMC(k)
 2: compute the probability vector solution of π_k · Q_k = 0
 3: compute the probability Pr{p} of each node p ∈ L[k]
```

```
Solve()
 1: set up the structures to store A and b
 2: initialize local stationary probabilities to some guess
 3: while not converged do
 4:    for k ← 1 to K do
 5:       ComputeBs(k)
 6:    end for
 7:    for k ← K down to 1 do
 8:       ComputeAs(k)
 9:       SolveLevel(k)
10:    end for
11: end while
```

Algorithm 8.3: Our fixed-point iteration

to the next. Indeed, we stop the fixed-point iterations when all K CTMC solutions require just one Gauss-Seidel iteration, as this implies that the transition rate matrix $\mathbf{R}_{\mathcal{M}_k}$ has not changed enough to require a change (above a given relative precision ϵ) in the stationary solution $\boldsymbol{\pi}_k$.

Multiple fixed-point iterations are required only when there is a cyclic dependency among the synchronizing events. This occurs when a local state i_k for submodel k affects a rate in $\mathbf{R}_{\mathcal{M}_l}$, and a local state i_l for submodel l affects a rate in $\mathbf{R}_{\mathcal{M}_k}$, for $k \neq l$. If the dependency is only one-way and the submodels are ordered accordingly, then our algorithm will not require multiple iterations. Instead, the second time it computes the vectors $\boldsymbol{\pi}_k$, they will not change (beyond the relative precision ϵ), since no transition rate matrix $\mathbf{R}_{\mathcal{M}_k}$ changes after the first update.

8.3.2 Data structures

For a given level k, we must conceptually represent $|\mathcal{E}|$ matrices $\mathbf{A}_k^e \in \mathbb{R}^{\mathcal{L}[k]\times\mathcal{L}[k]}$, one for each event e. As these matrices are typically quite sparse, in practice we use a single matrix per level in which only the non-zero elements are stored, with an associated event index. Hence, an entry is a list of pairs (real-value, event-index). For instance, the $\mathbf{A}$ matrix would be equivalent to the diagram in Figure 8.6 with real values associated with each arc in addition to the event. Since only the rate information changes during the fixed-point iterations, we can build the sparse structure of these matrices once during initialization, and simply adjust the real-valued component during the iterations. As discussed earlier, we do not need to store $\mathbf{A}_k^e$ for $k \geq \mathit{First}(e)$, since these matrices are identities.

We must also compute and store the value of $\mathbf{b}$ for each (event, node) pair. Our implementation uses a vector of size $\mathcal{E}_{synch}$ per node where $\mathcal{E}_{synch}$ is the set of synchronizing events, since the $\mathbf{b}$ values are always 1 for local events. We could also use a vector of size $|\{e \in \mathcal{E} : \mathit{First}(e) > k > \mathit{Last}(e)\}|$ for each node at level k, provided we maintain appropriate indexing information. This is because we do not need to store values for $\mathbf{b}$ for event e at level k when $k \leq \mathit{Last}(e)$, since those values are always 1, nor do we need to store values when $k \geq \mathit{First}(e)$, since e will not cause a state change in this case. Of course, for each level-k CTMC, we must also store its stationary probability vector $\boldsymbol{\pi}_k \in \mathbb{R}^{|\mathcal{M}_k|}$, and a vector of size $|\mathcal{L}_k|$ containing $\Pr\{p\}$ for each node p at that level.

8.3.3 Computing measures

Once our fixed-point computation has converged, we can approximate the measures of interest for our model. The straightforward way to compute a measure is to sum the

product of a state's probability and that state's reward over all the reachable states. That is, we must compute

$$M = \sum_{\forall \mathbf{i} \in \mathcal{S}} \boldsymbol{\pi}[\mathbf{i}] \cdot Reward(\mathbf{i}) \tag{8.16}$$

where M is our measure of interest. However, performing the sum in Equation 8.16 has a computational cost of $O(|\mathcal{S}|)$, which can be extremely large.

We now show how to efficiently compute our measure of interest, assuming the reward function can also be expressed in a product form:

$$Reward((i_K, \ldots, i_1)) = \prod_{k=1}^{K} Reward_k(i_k). \tag{8.17}$$

By substituting Equation 8.7 and Equation 8.17 into Equation 8.16, we obtain

$$\begin{aligned} M &= \sum_{\forall \mathbf{i} \in \mathcal{S}} \boldsymbol{\pi}[\mathbf{i}] \cdot Reward(\mathbf{i}) \\ &= \sum_{\forall \mathbf{i} \in \mathcal{S}} \prod_{k=1}^{K} \frac{\boldsymbol{\pi}_k[(p_k, i_k)]}{\Pr\{p_k\}} \cdot \prod_{k=1}^{K} Reward_k(i_k) \\ &= \sum_{\forall \mathbf{i} \in \mathcal{S}} \prod_{k=1}^{K} \frac{\boldsymbol{\pi}_k[(p_k, i_k)] \cdot Reward_k(i_k)}{\Pr\{p_k\}}. \end{aligned} \tag{8.18}$$

We can compute Equation 8.18 recursively by computing the measure for each node p:

$$M(p) = \frac{1}{\Pr\{p\}} \cdot \sum_{\forall i_k : p[i_k] \neq \mathsf{null}} \boldsymbol{\pi}_k[(p_k, i_k)] \cdot Reward_k(i_k) \cdot M(p[i_k])$$

where $M = M(root)$ and the recursion terminates with $M(one) = 1$. Thus the computational requirement for computing measures that can be expressed in a product form depends on the number of nodes in the MDD representing $\mathcal{S}$, instead of the number of states.

8.4 Product-form models

We now show that our approximation algorithm performs an exact aggregation, and thus produces exact results, if the model under study possesses a product-form solution. For simplicity, we limit our proof [1] to the case of a single-class, product-form, closed queueing network with K queues and N customers. Since our only assumption is that of Equation 8.6, all we have to show is that the assumption holds for the case of our product-form queueing network.

A reachable state of our queueing network is one in which the sum of the customers at all K queues is exactly N:

$$\mathcal{S} = \left\{ \mathbf{i} = (i_K, \ldots, i_1) : \sum_{k=1}^{K} i_k = N \right\}.$$

If we use a partition where each queue is its own submodel, then the MDD representation for $\mathcal{S}$ will contain a single node at level K and $N+1$ nodes at all other levels, where node $n+1$ at level k means that there are n customers at levels K through $k+1$. Figure 8.10 shows an example with 4 queues and 4 customers. In the figure, the left-most nodes correspond to the case where no customers have yet been placed, and the right-most nodes correspond to the case where all customers have been placed. Thus, for any path $\alpha = (i_K, \ldots, i_{k+1})$ leading to node n at level k, we have $i_K + \cdots + i_{k+1} = n$.

Since our model has a product-form solution, there exist K functions f_k and a constant G such that [7]

$$\forall \mathbf{i} \in \mathcal{S}, \quad \boldsymbol{\pi}[\mathbf{i}] = \frac{\prod_{k=1}^{K} f_k(i_k)}{G}.$$

[1] A preliminary version of this proof originally appeared in [69], and I am greatly indebted to Professor Susanna Donatelli of the Universitá di Torino, Italy for her help in preparing it.

We can then rewrite $\Pr\{i_k|\alpha\}$ as

$$\begin{aligned}
\Pr\{i_k|i_K,\ldots,i_{k+1}\} &= \frac{\Pr\{i_K,\ldots,i_k\}}{\Pr\{i_K,\ldots,i_{k+1}\}} \\
&= \frac{\displaystyle\sum_{i_{k-1},\ldots,i_1:i_K+\cdots+i_1=N} \frac{f_K(i_K)\cdots f_1(i_1)}{G}}{\displaystyle\sum_{i_k,\ldots,i_1:i_K+\cdots+i_1=N} \frac{f_K(i_K)\cdots f_1(i_1)}{G}} \\
&= \frac{\dfrac{f_K(i_K)\cdots f_{k+1}(i_{k+1})}{G} \displaystyle\sum_{i_{k-1},\ldots,i_1:i_K+\cdots+i_1=N} f_k(i_k)\cdots f_1(i_1)}{\dfrac{f_K(i_K)\cdots f_{k+1}(i_{k+1})}{G} \displaystyle\sum_{i_k,\ldots,i_1:i_K+\cdots+i_1=N} f_k(i_k)\cdots f_1(i_1)} \\
&= \frac{\displaystyle\sum_{i_{k-1},\ldots,i_1:n+i_k+\cdots+i_1=N} f_k(i_k)\cdots f_1(i_1)}{\displaystyle\sum_{i_k,\ldots,i_1:n+i_k+\cdots+i_1=N} f_k(i_k)\cdots f_1(i_1)} \qquad (8.19)
\end{aligned}$$

where $n = i_K + \cdots + i_{k+1}$. Since Equation 8.19 does not directly depend on α, we have

$$i_K + \cdots + i_{k+1} = i'_K + \cdots + i'_{k+1} \quad \Longrightarrow \quad \Pr\{i_k|i_K,\ldots i_{k+1}\} = \Pr\left\{i_k|i'_K,\ldots,i'_{k+1}\right\}.$$

In other words, for any paths α and α' leading to node $n+1$ at level k, we have $\Pr\{i_k|\alpha\} = \Pr\{i_k|\alpha'\}$. From this, we can show that $\Pr\{i_k|\alpha\} = \Pr\{i_k|node(\alpha)\}$, which is the assumption of Equation 8.6.

It is important to note that the above result is intended as a validation of our technique. We are not suggesting that the iterative solution of K CTMCs is necessarily a good way to analyze a product-form model.

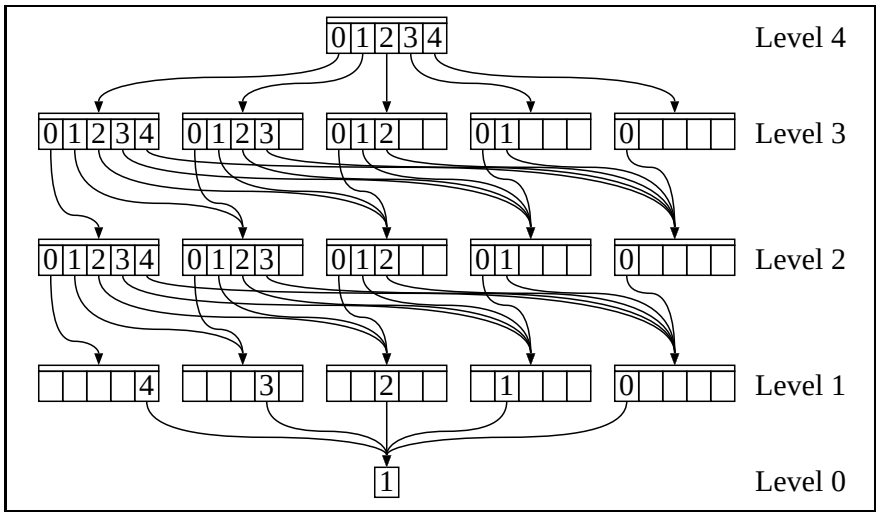


Figure 8.10: MDD for a product-form network with 4 queues and 4 customers

8.5 Experimental results

In this section, we evaluate the proposed approximation technique on a set of experiments. Two small examples are considered initially to observe the effect of synchronization: a fork and join model and a juggling server model. We then investigate the Kanban and FMS models described in Appendix B. All experiments were performed on a 400Mhz Pentium-II workstation with 384Mb of main memory, under the Linux operating system, and without the use of virtual memory. Our approximation uses Gauss-Seidel with a relaxation parameter of $\omega = 0.95$ to compute the stationary probability vectors for each level, stopping when the relative error between subsequent solution vectors is less than 10^{-6}. When comparison is possible, we use exact results obtained using solution algorithms described in earlier chapters. In each case, we report the relative error in aggregated measures.

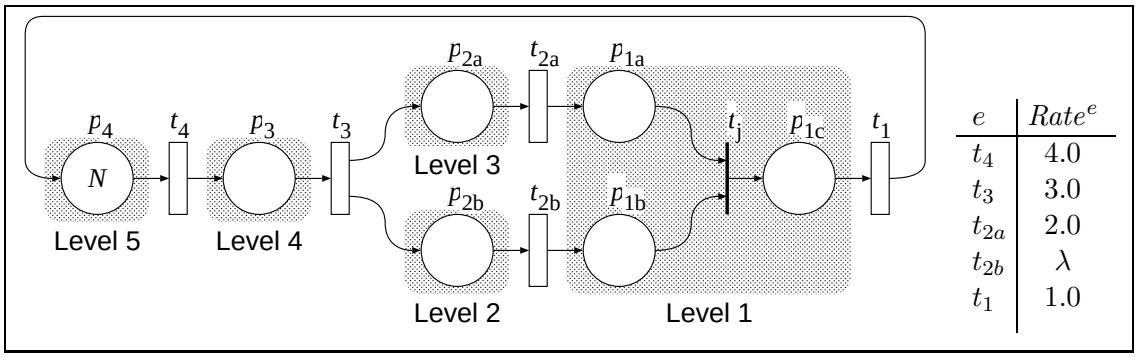


Figure 8.11: A Fork-join model

8.5.1 Fork and join model

Figure 8.11 depicts a Petri net where jobs undergo a certain sequence of activities (transitions t_1, t_4 and t_3), followed by a parallel execution (transitions t_{2a} and t_{2b}) and a join (transition t_j). All transitions have exponentially-distributed delays as indicated in the figure, except for transition t_j, which is immediate.

The parameters of the model are the number of jobs N in the system and the rate λ of transition t_{2b}. In the experiments, we vary the values of both parameters, and assume the model is partitioned into five components as shown. We compute the expected number of tokens in each place and the throughput of each transition. The size of the state space obviously increases with the number of circulating jobs; Table 8.1 lists the total number of states in the complete model (column "$|\mathcal{S}|$") and the sizes of the Markov chains used by the approximation for each level k (column "$|\mathcal{M}_k|$"). For the largest case, $N = 40$, there is about one order of magnitude between the size of the whole system and that of the largest Markov chain solved by our approximation. Much larger savings are realized for larger values of N.

N	$\lvert\mathcal{S}\rvert$	$\lvert\mathcal{M}_5\rvert$	$\lvert\mathcal{M}_4\rvert$	$\lvert\mathcal{M}_3\rvert$	$\lvert\mathcal{M}_2\rvert$	$\lvert\mathcal{M}_1\rvert$
10	1,716	11	66	66	506	121
20	19,481	21	231	231	3,311	411
30	87,296	31	496	496	10,416	961
40	259,161	41	861	861	23,821	1,681

Table 8.1: CTMC sizes for the fork-join model

fixed-point	Gauss-Seidel iterations ($\omega = 0.95$)				
iteration #	level 5	level 4	level 3	level 2	level 1
1	110	293	526	3,082	1,965
2	90	161	230	1	1
3	62	86	1	1	1
4	20	1	1	1	1
5	1	1	1	1	1

Table 8.2: Iterations for the fork-join model, $N = 40$, $\lambda = 1$

Another interesting measure of complexity is the number of iterations required by the fixed-point algorithm. In each case, only five fixed-point iterations are required. Table 8.2 reports the number of Gauss-Seidel iterations required for the solution of each level-k CTMC. In this model, the solution for levels 1 and 2 converges very quickly, as indicated by a single Gauss-Seidel iteration after the first step.

In all our experiments for this model, the highest relative error occurs in the aggregated measure of the expected number of tokens in place p_{1c}. Figure 8.12 shows the relative error for fixed values of λ against the number of jobs N, and Figure 8.13 shows the relative error for fixed values of N against the rate λ. For this model, the approximation is quite accurate: usually within 0.01%. The maximum observed throughput error for transitions was only 0.02%.

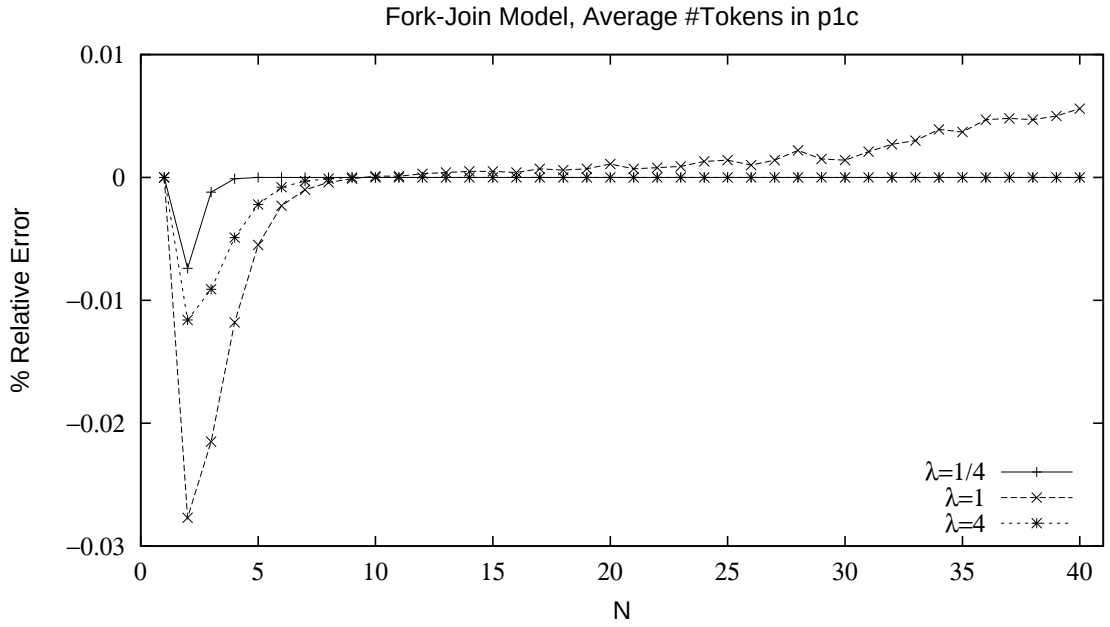


Figure 8.12: Error for the fork-join model against N

8.5.2 Load-dependent service model

Figure 8.14 depicts a Petri net where two stations share S servers. Jobs in the system progress cyclically through four activities (transitions t_4, t_3, t_2, and t_1), and S servers move between transitions t_2 and t_1, through transitions i_{21} and i_{12}. All delays are exponentially distributed with rates as shown. Note that activity t_3 has infinite servers with rate 1, activity t_2 has $\#s_2$ servers (i.e., the number of servers is given by the number of tokens in place s_2), and activity t_1 has $\#s_1$ servers. Transitions i_{21} and i_{12} control the movement of the servers. This is done by disabling transition i_{21} whenever $\#p_2 > \#p_1$, and disabling transition i_{12} whenever $\#p_1 > \#p_2$.

The parameters of the model are the number of jobs N in the system, the number of servers S, and the rate λ of servers for activity t_1. We compute the expected number of

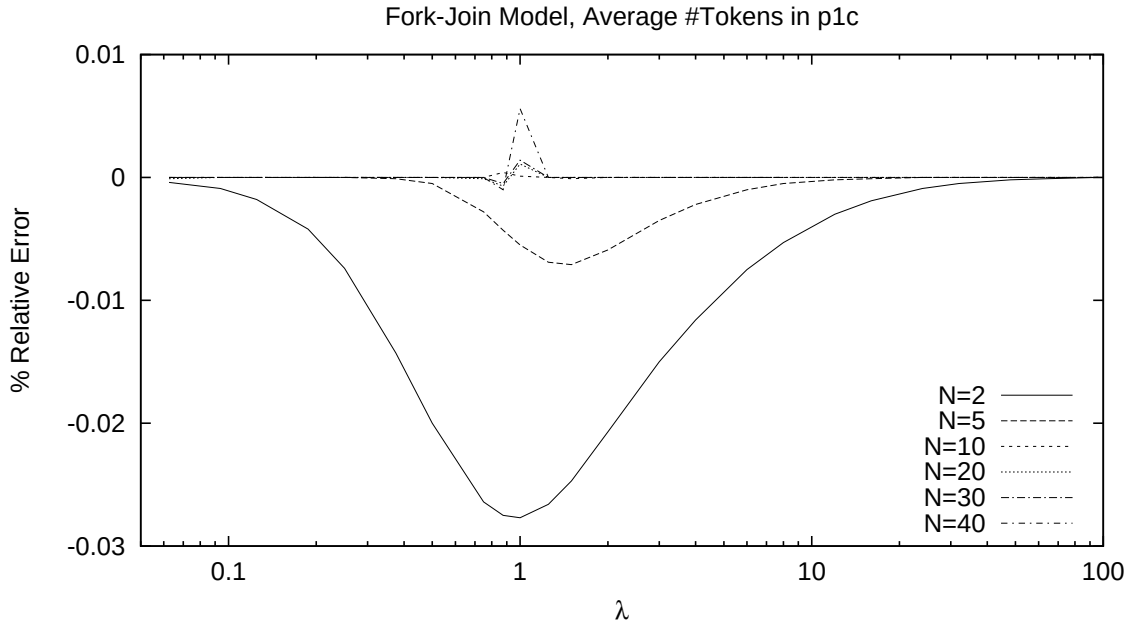
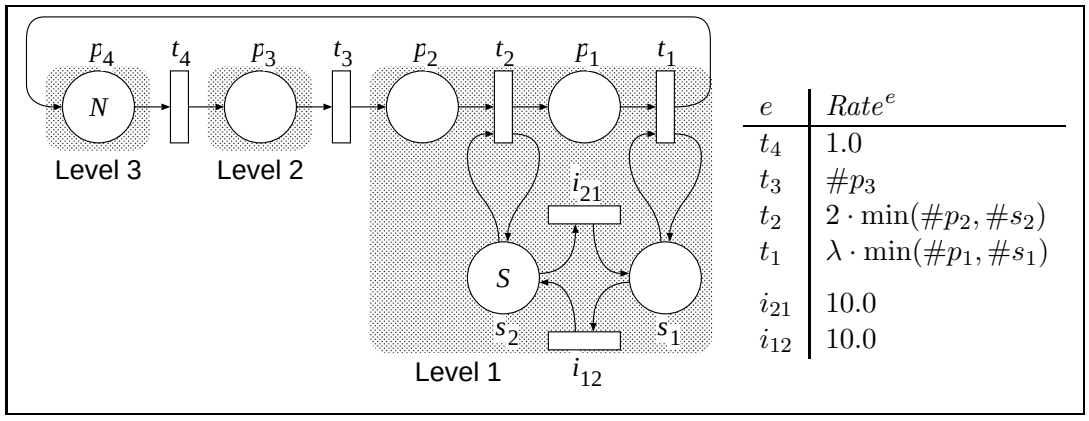


Figure 8.13: Error for the fork-join model against λ



e	$Rate^e$
t_4	1.0
t_3	$\#p_3$
t_2	$2 \cdot \min(\#p_2, \#s_2)$
t_1	$\lambda \cdot \min(\#p_1, \#s_1)$
i_{21}	10.0
i_{12}	10.0

Figure 8.14: A load-dependent service model

tokens in each place and the throughput of each transition. The model is partitioned into three components as shown in Figure 8.14. The size of the state space increases with the number of circulating jobs and the number of servers. Table 8.3 lists the total number of

N	S	$\|\mathcal{S}\|$	$\|\mathcal{M}_3\|$	$\|\mathcal{M}_2\|$	$\|\mathcal{M}_1\|$
10	1	572	11	66	132
20	1	3,542	21	231	462
30	1	10,912	31	496	992
40	1	24,682	41	861	1,722
10	2	858	11	66	198
20	2	5,313	21	231	693
30	2	16,368	31	496	1,488
40	2	37,023	41	861	2,583
10	3	1,144	11	66	264
20	3	7,084	21	231	924
30	3	21,824	31	496	1,984
40	3	49,364	41	861	3,444

Table 8.3: CTMC sizes for the load-dependent service model

f-pt	Gauss-Seidel iterations ($\omega = 0.95$)								
iter	1 server			2 servers			3 servers		
#	level 3	level 2	level 1	level 3	level 2	level 1	level 3	level 2	level 1
1	199	446	593	637	1,651	1,076	694	1,064	462
2	329	564	2	667	1,217	5	230	385	1
3	261	1	1	364	2	1	134	1	1
4	1	1	1	1	1	1	1	1	1

Table 8.4: Iterations for the load-dependent service model, $N = 40$, $\lambda = 1$

states in the complete model (column "$|\mathcal{S}|$") and the sizes of the Markov chains used by the approximation for each level k (column "$|\mathcal{M}_k|$"). In all the experiments performed, the approximation converges in four iterations, with a pattern very similar to the fork and join model. Table 8.4 reports the number of Gauss-Seidel iterations required for the solution of each level-k CTMC, for the case of $N = 40$.

In all our experiments for this model, the highest relative error occurs in the aggregated measure of the average number of tokens in place p_4. Figure 8.15 shows the relative error for this measure against the number of jobs N, for different values of S and λ. As seen in

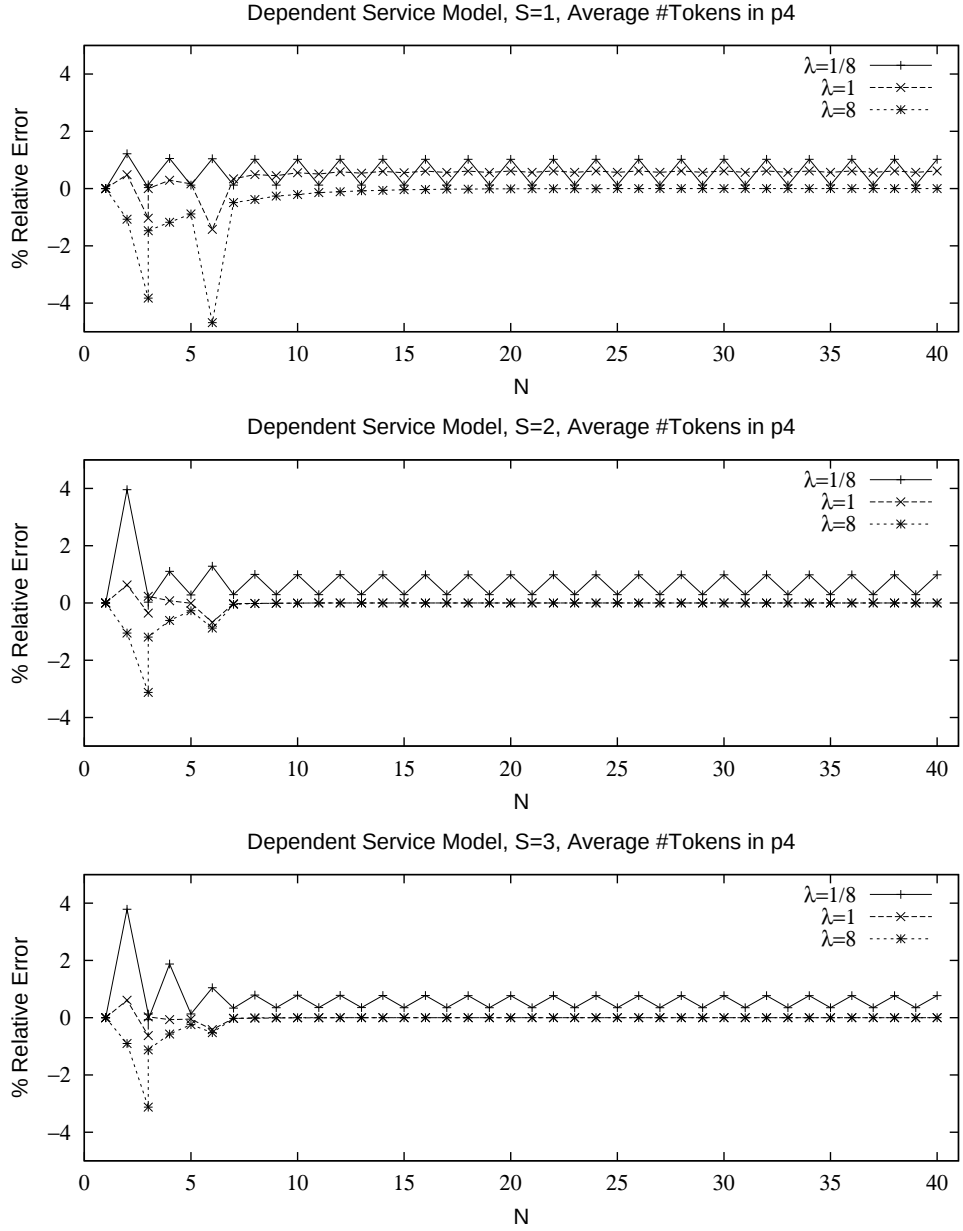


Figure 8.15: Error for the load-dependent service model against N

the plots, the highest errors occur for the case of $S = 1$; as S increases, the approximation becomes more accurate. Figure 8.16 shows the relative error for the same measure against the rate λ, for different values of N and S. As we can see, the highest error occurs when

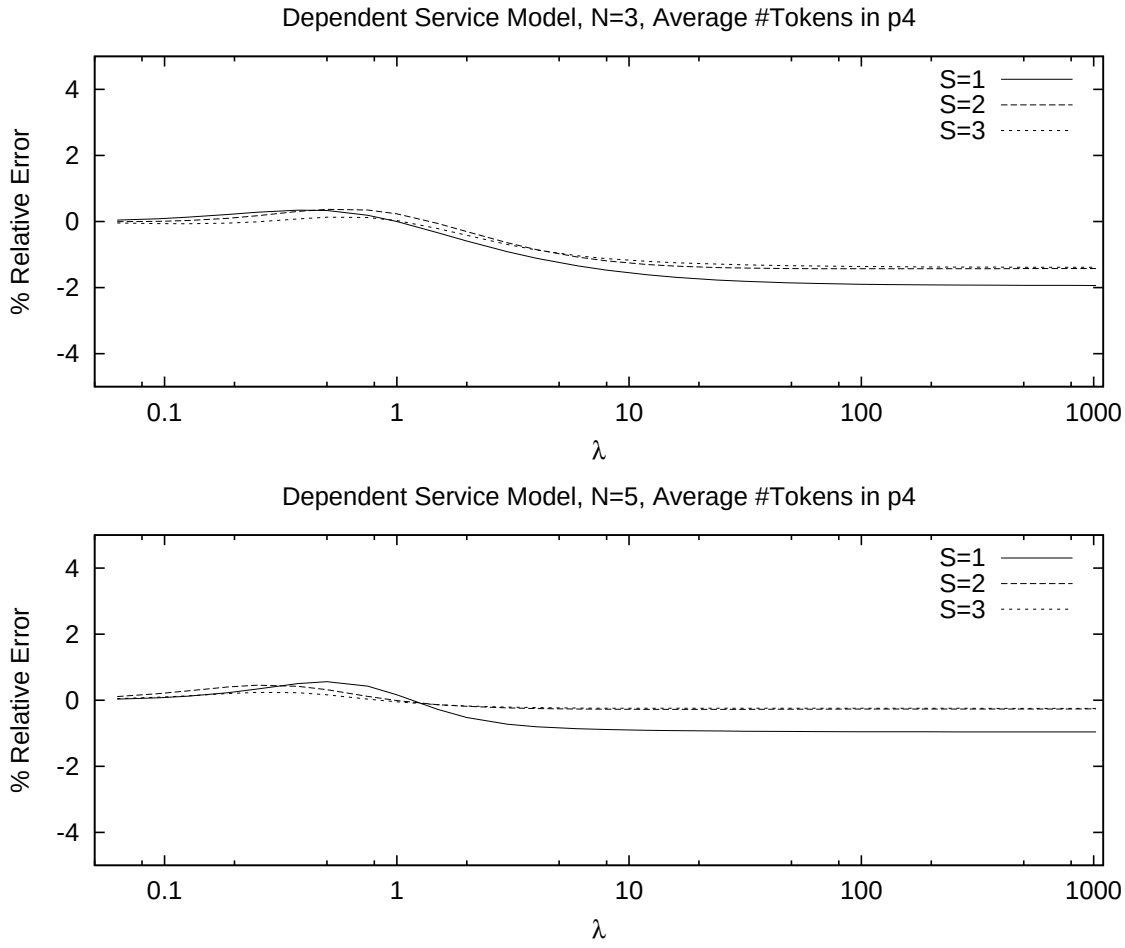


Figure 8.16: Error for the load-dependent service model against λ

the service rate is high and the number of servers is small. In general, we have observed that our approximation tends to have a high error due to synchronizing events whose rates can change significantly; this tends to occur when a synchronizing event has a relatively high, state-dependent rate. The maximum observed throughput error for transitions was 2.3% for this model.

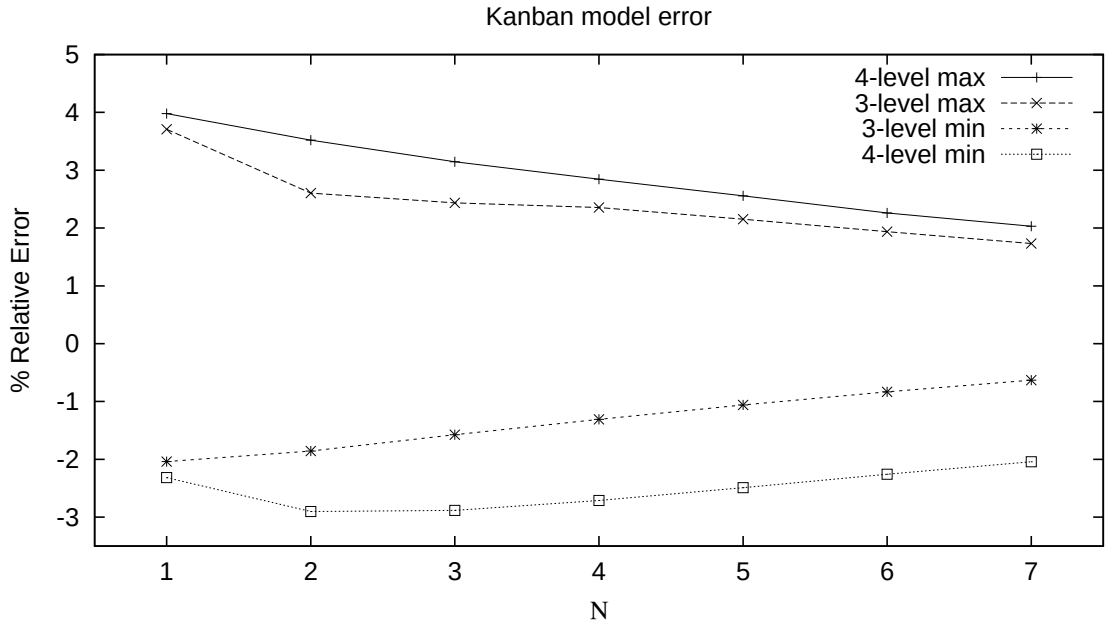


Figure 8.17: Relative error for the Kanban model

8.5.3 Kanban model

We apply our approximation to the Kanban model, using both a four-component decomposition and a three-component decomposition. We compute the relative error for the measurements of the average number of tokens in each place. The relative error for this model is shown in Figure 8.17. Exact solution (and thus relative error calculation) is possible only for N up to 7. Figure 8.17 shows the maximum and minimum of these relative errors for both the four-level and three-level decompositions. The relative error of the computed transition throughputs is quite small for this model (always less than 1.5%), and thus is not shown. As the plot indicates, the relative errors tend to decrease as the number of jobs N increases. For this particular model, the three-level decomposition gives more accurate results than the four-level decomposition.

		3-level approx.			4-level approx.	
	# states	CTMC sizes		CPU time	CTMC sizes	CPU time
N	$\|\mathcal{S}\|$	$\|\mathcal{M}_1\|, \|\mathcal{M}_4\|$	$\|\mathcal{M}_{2,3}\|$	(seconds)	$\|\mathcal{M}_1\|, \ldots, \|\mathcal{M}_4\|$	(seconds)
6	1.13×10^7	84	1,596	8.5	84	1.4
12	5.52×10^9	455	26,663	264.9	455	15.4
18	2.82×10^{11}	1,330	159,334	2,221.8	1,330	65.0
24	5.07×10^{12}	2,925	592,605	10,887.2	2,925	191.9
30	4.99×10^{13}	—	—	—	5,456	462.5
42	1.66×10^{15}	—	—	—	14,190	1,865.8
54	2.36×10^{16}	—	—	—	29,260	5,533.2
66	1.99×10^{17}	—	—	—	52,394	13,424.5

Table 8.5: CTMC sizes for the Kanban model

Table 8.5 shows the number of states in the exact CTMC, the sizes of each of the level-k CTMCs, and the total CPU time required by our approximation (including the CPU time required to generate $\mathcal{S}$) for both the three-level and four-level decompositions. The size of the CTMC for the combined submodels 2 and 3 ($|\mathcal{M}_{2,3}|$) makes the three-level decomposition more expensive in terms of both memory and CPU requirements. The four-level decomposition can be used to analyze much larger systems. The number of Gauss-Seidel iterations at each of the required fixed-point iterations of our algorithm is shown in Table 8.6 for the case $N = 66$. Note that the number of Gauss-Seidel iterations drops significantly after the first few iterations of our algorithm.

8.5.4 Flexible manufacturing system (FMS) model

We analyze the FMS model using both a 20-component decomposition and a 5-component decomposition. We compute the throughput of transitions $tP1, tP2, tP12$, and $tP3$, and the "productivity" measure of the model (see Appendix B). The relative error is the same for both decompositions. Exact solution is possible only for N up to 6. We show the relative

fixed-point	Gauss-Seidel iterations ($\omega = 0.95$)			
iteration #	level 4	level 3	level 2	level 1
1	1,965	2,919	2,602	2,277
2	1,169	1,649	1,371	1,143
3	700	854	491	2
4	1	223	100	13
5	1	25	13	87
6	1	7	5	2
7	1	3	2	1
8	1	1	1	1

Table 8.6: Iterations for the Kanban model, $N = 66$

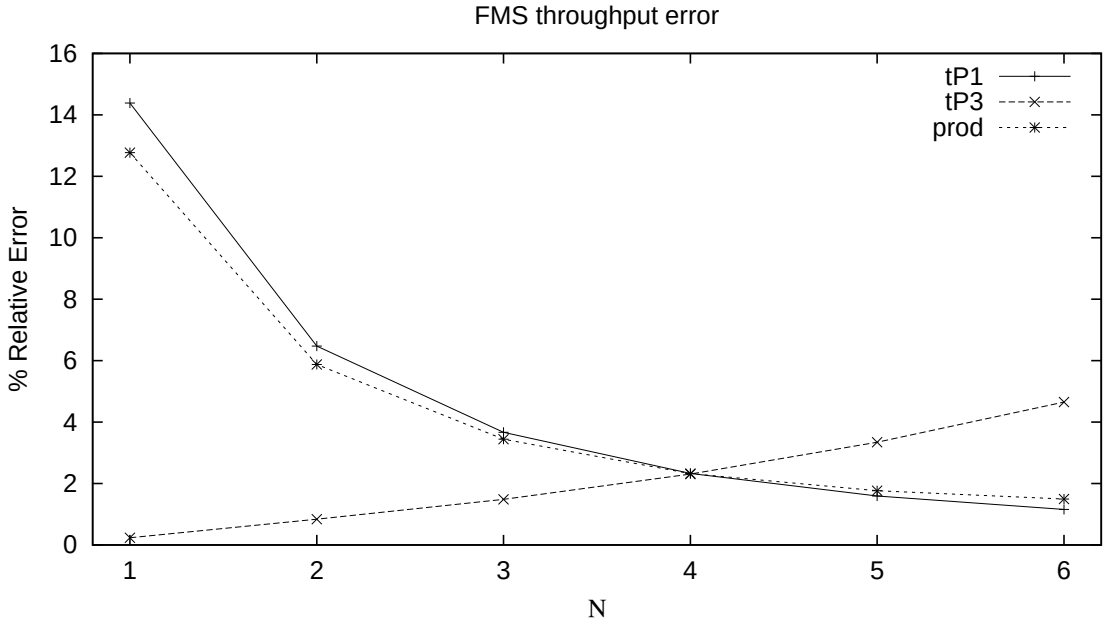


Figure 8.18: Relative error for the FMS model

error for the throughput of transitions $tP1$ and $tP3$ and for the productivity in Figure 8.18. The relative errors for the throughput of transitions $tP2$ and $tP12$ are similar to the error for $tP1$, and thus are not included in the plot.

Table 8.7 shows the number of states in the exact CTMC, the sizes of the smallest and

N	# states $\lvert\mathcal{S}\rvert$	5-level approx. CTMC sizes min	5-level approx. CTMC sizes max	5-level approx. CPU time (seconds)	20-level approx. CTMC sizes min	20-level approx. CTMC sizes max	20-level approx. CPU time (seconds)
5	2.90×10^6	16	2,268	20.0	6	406	24.0
10	2.50×10^9	31	55,055	852.2	11	3,861	411.7
15	2.17×10^{11}	46	434,112	11,660.6	16	16,116	1,960.7
21	1.07×10^{13}	—	—	—	22	55,154	9,848.2
27	2.16×10^{14}	—	—	—	28	141,085	18,792.2
33	2.48×10^{15}	—	—	—	34	301,665	35,273.8

Table 8.7: CTMC sizes for the FMS model

fixed-point iteration #	Gauss-Seidel iterations min iters	avg iters	max iters
1	60	1,531	10,000
2	36	292	836
10	31	146	275
20	25	227	1,549
30	1	56	128
40	1	10	82
50	1	2	3
71	1	1	1

Table 8.8: Iterations for the FMS model, $N = 33$

largest CTMC, and the total CPU time required by our approximation (including the CPU time required to generate $\mathcal{S}$) for both the 5-level and 20-level decompositions. As with the Kanban model, the decomposition that produces smaller CTMCs can be used to analyze larger systems. In this case, we can analyze the FMS model for N up to 33 if we use the 20-level decomposition.

The minimum, average, and maximum number of Gauss-Seidel iterations are shown in Table 8.8 for each of the required fixed-point iterations of our algorithm, for the case $N = 33$. Note that we only allow 10,000 iterations for Gauss-Seidel to converge; thus, we

see from the table that for at least one CTMC in the first fixed-point iteration, Gauss-Seidel failed to converge. As with the other models, the number of Gauss-Seidel iterations drops quickly after the first few fixed-point iterations of our algorithm. Note that for the last 30 fixed-point iterations, the average number of Gauss-Seidel iterations required to solve the CTMCs for each level is 10 or fewer.

8.6 Conclusion

Using data structures from the previous chapters, we can obtain compact and exact representations for the set of reachable states $\mathcal{S}$ and the transition rate matrix $\mathbf{R}$ of the underlying CTMC of a model. However, to analyze the CTMC, we still need to explicitly store the solution vector $\boldsymbol{\pi}$, which is of size $|\mathcal{S}|$. For large models, the memory requirements of $\boldsymbol{\pi}$ can easily exceed the available memory on a modern workstation.

We presented a technique for approximating stationary measures for *Kronecker-product-form* models. By using decision diagrams and Kronecker operators, we build K Markov chains corresponding to approximate aggregations of the exact process. This allows us to obtain accurate approximations for large models whose exact numerical solution is infeasible. As a special case, we have shown that our algorithm provides exact results for models possessing a product-form solution. Unlike other approximations based on model decomposition, our technique correctly assigns zero probabilities to unreachable states. In our experience, our technique produces more accurate results but tends to be more expensive (i.e., requires the solution of larger CTMCs) than other model-based decompositions. Our technique is ultimately limited by the sizes of the CTMCs to be solved at each level, which

depend on the decision diagram encoding of the state space. An additional strength of our technique is that experimenting with different decompositions of the overall model simply involves defining a new partition of the model.

Chapter 9

Applications

In this chapter we apply our techniques from previous chapters to some interesting practical examples. Section 9.1 shows an example of how a distributed algorithm can be modeled and how certain properties can be verified. Section 9.2 presents an example in which we evaluate the performance of a cluster of web servers.

9.1 Distributed algorithm verification

In this section, we show by example how distributed algorithms can be modeled and verified. We do so by investigating termination detection algorithms. This problem is fairly well studied in the literature (see for instance [40, 67, 77, 87, 101]), and sophisticated solutions are available. The simple algorithms we present are intended for illustration purposes only.

The termination detection problem arises when several communicating machines are performing a computation. Formally, we have N machines that are either *active* or *passive*. The active machines are busy computing, and may occasionally send computation messages to any of the other machines. Eventually, an active machine will complete its computation, and will then become passive. Passive machines become active only when receiving a com-

putation message. The computation is said to be *terminated* when all messages have been received and all machines are passive. Since passive machines cannot send messages, once the system has terminated, it remains terminated forever. Once the system has terminated, we would like at least one of the machines to be able to detect that this is the case. This is a non-trivial problem, because it requires a single machine to obtain global knowledge of the system. Clearly, a machine must send messages to other machines before it can safely conclude that the system has terminated.

We will consider variants of Dijkstra's algorithm [40]. By constructing a model and generating the reachable states of the model, we can determine if the algorithm incorrectly determines that the system has terminated. That is, we can conclude that it is or is not possible for the algorithm to think that the system has terminated when in fact one or more machines are still active. Another property of a candidate termination detection algorithm that we would like to verify is that the algorithm always eventually concludes that the system has terminated. That is, we want to know whether it is possible to reach a "good" state (in which the algorithm has correctly concluded that the system has terminated) from every reachable state of the model. Unfortunately, this property cannot be determined by simply generating the reachable states. This type of property can be determined from "backwards reachability" analysis; i.e., by determining all the states that can reach a set of states. We intend to investigate efficient algorithms for more general types of analysis, including this important one, in the future.

We assume that our distributed computation is running on N machines that do not fail. Machines communicate via message passing, and it is assumed that the time between the sending and receiving of a message is finite (and non-zero). We do not assume that

Rules for termination detection

1: When active, machine $i+1$ keeps the probe, otherwise it passes the probe to machine i.
2: A machine becomes "red" when it sends a message.
3: If a machine is "red", it passes a "red" probe, otherwise it does not change the probe color.
4: After passing the probe, a machine makes itself "green".
5: Machine 0 initiates the probe by passing a "green" probe to machine $N-1$.
6: When machine 0 receives the probe, it concludes that the system has terminated if the probe is "green" and machine 0 is passive. Otherwise, it initiates another probe.
7: When machine i sends a message, it increments δ_i; when machine i receives a message, it decrements δ_i.
8: When machine i receives the probe, it adds δ_{probe} to δ_i.
9: When machine i sends the probe, it sends δ_i and then resets δ_i to zero.
10: Machine 0 does not conclude that the system is terminated if $\delta_0 \neq 0$.

Algorithm 9.1: Modified termination detection

messages are received in any particular order. Since Dijkstra's algorithm as described in [40] works only for instantaneous message passing, we will attempt to modify the algorithm to handle delays between the transmission and reception of a message. Algorithm 9.1 shows our modified algorithm, described as a set of rules. The first 6 rules correspond to Dijkstra's algorithm in [40]. Machine 0 initiates a *probe*, which is used to determine if the system is terminated. The probe is passed from machine to machine, and when machine 0 receives the probe it will either conclude that the system has terminated or initiate another probe. To handle the message delays, we also have each machine keep track of the difference between the number of messages it has sent and the number it has received since the last time it saw the probe. The probe will accumulate these values. When the probe returns to machine 0, if the total accumulated difference is non-zero, then there must be a message that has not yet been received.

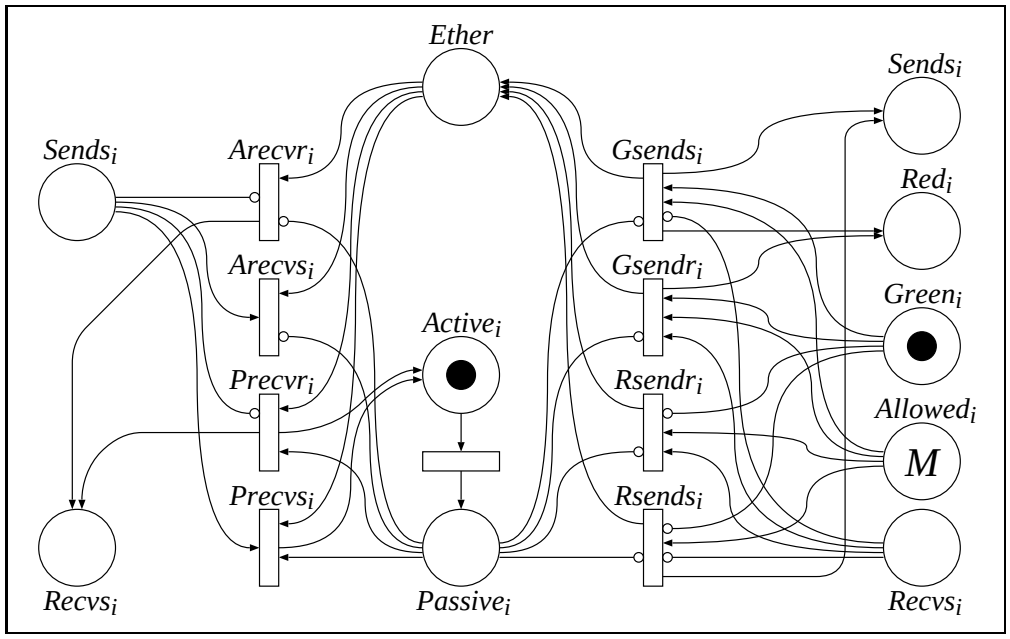


Figure 9.1: Message passing subnet for machine i, Algorithm 9.1

While the rules of Algorithm 9.1 are sufficient to completely describe a model using some "distributed algorithm formalism", we will also discuss a Petri-net model for the algorithm. We will use a subnet for each of the N machines. Figure 9.1 shows the portion of the subnet for machine i that handles the message passing behavior of the machine. Note that we must bound the number of messages each machine can send, otherwise the value of δ_i could be arbitrarily large. We allow each machine to send at most M computation messages; this corresponds to the value M in place $Allowed_i$. The places $Sends_i$ and $Recvs_i$, drawn twice for clarity, are used to represent the value of δ_i. If δ_i is positive, then the number of tokens in $Sends_i$ will be equal to δ_i, and place $Recvs_i$ will be empty. If δ_i is negative, then the number of tokens in $Recvs_i$ will be equal to $-\delta_i$, and place $Sends_i$ will be empty. If δ_i is zero, then both places will be empty.

A message is sent by transitions $Gsends_i$, $Gsendr_i$, $Rsends_i$, and $Rsendr_i$. At most

one of these four transitions will be enabled. The "G" and "R" correspond to the different behavior if the machine is "green" or "red" (places $Green_i$ and Red_i), and the "r" and "s" correspond to whether the sent message causes us to subtract one from place $Recvs_i$ or to add one to place $Sends_i$. Sent messages go across the ether (global place *Ether*), where they can be received by any machine. Note that a message could be received by its sender. A message is received by transitions $Arecvr_i$, $Arecvs_i$, $Precvr_i$, and $Precvs_i$. At most one of these four transitions will be enabled. The "A" and "P" correspond to the different behavior if the machine is "active" or "passive" (places $Active_i$ and $Passive_i$), and "r" and "s" correspond to whether the received message causes us to add one to place $Recvs_i$ or to subtract one from place $Sends_i$.

Figure 9.2 shows the portion of the subnet for machine i that handles the rules for passing the probe. Note that some of the places from Figure 9.1 appear in this subnet also. At any point in time, a machine either has a green probe (place $Gprobe_i$), has a red probe (place $Rprobe_i$), or does not have the probe (place $noprobe_i$). If the machine has a probe, then the tokens in place $Sends_i$ or $Recvs_i$ are transferred to the next machine (machine $i - 1 \bmod N$) via transitions $sendp_i$, $sendm_i$, $recvp_i$, and $recvm_i$. When δ_i has been transferred (i.e., places $Sends_i$ and $Recvs_i$ are empty) and machine i is not active, the probe is passed to the next machine.

Note that some modifications are necessary for machine 0. Initially, machine 0 has the probe, so we initially assign a token to place $Gprobe_0$ instead of place $noprobe_0$. Also, machine 0 always sends a green probe, so the arcs to place $Rprobe_{N-1}$ are directed instead to place $Gprobe_{N-1}$. Finally, some additional structure is necessary to handle the termination detection; this is shown in Figure 9.3. When the algorithm thinks the computation has

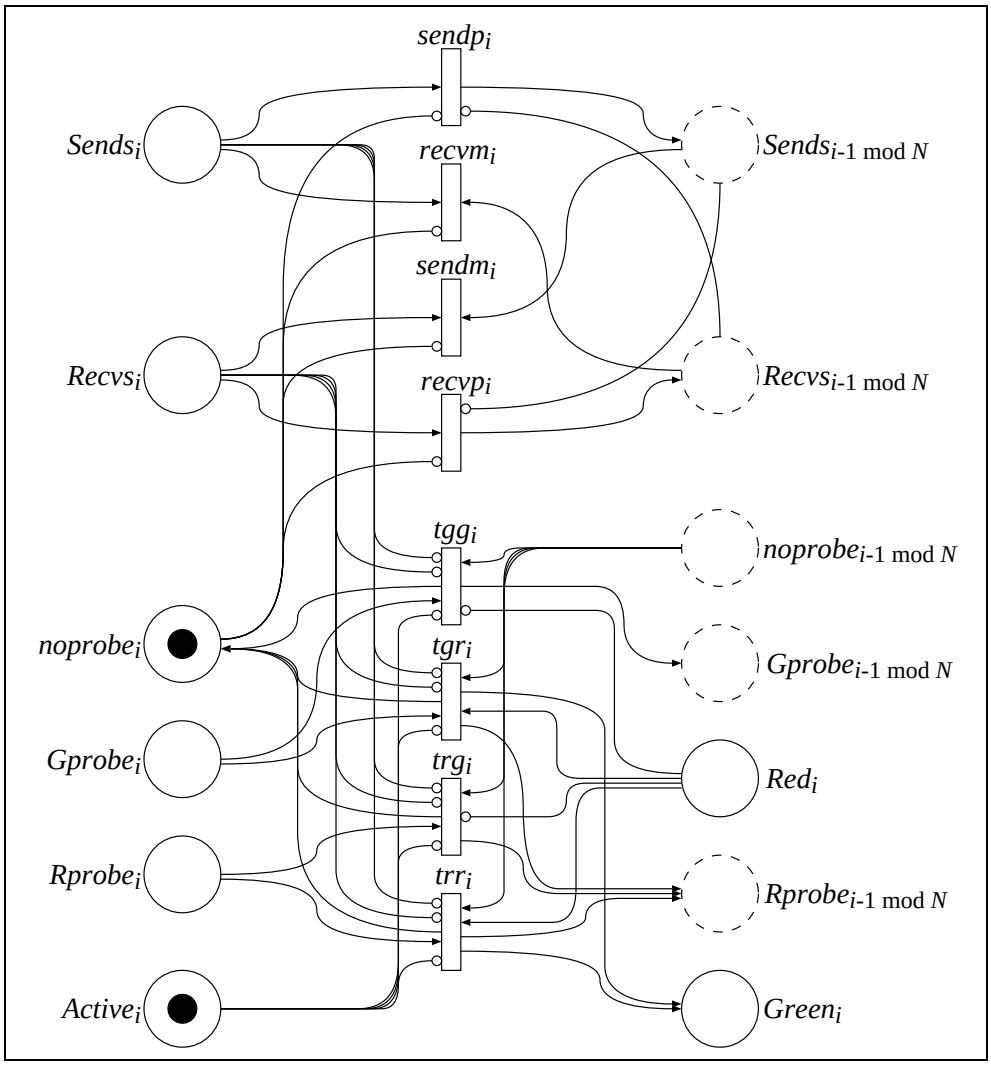


Figure 9.2: Probe subnet for machine i, Algorithm 9.1

terminated, it puts a token into place *terminated*.

Table 9.1 shows the results for the analysis of this model for different values of M and N. In the table, we show the total number of reachable states (column $|\mathcal{S}|$), whether the algorithm can conclude termination (column "Terminates"), and whether or not the algorithm can incorrectly conclude termination (column "False terminate"). To check if the algorithm can conclude termination, we see if there is any reachable state with a token

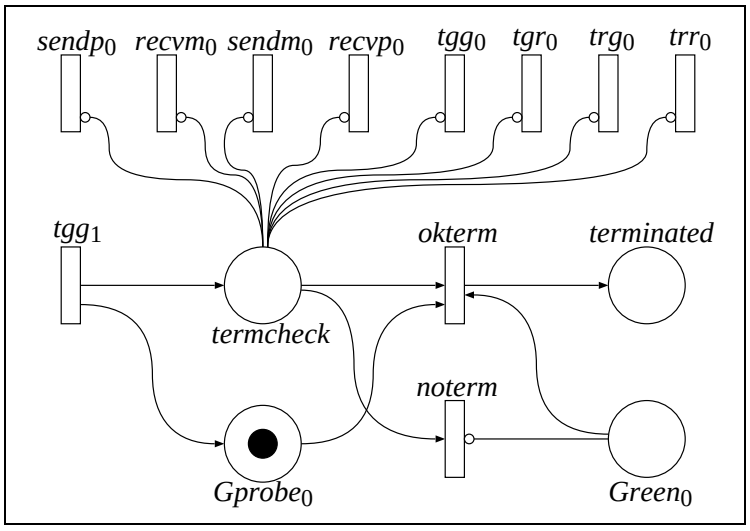


Figure 9.3: Special subnet for machine 0, Algorithm 9.1

in place *terminated*. To check if the algorithm incorrectly concludes termination, we see if any reachable states exist with a token in place *terminated* and a token in place $Active_i$ for some machine i.

As we expect, the number of reachable states grows quickly with N and M. In every case, the algorithm does conclude termination. The algorithm correctly concludes termination if there are only two machines or if no computation messages are passed. However, if more than two machines are involved in the computation, and they are allowed to send even one message, the algorithm may incorrectly conclude that the computation has terminated. The problem occurs when messages remain in the ether for a very long time. In this particular case, it is possible for a machine to send a message that remains in the ether long enough so that the probe circulates enough times that all machines become green and the probe becomes green. Then, after sending the green probe, another machine receives the old message and becomes active. This newly activated machine sends a message to machine

N	M	$\lvert\mathcal{S}\rvert$	Terminates	False terminate
2	0	8.00×10^0	yes	no
2	1	2.77×10^2	yes	no
2	2	2.15×10^3	yes	no
2	3	8.49×10^3	yes	no
2	4	2.37×10^4	yes	no
2	5	5.37×10^4	yes	no
3	0	1.60×10^1	yes	no
3	1	4.70×10^3	yes	yes
3	2	1.18×10^5	yes	yes
3	3	1.00×10^6	yes	yes
3	4	4.96×10^6	yes	yes
3	5	1.75×10^7	yes	yes
4	0	3.20×10^1	yes	no
4	1	6.95×10^4	yes	yes
4	2	5.73×10^6	yes	yes
4	3	1.07×10^8	yes	yes
4	4	9.25×10^8	yes	yes
5	0	6.40×10^1	yes	no
5	1	9.61×10^5	yes	yes
5	2	2.64×10^8	yes	yes
5	3	1.08×10^{10}	yes	yes
6	0	1.28×10^2	yes	no
6	1	1.28×10^7	yes	yes
6	2	1.18×10^{10}	yes	yes

Table 9.1: Results for verification of Algorithm 9.1

0, who receives the message and becomes passive before the probe returns. When machine 0 receives the probe, both the probe and machine 0 are still green, and δ_0 is zero. Thus machine 0 concludes that the system has terminated, but one machine is still active.

To fix Algorithm 9.1, we add an another rule:

11: If $\delta_i \neq 0$, then machine i becomes "red".

This rule forces the probe to circulate once to make sure that all messages have been received, and a second time to make sure that all machines are passive. After making the

N	M	$\lvert\mathcal{S}\rvert$	Terminates	False terminate
2	1	4.11×10^2	yes	no
2	2	2.45×10^3	yes	no
2	3	8.47×10^3	yes	no
2	4	2.19×10^4	yes	no
2	5	4.72×10^4	yes	no
3	1	8.51×10^3	yes	no
3	2	1.52×10^5	yes	no
3	3	1.07×10^6	yes	no
3	4	4.68×10^6	yes	no
3	5	1.53×10^7	yes	no
4	1	1.64×10^5	yes	no
4	2	8.82×10^6	yes	no
4	3	1.27×10^8	yes	no
4	4	9.46×10^8	yes	no
5	1	3.04×10^6	yes	no
5	2	4.97×10^8	yes	no
5	3	1.48×10^{10}	yes	no
6	1	5.49×10^7	yes	no
6	2	2.73×10^{10}	yes	no

Table 9.2: Results for verification of modified Algorithm 9.1

appropriate changes to our Petri net, we can again check that the algorithm can conclude termination and the algorithm correctly concludes termination. These results are shown in Table 9.2. Note that in some cases, the number of states is larger in Table 9.1, and in other cases, the number of states is larger in Table 9.2. On the one hand, we expect the number of states to be larger in Table 9.2, because our modified Petri net uses an additional place per machine to enforce the new rule. On the other hand, we expect the number of states to be smaller in Table 9.2, for two reasons. First, the new rule places an additional restriction on reachable states: it is no longer possible for machine i to be either red or green when $\delta_i \neq 0$. Second, the number of states in which Algorithm 9.1 has incorrectly concluded that the system has terminated can be quite large, and the modified algorithm does not reach

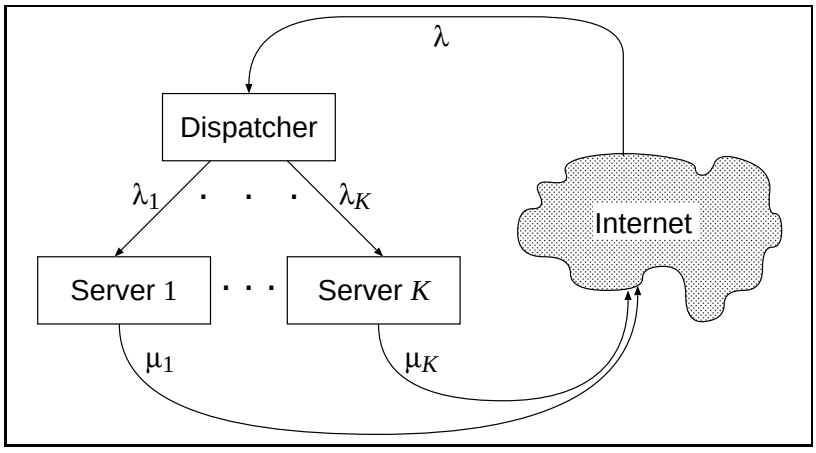


Figure 9.4: Model of a group of K Web servers

these states. Thus, as the number of allowed messages increases, the number of reachable states for Algorithm 9.1 increases relative to the number of reachable states for the modified algorithm.

9.2 Web server performance evaluation

In this section, we investigate a simple cluster of web servers, illustrated in Figure 9.4. Subscribers request information using a web browser. This portion of the system is represented in Figure 9.4, according to tradition, as a grey blob labeled "internet". Requests for information arrive with rate λ, which are handled at the front end by a dispatcher. The dispatcher sends each request to one of K back-end servers. Note that the dispatcher may have several requests in its queue, since requests may arrive while the dispatcher is sending a request. Each server k maintains its own queue of requests, and can process requests with rate μ_k. The back-end servers send the requested information to the subscriber over the internet.

We assume that the dispatcher has perfect knowledge of the queue lengths for each server. This is not unrealistic, as the time required to obtain this information can be included in the rate λ_k, which is the rate at which the dispatcher can send a request to server k. The dispatcher sends each request to the server with the fewest jobs in its queue. In case of a tie, the dispatcher sends the request to the lowest numbered server.

The system as described is an open system with infinite state space. To limit the state space to a finite size, we assume that each server can handle at most J jobs. That is, the maximum queue length for each server is J. This behavior is represented by forcing the dispatcher to drop requests when there are JK requests in the system. Since the dispatcher always balances the load perfectly, if there are JK requests in the system and the dispatcher's queue is empty, then there must be exactly J requests at each server. If a subscriber's request is dropped, an error message will appear on the browser saying that the server is busy.

The state of the system is completely described by the number of requests at the dispatcher and the number of requests at each server. However, to obtain a *Kronecker-product-form* decomposition, we need an additional state variable to keep track of the total number of requests in the system. This is because the event corresponding to an arrival of a request to the dispatcher must be disabled if the system is full. Without the extra state variable, the event would need knowledge of every state variable, which violates the *Logical-product-form* constraint.

The stationary measures of interest of this model are the probability that the system is full (i.e., the probability that a request is dropped) and the average number of requests in the system. We compute the measures of interest for a variety of values for the arrival

λ_1	= 11	μ_1	= 2
λ_2	= 10.8	μ_2	= 1.8
λ_3	= 10.6	μ_3	= 1.6
λ_4	= 10.4	μ_4	= 1.4
λ_5	= 10.2	μ_5	= 1.2

Table 9.3: Rates for the model of Figure 9.4 with 5 servers

rate λ, using exact analysis (i.e., generating and analyzing the complete underlying CTMC), the approximation described in the previous chapter, and discrete-event simulation. For exact and approximate analysis, we use Gauss-Seidel with an under-relaxation parameter $\omega = 0.95$, stopping when the relative difference between successive solution vectors is within 10^{-5}. Our discrete-event simulation uses batched-means with 16 runs per batch and 400 batches, for a total of 6,400 simulation runs.

First, we assume that we have $K = 5$ servers that can each handle $J = 5$ jobs. The CTMC for this model has $104,976$ states, so we can solve it exactly. We also use our approximation and discrete-event simulation. The rates for this case are shown in Table 9.3, except for the arrival rate λ, which is a parameter of the model. Each rate λ_k captures the time required for the dispatcher to decide to send a request to server k, the transmission time of the request, and the time required for server k to receive the request and place it in the queue. The differences between the λ_k rates are due to hardware differences. Each rate μ_k captures the time required for the server to process the request and send the requested information to the subscriber. Note that we assume only about one order of magnitude difference between the rate at which the dispatcher can distribute requests and the rate at which requests can be served.

In Figure 9.5 and Figure 9.6, we respectively plot the probability of a full system and

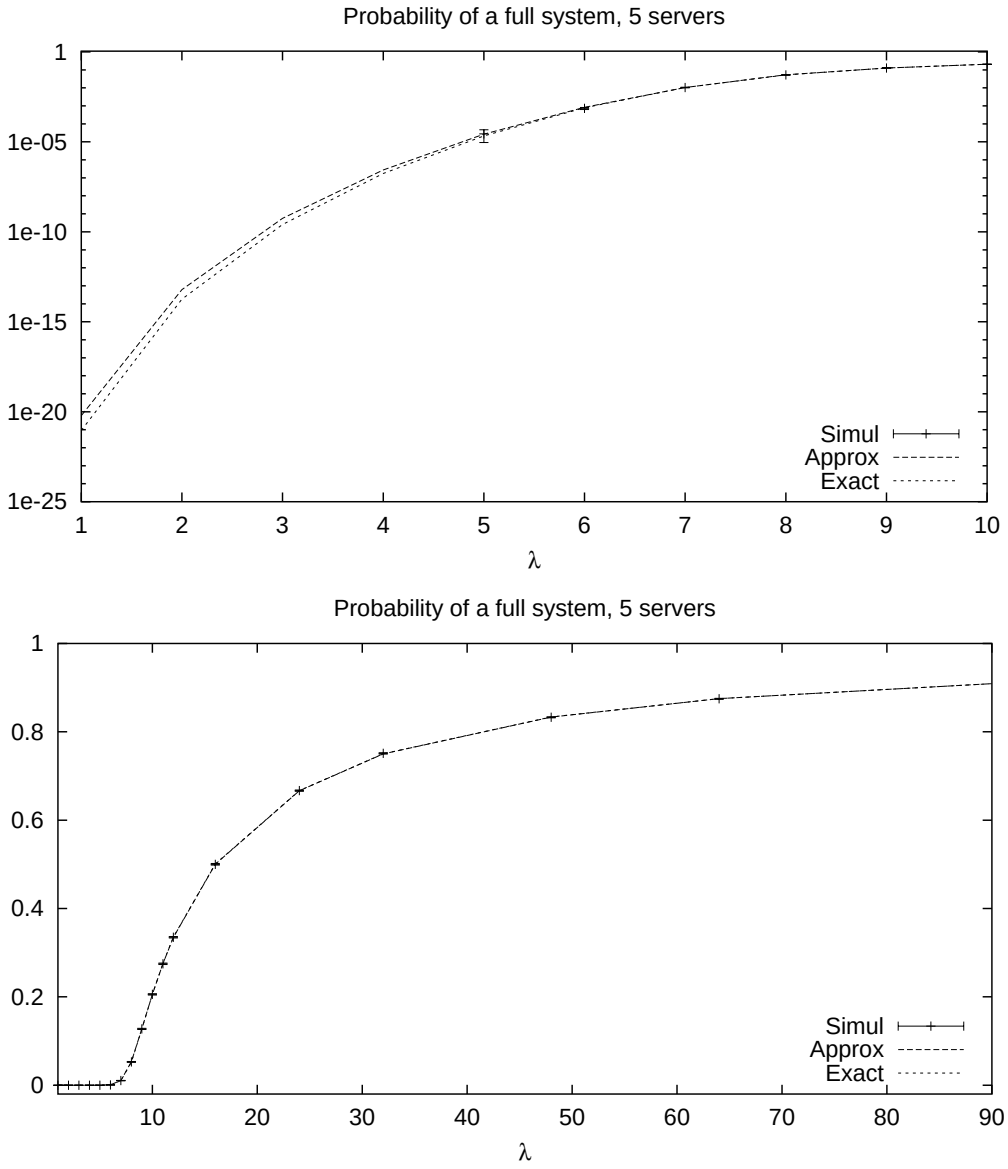


Figure 9.5: Probability of a full system, $K = 5$ servers and $J = 5$ jobs

the average number of requests in the system versus λ, as computed using exact analysis, approximate analysis, and discrete-event simulation. The top plot of Figure 9.5 shows the probability of a full system when the arrival rate is quite low; in these cases, the probability is quite small. Note that the approximation tends to overestimate extremely small

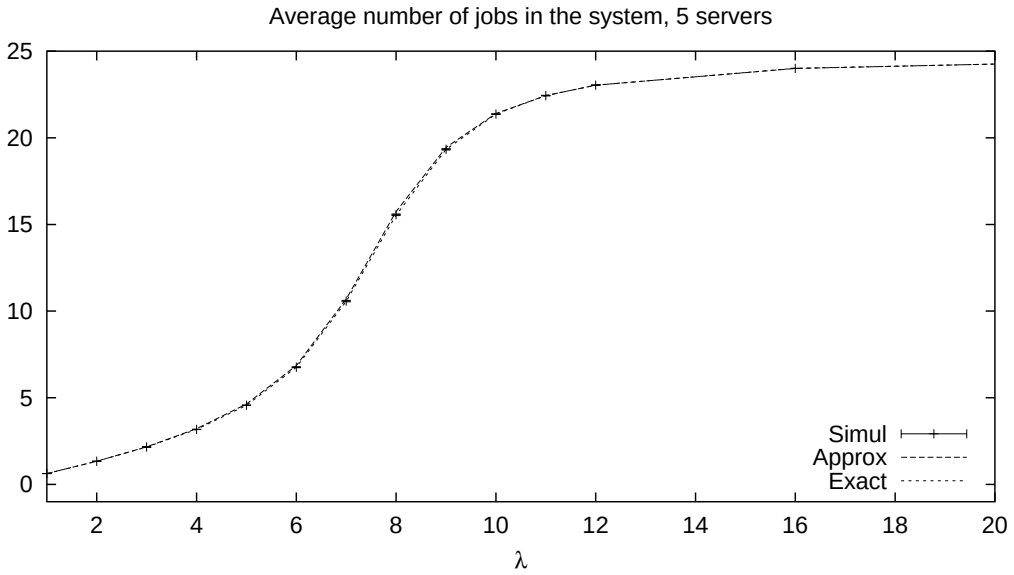


Figure 9.6: Average number of requests in the system, $K = 5$ servers and $J = 5$ jobs

probabilities. Also, note that the probability computed with our discrete-event simulation is zero for $\lambda < 5$, which accounts for the missing points in the plot. The 95% confidence intervals are shown in the plots, but for most cases the interval is too small to be observed. In the bottom plot, as with the plot in Figure 9.6, the results obtained by all three methods coincide nearly perfectly.

The CPU times required to compute these measures are shown in Figure 9.7. Note that the solution times vary slightly with the value of λ. As we expect, the CPU time required for approximation is significantly less than that for exact analysis.

Next, we examine a larger system, with $K = 10$ servers that can each handle $J = 10$ jobs. The rates for this case are shown in Table 9.4, and are similar to the previous case. The rate λ is a parameter of the model, as before.

The CTMC for this model has $1,322,808,654,651$ states, so we cannot solve it exactly.

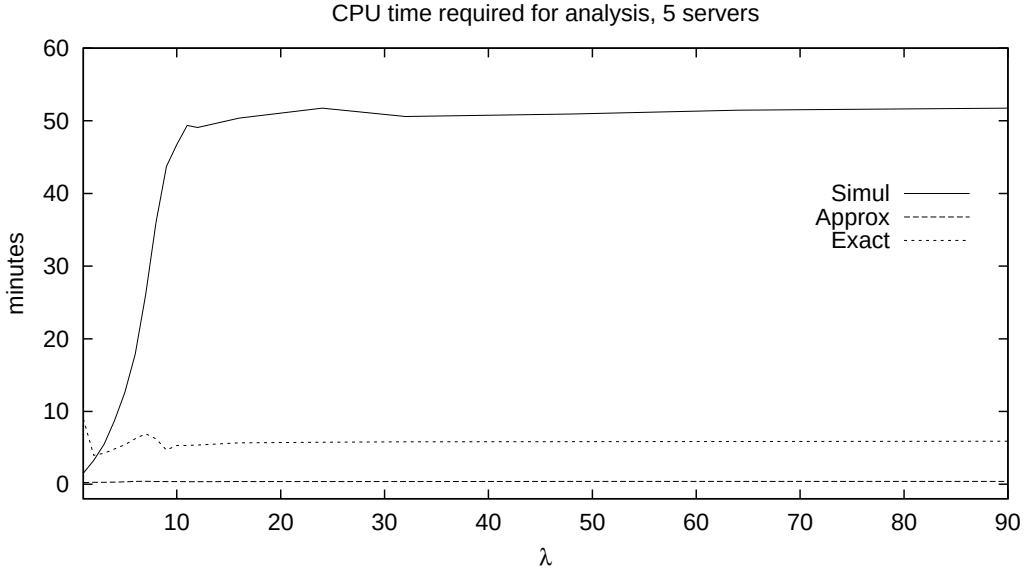


Figure 9.7: CPU times for $K = 5$ servers and $J = 5$ jobs

$\lambda_1 = 11$	$\lambda_6 = 10.5$	$\mu_1 = 2$	$\mu_6 = 1.5$
$\lambda_2 = 10.9$	$\lambda_7 = 10.4$	$\mu_2 = 1.9$	$\mu_7 = 1.4$
$\lambda_3 = 10.8$	$\lambda_8 = 10.3$	$\mu_3 = 1.8$	$\mu_8 = 1.3$
$\lambda_4 = 10.7$	$\lambda_9 = 10.2$	$\mu_4 = 1.7$	$\mu_9 = 1.2$
$\lambda_5 = 10.6$	$\lambda_{10} = 10.1$	$\mu_5 = 1.6$	$\mu_{10} = 1.1$

Table 9.4: Rates for the model of Figure 9.4 with 10 servers

We must turn to our approximation or to discrete-event simulation. For each of these techniques, we show the probability that the system is full in Figure 9.8, and the average number of requests in the system in Figure 9.9. As with the previous model, we show the probability of a full system for low arrival rates λ in the top plot of Figure 9.8. Again, our discrete-event simulation computes zero for this value when λ becomes too small (in this case $\lambda < 10$). Except for this extreme case, the values computed with our approximation and with discrete-event simulation match quite well. The CPU times required to compute

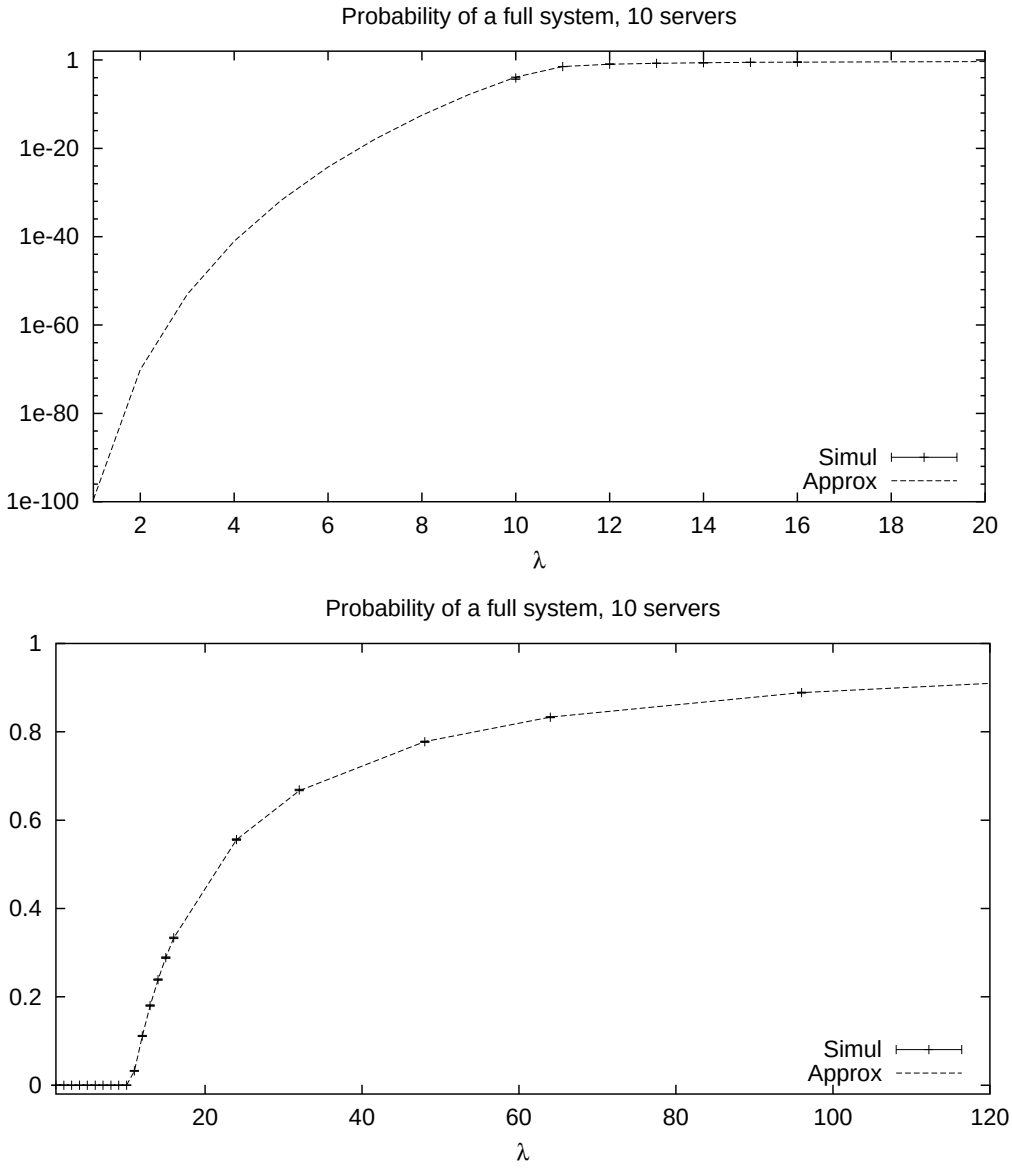


Figure 9.8: Probability of a full system, $K = 10$ servers and $J = 10$ jobs

the measures for this model are shown Figure 9.10.

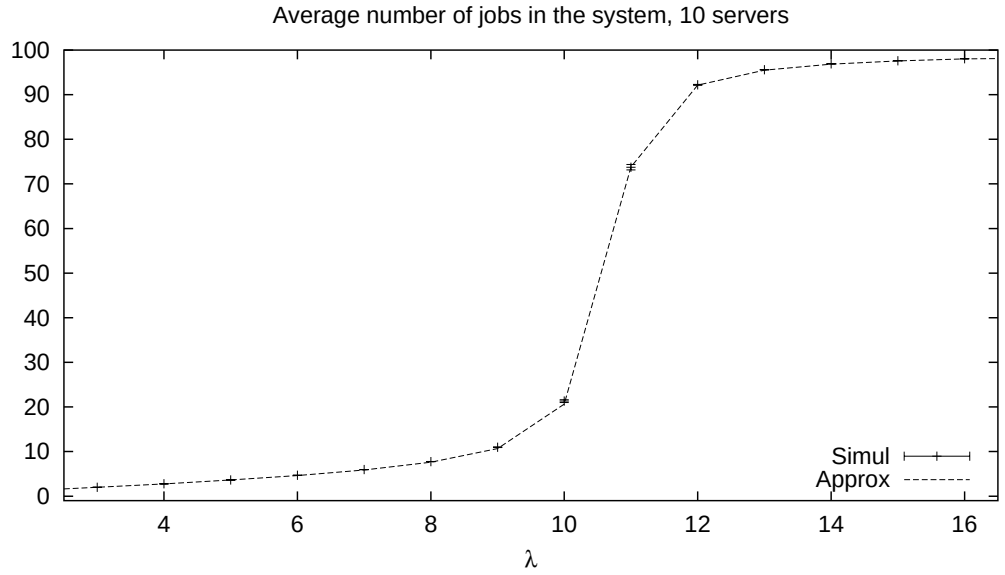


Figure 9.9: Average number of requests in the system, $K = 10$ servers and $J = 10$ jobs

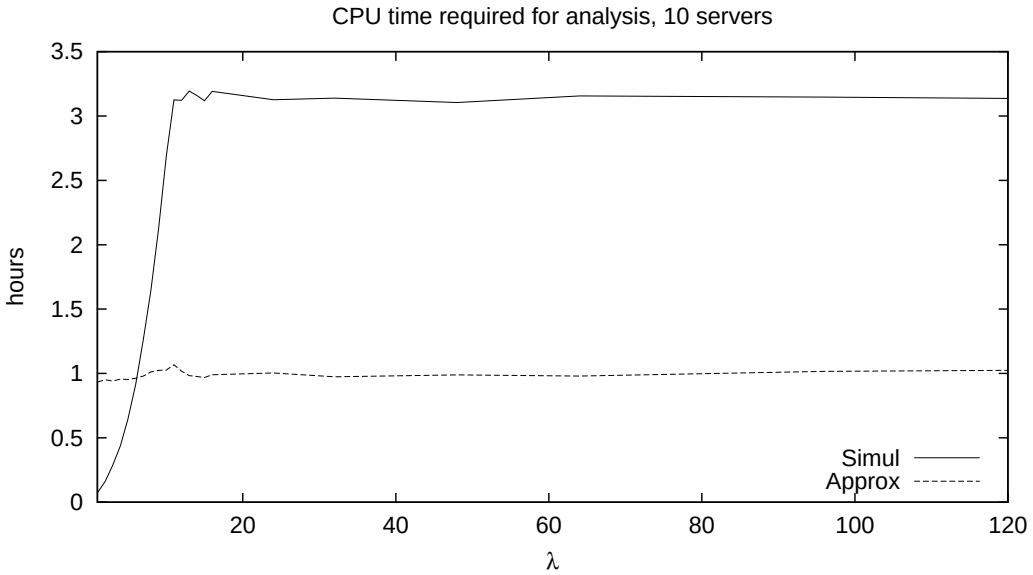


Figure 9.10: CPU times for $K = 10$ servers and $J = 10$ jobs

Chapter 10

Conclusion

High-level modeling formalisms, such as stochastic Petri nets, are gaining increased attention as tools for analyzing complex systems. An exact mathematical analysis of such a model can easily require excessive amounts of storage and CPU time. This is due to three large structures that must be computed: the set of reachable states of the model, the matrix of transition rates between states, and the vector of state probabilities. In this thesis, we described new techniques for computing and representing each of these structures. Our contributions extend the size of problems that can be reasonably attempted over previous state-of-the-art techniques, in some cases by several orders of magnitude.

We first developed a multi-level data structure for storing the set of reachable states $\mathcal{S}$. The only requirement for using our structure is that our model must be decomposable into submodels. We introduced an important concept: the "locality" of an event. This idea allows us to use properties of events that can easily be determined from the model to reduce computation by recognizing portions of the data structure that remain unchanged. We demonstrated experimentally that modifying generation algorithms to consider event locality can reduce total execution times by as much as 20%, at essentially no cost in terms of memory.

We then presented a new symbolic algorithm for generating $\mathcal{S}$, in which large sets of states can be manipulated efficiently using our specialized operators. Our symbolic algorithm can be applied to a wide class of structured models; namely, any structured model whose submodels are structurally "well-behaved". We demonstrated experimentally that our algorithm performs better than other symbolic algorithms in terms of both storage and CPU requirements. Experiments have shown that the efficiency of our algorithm is sensitive to the choice of model decomposition and ordering of submodels.

Kronecker algebra can be used as a memory-efficient representation for the transition rate matrix $\hat{\mathbf{R}}$ when the model is composed of submodels whose event rates are somewhat independent. Previous state-of-the-art solution techniques based on Kronecker representations suffered from several sources of computational overheads, including binary searches in data structures for $\mathcal{S}$, additional computation costs due to "spurious" entries in $\hat{\mathbf{R}}$, and the inability to exploit common products when performing multiplications. We presented an efficient data structure for $\mathcal{S}$ which does not require binary searches, but instead incurs a fixed computational cost proportional to the number of submodels. We also demonstrated how a new data structure we call *matrix diagrams* can be used to eliminate the computational cost due to "spurious" entries by representing $\mathbf{R}$, and can exploit common products by using a cache. Experimentally, we found that our technique requires significantly less CPU time as compared to previous state-of-the-art techniques, especially as the problem size increases.

Finally, we developed a technique for approximating the stationary probability vector $\boldsymbol{\pi}$ that can be used for models meeting the same restrictions as required for Kronecker representations. The uniqueness of our technique lies in its use of exact knowledge of the

set $\mathcal{S}$ and the matrix $\mathbf{R}$, which are stored in their entirety, albeit implicitly, thanks to our earlier results. We showed that our approximation gives exact results when the model of study possesses a product-form solution. We studied the effect of different types of model constructs on the accuracy of the approximation, and demonstrated experimentally that the technique can give results that are quite accurate.

10.1 Future research

The work we have presented in this thesis leads to many directions for future work. Some of these directions are simply extensions of our work to more general classes of models. Other directions can be viewed as related work, in the sense that they involve developing techniques similar to those we have presented here. Our contributions in this thesis also make new directions of research possible.

10.1.1 Extensions

The algorithms we describe in this thesis work when the occurrence of an event requires a random amount of time that is exponentially distributed. Our symbolic generation algorithm for $\mathcal{S}$, while it does not specifically require exponential distributions, requires that when more than one event is enabled in a given state, it is possible for any event to occur. One possible extension is to allow for immediate events (i.e., events that require zero time to occur). It is trivial to handle immediate *local* events (since they can be removed at the submodel level), so the only difficulty is due to immediate *synchronizing* events. In terms of symbolic generation, the current algorithm cannot be used, since it is impossible for a timed event to occur in a state in which an immediate event is enabled. It should be possible to

modify our symbolic generation algorithm to take immediate events into account by distinguishing between *tangible* states, in which only timed events are enabled, and *vanishing* states, in which an immediate event is enabled. It has already been shown how a Kronecker representation can be used with synchronizing events [29]. It should then also be possible to use a matrix diagram representation and to modify our approximation to handle immediate synchronizing events.

The next logical step after allowing immediate events is to allow events with continuous phase-type distributions or discrete phase-type distributions. In the case where the events of the model all have discrete (continuous) phase-type distributions, the underlying stochastic process is a discrete-time (continuous-time) Markov chain. In the continuous case, we must modify our symbolic generation algorithm for $\mathcal{S}$ to keep track of the current "phase" of each event. In the discrete case, it is possible (and in fact likely) that two events occur at the same time, while this happens with probability zero in the continuous case. A symbolic generation algorithm for $\mathcal{S}$ that correctly handles discrete-time events will thus be substantially different from the continuous case.

10.1.2 Related work

We have shown how the set of reachable states $\mathcal{S}$ can be generated symbolically. This allows us to determine if states satisfying a certain condition are reachable from the initial state. However, more general types of analysis are sometimes desired, such as if the system can reach one state from another state along a path of states which all satisfy a given condition. In particular, reachability *graph* manipulation can be used for other types of analysis. For instance, reachability graph analysis can determine the recurrent classes of the underlying

CTMC. Analysis of the reachability graph can be done conceptually by manipulating a matrix similar to the transition rate matrix $\mathbf{R}$, where the non-zero elements all have the value 1. This boolean matrix can be represented either by using MDDs or by using matrix diagrams. Operations that are required of the boolean matrix must be implemented in terms of the MDD or matrix diagram. We intend to investigate types of analyses that can be performed by combining reachability set and reachability graph manipulations.

We presented a technique for approximating the stationary probability vector $\boldsymbol{\pi}$. We should then investigate whether it is possible to modify our technique to approximate transient probabilities or the mean time spent in each state before absorption. Another important extension would be to develop a technique for computing bounds on the error of the approximation. As our approximation is based on a single simplifying assumption, determining the accuracy of our results should be equivalent to determining how closely our assumption holds. However, this is usually a very difficult problem.

10.1.3 New directions

Our symbolic generation algorithm for $\mathcal{S}$ using MDDs is ultimately limited by large CPU times and storage requirements. For manipulation of extremely large MDDs, *distributed* MDD manipulation algorithms are desirable. In this case, each MDD node is handled by one of N workstations. The primary difficulty with distributed MDD manipulations is the dynamic nature of MDDs: nodes are created and destroyed many times during a computation, and the lifetime of a node may be quite short. However, the potential for CPU speedup and analysis of much larger systems by combining the available memory of several workstations makes this a promising direction of research.

Another area in which distributed algorithms have great potential is that of computing $\boldsymbol{\pi}$. An efficient distributed algorithm would allow us to compute larger vectors $\boldsymbol{\pi}$ due to the combination of available memory, and to compute vectors $\boldsymbol{\pi}$ faster due to the combination of CPU power. Using our sophisticated data structures discussed in this thesis, it is now possible for every workstation to have an exact representation of $\mathcal{S}$ and $\mathbf{R}$, and a portion of the probability vector $\boldsymbol{\pi}$. By using a block iterative method to compute $\boldsymbol{\pi}$, we can reduce the communication costs of distributed numerical solution, since only the outer iterations require that messages be sent. The number of outer iterations will depend on the choice of partition for the probability vector $\boldsymbol{\pi}$. Exact knowledge of $\mathcal{S}$ and $\mathbf{R}$ should enable us to partition $\boldsymbol{\pi}$ effectively.

Finally, we recall that each of our techniques requires that the model be decomposed into submodels. For some techniques, most notably symbolic generation of $\mathcal{S}$, the choice of model decomposition and the ordering of the submodels has a dramatic effect on the efficiency of the approach. Currently, the choice of model decomposition rests entirely on the modeler. An interesting area of research is that of heuristics for automatically determining model decompositions for which our symbolic generation algorithm works well. Such a heuristic might use invariants of the model to determine the submodels and structural properties of the model to determine the ordering of the submodels.

Appendix A

SMART

This chapter discusses SMART (Simulation and Markovian Analyzer for Reliability and Timing) [26], a computer package to study the performance, reliability, and behavior of complex systems. SMART integrates various high-level stochastic modeling formalisms (such as stochastic Petri nets) and various solution techniques into a single modeling study. Since SMART is intended as a research tool, it is written in a modular way that allows researchers to integrate new formalisms and solution algorithms.

SMART was developed from scratch starting in fall of 1994. Gianfranco Ciardo and I worked out the initial design, and I wrote the first 40,000 or so lines of C++ code. Since 1998, four additional students have joined the SMART project. Currently, SMART consists of over 80,000 lines of C++ code. In its final version, the following features will be present.

- Both numerical solution and discrete-event simulation algorithms will be available to the user for the solution of models.
- Numerical solution algorithms will be available for discrete- and continuous-time Markov chains, and to some classes of stochastic processes obtained by combining both discrete and continuous phase-type distributions in the same model.

- A unifying language will be used to define the structure and measures of models, regardless of the formalism used to express them (queueing networks, Petri nets, etc.) and the type of study required.
- A mechanism for fixed-point iterative techniques can be used for the decomposition and solution of complex models.
- A mechanism will be available for distributing computations over a network of workstations.

A.1 SMART Language

Models are described to SMART using a strongly-typed, declarative language. The four basic predefined *types* for the objects defined in SMART are:

`bool`**:** `true` or `false`.

`int`**:** integer values (machine-dependent range).

`real`**:** floating-point values (machine-dependent range and precision).

`string`**:** character-array values, used for file names, print statements, etc.

Composite types can be defined using the concepts of:

sets**:** collections of homogeneous objects.

arrays**:** multidimensional data structures of homogeneous objects indexed by set elements.

aggregates**:** collections of heterogeneous objects, analogous to the Pascal "record".

A type can be further modified by the following *natures*, which describe stochastic characteristics:

`const:` (the default) a non-stochastic quantity.

`ph:` a random variable with discrete or continuous phase-type distribution.

`rand:` a random variable with arbitrary distribution.

A.1.1 Function declarations

Declaration of new functions with parameters is allowed. Functions can be recursive:

```
int fact(int n) := cond(n==0,1,n*fact(n-1));
```

Function calls can pass parameters by position or by name:

```
int raiseto(int base, int exp) :=
  cond(exp==1,base,base*raiseto(base,exp-1));
raiseto(2,16);
raiseto(exp:=16, base:=2); /* same effect */
```

Parameters can be assigned default values that are used if the passed parameter is the keyword `default` or if not all of the named parameters are specified:

```
real raiseto(real base, int exp:=2) :=
  cond(exp==1,base,base*raiseto(base,exp-1));
raiseto(2.0, default);
raiseto(base:=2.0, exp:=default);
raiseto(base:=2.0);           /* 4.0 */
```

```
raiseto(base:=2.0, exp:=3);   /* 8.0 */
```

The previous example also shows how functions can be overloaded, as long as the types of their formal parameters can be used to distinguish between them.

A.1.2 Arrays

Arrays are functions declared within a `for` statement. The dimensionality of the array is determined by the enclosing `for` iterators. As the indices of the array along each dimension belong to a finite set, it is legal to define arrays with `real` indices. For example,

```
for (int s in {1..5}, real f in {1..s..0.1}) {
  real res[s][f]:= MyModel(s,f).out1;
}
```

stores the results of a parametric study of `MyModel` in array `res`, where the output measure `out1` is computed when the first input parameter ranges from one to five and the second one ranges from one to the value of the first parameter, with a step of one-tenth.

A.1.3 Fixed-point iterations

The approximate solution of a model is often based on a heuristic decomposition [30], where (sub)models are solved in a fixed-point iteration. This can be specified with the `converge` statement:

```
converge {
  real x guess 1.0;
  real y := fy(x, y);
```

```
    real x := fx(x, y);
  }
```

The iterations stop when two subsequent `x` values (and `y` values) differ by less than ϵ in either relative or absolute terms. The values for `x` and `y` are updated either immediately or at the end of each iteration. Both ϵ and the updating criterion are fine-tuned using *option* statements.

The `converge` and `for` statements can be nested arbitrarily within each other.

A.2 Random variables

SMART can manipulate discrete and continuous phase-type distributions, which correspond to the `ph int` and `ph real` types. An operation on a `ph` type produces another `ph` type if phase-type distributions are closed under that operation; otherwise, the result is a `rand` type:

```
ph int X := geometric(0.7);
ph int Y := discreteuniform(1,5);
ph int sumXY := X+Y;
ph int prodX := 4*X;
ph int chooseXY := choose(0.4:X, 0.6:Y);
ph int minXY := min(X, Y);
```

However, mixing `ph int` and `ph real`, or performing other operations not guaranteed to result in a phase-type distribution, forces SMART to consider the random variable as generally distributed:

```
rand int diffXY := X-Y;

rand real sumRX := 4.5+X;

rand int prodXY := X*Y;
```

A `rand` random variable can then be manipulated only via discrete-event simulation (under development).

A.3 Model formalisms

The declaration of a model is similar to that of a function. Instead of a return value, a model declaration specifies a block that defines the model. A model definition consists of three parts: declarations, specification, and user measures. Components of the model, such as the places and transitions of a Petri net, are defined in the declaration section. The model is then specified by calling formalism-specific functions. In the case of a Petri net, these are functions used to specify the structure of the net, the firing times of the transitions, the initial marking, and so on. Measures are declared as user-defined functions that compute some quantity of interest, such as the expected number of tokens in a given place. Measures are the only part of a model that can be accessed outside of the model definition block. Algorithm B.1 shows an example of a model defined in SMART.

The design of SMART allows for relatively easy addition of new model formalisms. Currently, the following formalisms are implemented:

`dtmc, ctmc:` discrete or continuous time Markov chains.

`spn:` stochastic Petri nets.

`sml`(under development): stochastic modeling language (models software tasking systems using a pseudo-code syntax).

For the last two formalisms, the type of the underlying stochastic process is determined by the distributions used for the durations of the events being modeled.

A.4 Distributed version (under development)

SMART can distribute work on a network of workstations in two ways: (i) large models can be solved using distributed algorithms, and (ii) different models, or the same model with different parameters, can be solved simultaneously on different machines.

A.4.1 Distributed algorithms

Following the ideas in [25, 76], the state space of any discrete-state model can be explored using a distributed algorithm. If N workstations are available, the resulting state space is partitioned in N classes, one per workstation. We are currently implementing the distributed numerical solution of the process, assuming it is a Markov chain. This approach provides the ability of solving large models by exploiting the overall available memory.

A.4.2 Concurrent solutions

In many cases, the same model needs to be solved for a large number of parameter combinations. These can be easily specified using the array feature in SMART, which is then able to maintain a pool of jobs to be sent to remote hosts. SMART builds a dependency graph, so that solutions that depend on previous results wait until those results have been computed.

This is by far the most efficient use of multiprocessing, provided each solution is small enough to fit in the main memory of the processor to which it is assigned. The amount of communication is limited to a scheduler sending solution requests to the remote hosts, and receiving back the requested measures.

Appendix B

Benchmarks

This chapter contains a description of the models used in our experiments.

B.1 Kanban model

Figure B.1 depicts the Petri net of a Kanban system [27, 28, 29, 68, 69] composed of four stations; indices in the place and transition names identify the station. A part enters a station only if a Kanban "ticket" is available (places p_{kan}). The part is then processed. If the work is performed correctly (transitions t_{ok}) the part can move to the next station, otherwise the part must be fixed and sent back (transitions t_{back}) to be processed again. A part that passes station 1 forks into two parts, one for station 2 and one for station 3; when the two parts pass their stations they then join and move to station 4. The initial state of the model is the absence of parts in each station and N tickets at each station. All transitions have exponentially-distributed delays with rates given in Table B.1. The SMART code for a Petri net of the Kanban model is shown in Algorithm B.1. The measure of interest for each station k is the expected number of parts e_k within the station. We compute this for each station by summing the expected number of tokens in places p_m,

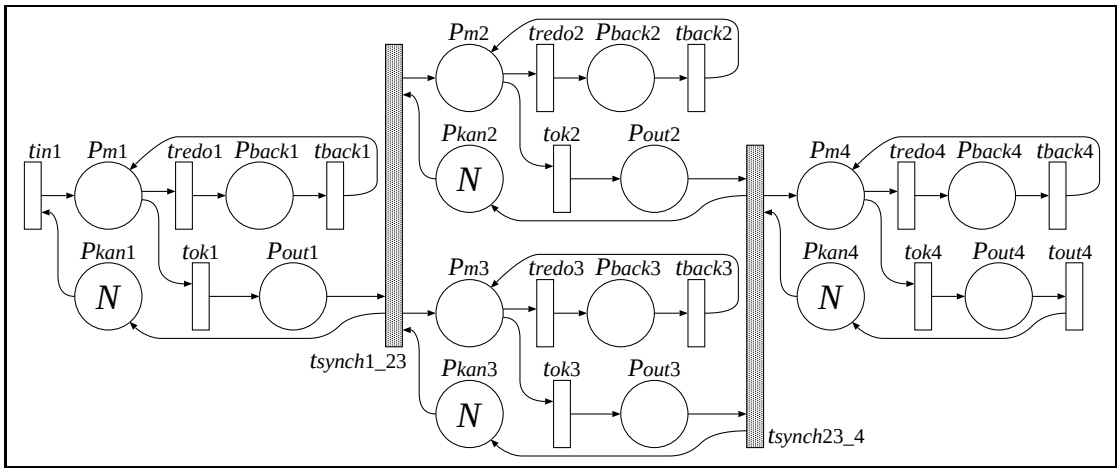


Figure B.1: Petri net of the Kanban model

t	$Rate^t$	t	$Rate^t$	t	$Rate^t$	t	$Rate^t$
t_{in1}	1.0	t_{redo1}	0.36	t_{ok1}	0.84	t_{back1}	0.3
t_{synch1_23}	0.4	t_{redo2}	0.42	t_{ok2}	0.98	t_{back2}	0.3
$t_{synch23\_4}$	0.5	t_{redo3}	0.39	t_{ok3}	0.91	t_{back3}	0.3
t_{out4}	0.9	t_{redo4}	0.33	t_{ok4}	0.77	t_{back4}	0.3

Table B.1: Transition rates for the Kanban model

p_{back}, and p_{out}. Note that the Kanban model is a *Kronecker-product-form* model if the finest possible decomposition is used (i.e., one place per submodel).

B.2 Flexible manufacturing system (FMS) model

Figure B.2 depicts the Petri net of a flexible manufacturing system [21, 25, 27, 30, 49, 68, 69, 107]. The system consists of three types of machines: M_1, M_2, and M_3. There are three machines of type M_1, which process parts of type P_1. Machine M_2 processes parts of type P_2, one at a time. When M_2 is idle, it can process parts of type P_3. Two machines of type M_3 can assemble finished parts of type P_1 and P_2 into a single part. When a part of type

```
spn kanban(int n) := {
  place pm1, pback1, pkan1, pout1, pm2, pback2, pkan2, pout2,
        pm3, pback3, pkan3, pout3, pm4, pback4, pkan4, pout4;

  trans tin1, tredo1, tok1, tback1, tsynch1_23, tredo2, tok2, tback2,
        tsynch23_4, tredo3, tok3, tback3, tredo4, tok4, tback4, tout4;

  firing(tin1:expo(1.0), tredo1:expo(0.36), tok1:expo(0.84),
         tback1:expo(0.3), tsynch1_23:expo(0.4), tredo2:expo(0.42),
         tok2:expo(0.98), tback2:expo(0.3), tredo3:expo(0.39),
         tsynch23_4:expo(0.5), tok3:expo(0.91), tback3:expo(0.3),
         tredo4:expo(0.33), tok4:expo(0.77), tback4:expo(0.3),
         tout4:expo(0.9) );

  arcs(pkan1:tin1, tin1:pm1, pm1:tredo1, pm1:tok1, tredo1:pback1,
       tok1:pout1, pback1:tback1, tback1:pm1, pout1:tsynch1_23,
       tsynch1_23:pkan1,
       pkan2:tsynch1_23, tsynch1_23:pm2, pm2:tredo2, pm2:tok2,
       tredo2:pback2, tok2:pout2, pback2:tback2, tback2:pm2,
       pout2:tsynch23_4, tsynch23_4:pkan2,
       pkan3:tsynch1_23, tsynch1_23:pm3, pm3:tredo3, pm3:tok3,
       tredo3:pback3, tok3:pout3, pback3:tback3, tback3:pm3,
       pout3:tsynch23_4, tsynch23_4:pkan3,
       pkan4:tsynch23_4, tsynch23_4:pm4, pm4:tredo4, pm4:tok4,
       tredo4:pback4, tok4:pout4, pback4:tback4, tback4:pm4,
       pout4:tout4, tout4:pkan4 );

  init(pkan1:n, pkan2:n, pkan3:n, pkan4:n);

  real e1 := ssavg( tk(pm1)+tk(pback1)+tk(pout1) );
  real e2 := ssavg( tk(pm2)+tk(pback2)+tk(pout2) );
  real e3 := ssavg( tk(pm3)+tk(pback3)+tk(pout3) );
  real e4 := ssavg( tk(pm4)+tk(pback4)+tk(pout4) );
};
```

Algorithm B.1: SMART code for the Kanban model

P_1 or P_2 is finished, it may either join with its counterpart on a M_3 machine (transitions t_{P1j} and t_{P2j}) or exit (transitions t_{P1e} and t_{P2e}). Completed parts of all types are shipped and raw parts of the same type enter the system (transitions $t_{P\bullet s}$) to maintain a constant

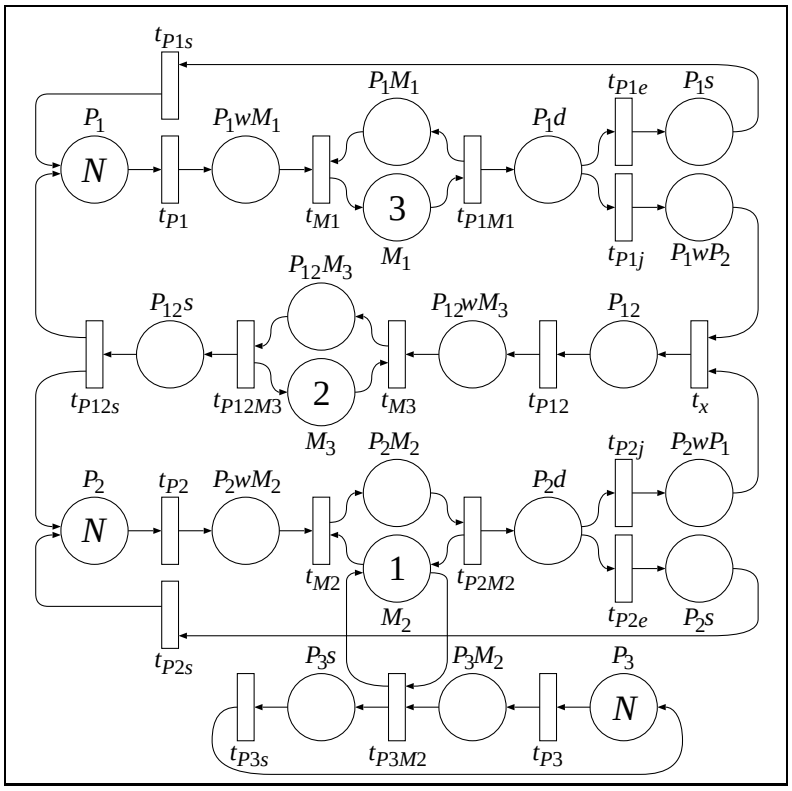


Figure B.2: Petri net of the Flexible Manufacturing System model

inventory of parts. Initially, there are N parts each type P_1, P_2 and P_3 in the system.

Raw parts are moved to machines using pallets, one at a time. A pallet is also required to move a P_1 and P_2 pair to one of the M_3 machines. There are $\lfloor \frac{3N}{2} \rfloor$ pallets available to move parts, and these are shared by all machines. This is modeled by marking-dependent rates on the transitions that signify the movement of parts on pallets (transitions $t_{P1}, t_{P2}, t_{P3}, t_{P12}$). The rates of transitions are shown in Table B.2. Note that some of the rates have been changed from the original version of the model in [21]. With these changes, a decomposition of a single place per submodel is a *Logical-product-form* model.

To obtain a *Kronecker-product-form* decomposition, we must eliminate the global rate

t	$Rate^t$
t_{P1}	$\#P_1 \cdot \min\left\{1, \left\lfloor \frac{3N}{2} \right\rfloor \cdot \frac{1}{\#P_1 + \#P_2 + \#P_3 + \#P_{12}}\right\}$
t_{P2}	$\#P_2 \cdot \min\left\{1, \left\lfloor \frac{3N}{2} \right\rfloor \cdot \frac{1}{\#P_1 + \#P_2 + \#P_3 + \#P_{12}}\right\}$
t_{P12}	$\#P_{12} \cdot \min\left\{1, \left\lfloor \frac{3N}{2} \right\rfloor \cdot \frac{1}{\#P_1 + \#P_2 + \#P_3 + \#P_{12}}\right\}$
t_{P3}	$\#P_3 \cdot \min\left\{1, \left\lfloor \frac{3N}{2} \right\rfloor \cdot \frac{1}{\#P_1 + \#P_2 + \#P_3 + \#P_{12}}\right\}$

t	$Rate^t$	t	$Rate^t$	t	$Rate^t$	t	$Rate^t$
t_{P1M1}	$\frac{\#P_1M_1}{4}$	t_{P1s}	$\frac{\#P_1 s^\dagger}{60}$	t_{P1e}	$80^\ddagger$	t_x	$100^\ddagger$
t_{P2M2}	$\frac{1}{6}$	t_{P2s}	$\frac{\#P_2 s^\dagger}{60}$	t_{P1j}	$20^\ddagger$	t_{M1}	$100^\ddagger$
t_{P12M3}	$\#P_{12}M_3$	t_{P12s}	$\frac{\#P_{12} s^\dagger}{60}$	t_{P2e}	$60^\ddagger$	t_{M2}	$100^\ddagger$
t_{P3M2}	$\frac{1}{2}$	t_{P3s}	$\frac{\#P_3 s^\dagger}{60}$	t_{P2j}	$40^\ddagger$	t_{M3}	$100^\ddagger$

† These were originally single-server transitions with flushing arcs.
‡ These were originally immediate transitions.

Table B.2: Transition rates for the Flexible Manufacturing System model

dependency of transitions t_{P1}, t_{P2}, t_{P3}, and t_{P12}. This is done by introducing an additional "redundant" place P_r such that

$$\#P_r = \#P_1 + \#P_2 + \#P_3 + \#P_{12} \tag{B.1}$$

for all reachable markings. To do so, we place an arc from P_r to any transition that has an input arc from one of these places, and we place an arc to P_r from any transition that has an output arc to one of these places. With these modifications, if Equation B.1 holds in the initial marking, then it will hold for all reachable markings. We can then rewrite the

global rate dependencies as the product of local rate dependencies. For instance, the rate of transition t_{P1} becomes the product of $\#P_1$ and $\min\{1, \lfloor \frac{3N}{2} \rfloor \cdot \frac{1}{\#P_r}\}$. Thus, whenever a *Kronecker-product-form* decomposition is required, we must add place P_r to the model.

The measures of interest for this model are the throughputs of transitions t_{P1}, t_{P2}, t_{P3} and t_{P12}, which signify the rate of production of each type of finished part. Additionally, we can compute the "productivity" of the FMS [30], which assigns a net gain to each type of finished part:

$$prod = 400thru(t_{P1}) + 600thru(t_{P2}) + 100thru(t_{P3}) + 1100thru(t_{P12}).$$

This measure is used to assess the amount of revenue generated by the manufacturing system.

B.3 Dining philosophers model

Unlike the previous models, the Petri net for the dining philosophers model [68, 78, 80] grows as the number of philosophers grows. Figure B.3 shows the portion of the Petri net corresponding to a single philosopher and right-hand fork. The place corresponding to the left-hand fork, represented by the dotted circle, is part another subnet; it is depicted to show how the subnets interact. To obtain a complete Petri net model for N dining philosophers, we "connect" N copies of the subnet from Figure B.3. For instance, the complete model for ten dining philosophers is shown in Figure B.4.

Our Petri net model captures the behavior of the classical dining philosophers paradigm. We have N philosophers in a circle around a table, with a fork between adjacent philosophers. Each philosopher is initially thinking, represented by a token in place *Idle*. Thinking

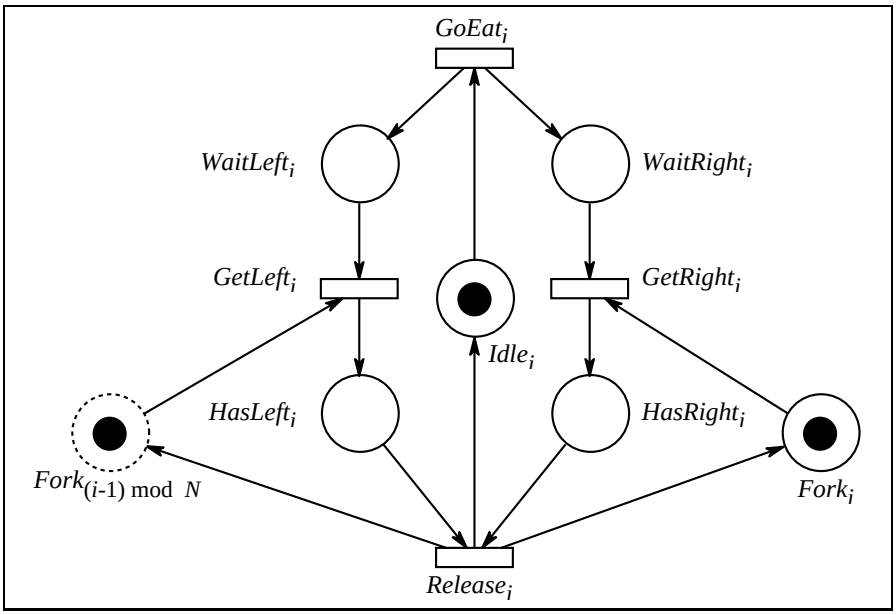


Figure B.3: The i^{th} philosopher subnet

philosophers eventually become hungry (transition *GoEat* fires). A hungry philosopher must obtain both a left-hand and a right-hand fork (transitions *GetLeft* and *GetRight*). When a fork is secured by a philosopher, it is not released until that philosopher has obtained both forks and has finished eating (transition *Release*). A full philosopher then releases both forks and continues thinking.

This model is used for logical analysis only. Note that the system contains two absorbing deadlock states, corresponding to every philosopher holding a left fork and every philosopher holding a right fork. The model is usually decomposed by grouping a single philosopher or multiple adjacent philosophers together into a submodel. It is also possible to consider a single place as a submodel. All of these are *Logical-product-form* decompositions.

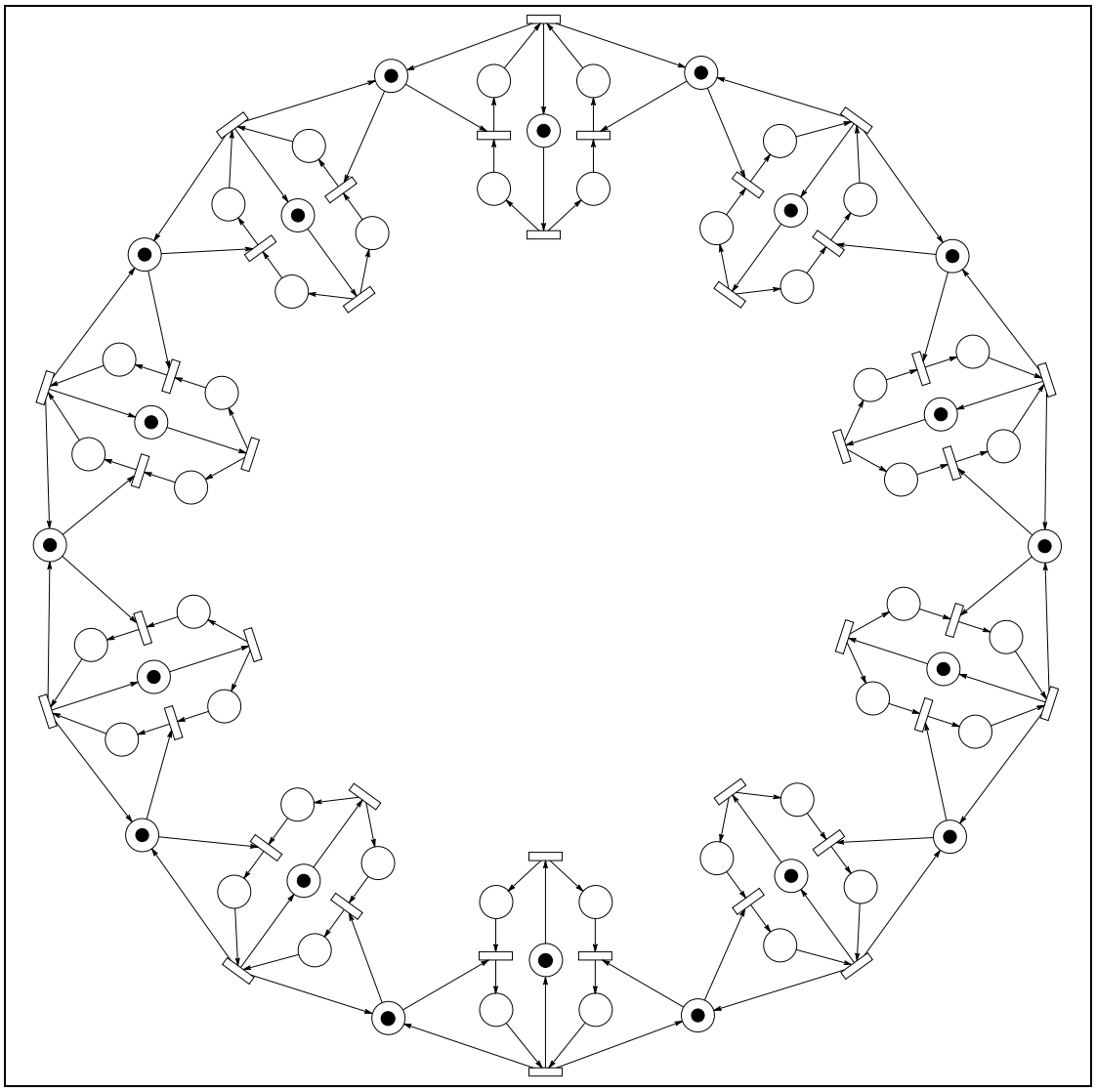


Figure B.4: The Petri net for ten dining philosophers

B.4 Slotted ring network protocol model

The slotted ring network protocol model [68, 78, 80] is similar to the dining philosophers model, as a subnet is used to describe the behavior of a single network node. Figure B.5 shows the subnet for the i^{th} network node. To study a network with N nodes, we use N copies of the subnet from Figure B.5, which interact via the shared transitions *Free* and

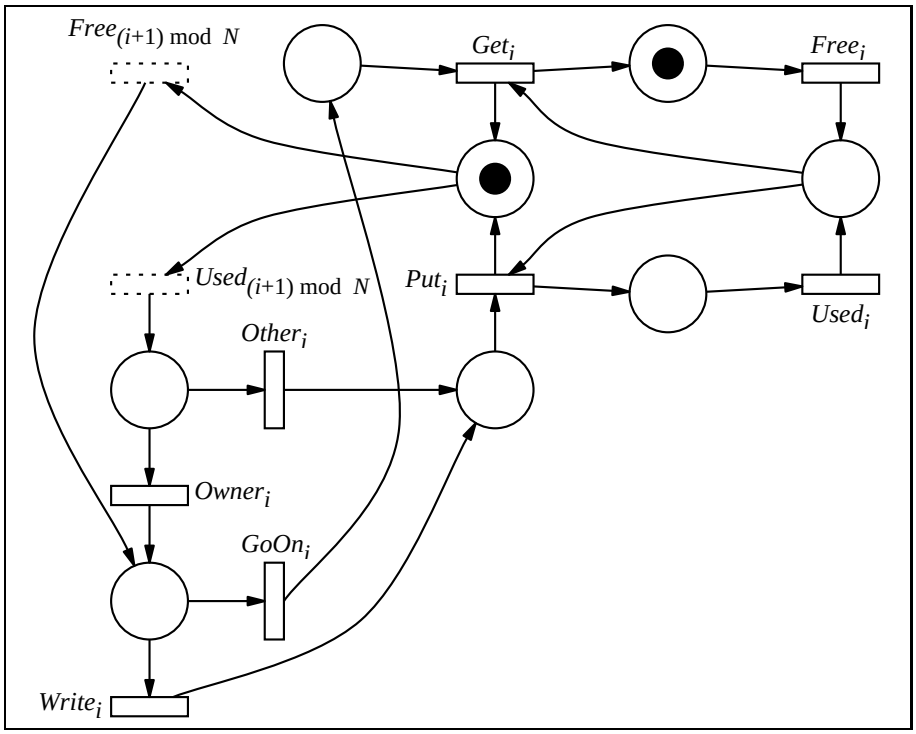


Figure B.5: The i^{th} network node for slotted ring

Used. As with the dining philosophers model, the slotted ring model is usually decomposed by grouping a single network node or multiple adjacent nodes together into a submodel. It is also possible to consider a single place as a submodel. All of these are *Logical-product-form* decompositions.

Appendix C

Analysis of Case

We analyze the worst-case complexity of a call to Case$(A^1, \ldots, A^M)$, where A^m is an MDD with K variables. We assume that a cache is used, so that a computational cost is incurred only for the unique calls to Case. We also assume that the costs of updating and searching the cache are constant, and the cost of a call to UniqueTableInsert (see Algorithm 6.1) is also constant. The MDD variable x_k can assume N_k possible values. We denote the number of nodes in MDD A as $|A|$, and we denote the number of level-k nodes in MDD A as $|A|_k$.

The total worst-case cost of a call to Case is given by the sum of the costs incurred at each level. That is, if C_k is the total number of (distinct) calls to Case when the top variable is x_k, then the total cost is at worst

$$W = \sum_{k=1}^{K} C_k \cdot N_k, \tag{C.1}$$

since the cost of a Case computation with top variable x_k is N_k. The value of C_k can be determined by counting the number of ways to call Case when the top variable is x_k:

$$C_k = \sum_{\mathbf{i} \in \mathcal{I}_k} \prod_{m=1}^{M} |A^m|_{\mathbf{i}[m]}, \tag{C.2}$$

where

$$\mathcal{I}_k = \{\mathbf{i} \in \{1, \ldots, k\}^M : \exists j \text{ such that } \mathbf{i}[j] = k\}.$$

In Equation C.2, $\mathbf{i}$ is an integer vector representing the top variables of each MDD (i.e., $x_{\mathbf{i}[m]}$ is the top variable of A^m). Since we know that the overall top variable is x_k, each element of $\mathbf{i}$ can assume values between 1 and k, and at least one element must be equal to k. The set of all such vectors is denoted $\mathcal{I}_k$.

Substituting Equation C.2 into Equation C.1 gives the exact worst-case complexity. This can be simplified if we rewrite Equation C.1 as

$$W \leq N \cdot \sum_{k=1}^{K} C_k, \tag{C.3}$$

where $N = \max(N_1, \ldots, N_K)$. Substituting for C_k gives us

$$\begin{aligned}
W &\leq N \cdot \sum_{k=1}^{K} \sum_{\mathbf{i} \in \mathcal{I}_k} \prod_{m=1}^{M} |A^m|_{\mathbf{i}[m]} && \text{(C.4)} \\
&\leq N \cdot \sum_{\mathbf{i} \in \{1,\ldots,K\}^M} \prod_{m=1}^{M} |A^m|_{\mathbf{i}[m]} && \text{(C.5)} \\
&\leq N \cdot \prod_{m=1}^{M} \sum_{k=1}^{K} |A^m|_k && \text{(C.6)} \\
&\leq N \cdot \prod_{m=1}^{M} |A^m|. && \text{(C.7)}
\end{aligned}$$

The step from Equation C.4 to Equation C.5 holds because $\bigcup_{k=1}^{K} \mathcal{I}_k = \{1, \ldots, K\}^M$ and $\mathcal{I}_k \cap \mathcal{I}_{k'} = \emptyset$ for distinct k, k'. The step from Equation C.5 to Equation C.6 is due to factoring, and the final step is because the sum of the number of nodes at each level gives the total number of nodes.

Appendix D

Dining philosophers states

We now study the reachable states $\mathcal{S}$ for the dining philosophers model. Our discussion will make use of the MDD encoding for $\mathcal{S}$ where each submodel consists of a philosopher and fork, as in Figure B.3. The MDD encoding for $\mathcal{S}$ in the case of 5 dining philosophers is shown in Figure D.1.

In local states $0, 1$ and 2, philosopher k is holding his left fork, which is also the right fork of philosopher $k-1$. Thus, downward pointers from these states always go to nodes of type A or C, in which philosopher $k-1$ cannot hold his right fork. In the other local states, philosopher k is not holding his left fork, so those downward pointers always go to nodes of type B or D, in which philosopher $k-1$ can hold his right fork.

Notice from the figure that the A and C nodes have similar patterns of downward pointers, and so do the B and D nodes, except at level 1. The shaded local states in the level-5 node correspond to the states in which philosopher 1 can obtain his left fork, which is the right fork of philosopher 5. This must be "remembered" until we reach level 1 of the MDD; thus, nodes A and B correspond to the case where philosopher 1 can obtain his left fork, and nodes C and D correspond to the case where philosopher 1 cannot obtain his left fork.

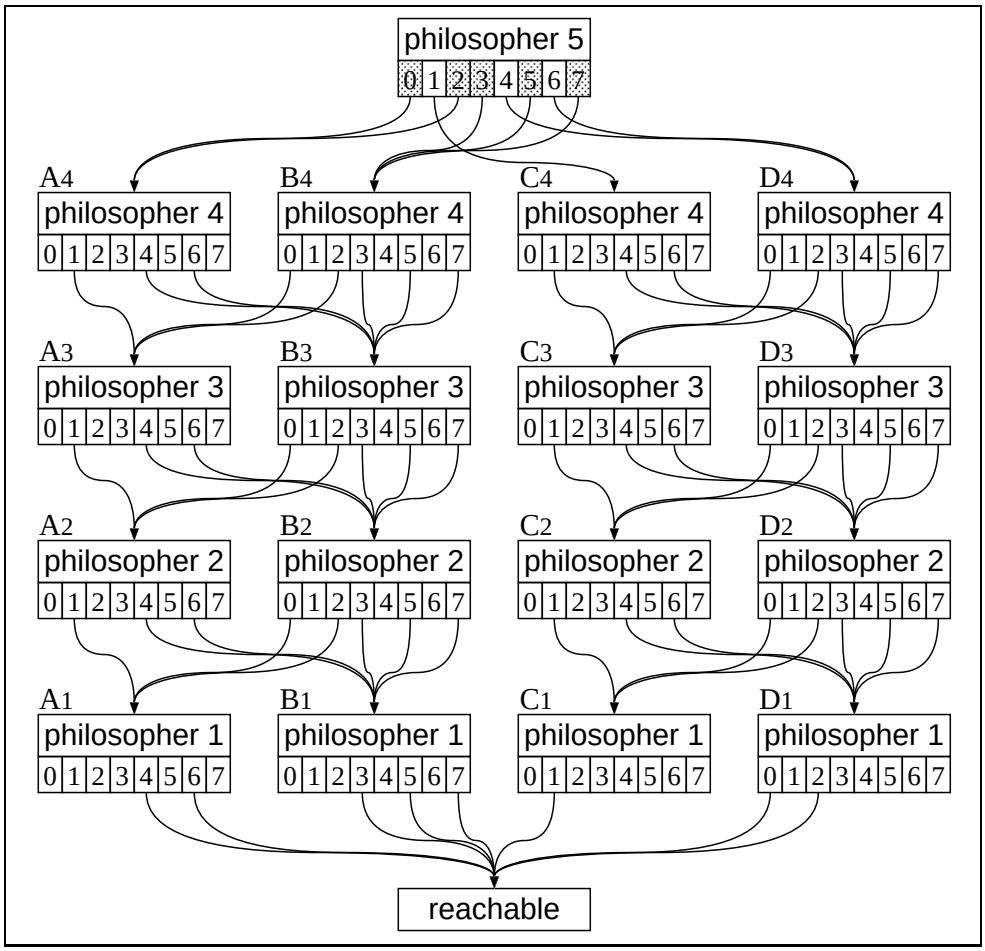


Figure D.1: MDD encoding for $\mathcal{S}$, 5 dining philosophers

It should be clear that additional philosophers will produce more "middle levels" of the MDD. That is, the MDD for $K > 2$ philosophers will have a level-K node identical to the level-5 node in the figure, level-1 nodes identical to the level-1 nodes of the figure, and the nodes at other levels k will be copies of the nodes at levels 2,3, and 4 of the figure.

The number of states encoded by the K-level MDD can be computed by the following

recurrences, which can be seen by studying Figure D.1.

$$\begin{aligned} |\mathcal{S}(K)| &= 2A_{K-1} + 3B_{K-1} + C_{K-1} + 2D_{K-1} \\ A_n &= A_{n-1} + 2B_{n-1} \\ B_n &= 2A_{n-1} + 3B_{n-1} \\ C_n &= C_{n-1} + 2D_{n-1} \\ D_n &= 2C_{n-1} + 3D_{n-1} \end{aligned} \tag{D.1}$$

Note that the recurrences for A and C are identical, as are the ones for B and D. We can rewrite the recurrences for A and B as

$$\begin{aligned} A_n &= Fib(t-1) \cdot A_{n-1} + Fib(t) \cdot B_{n-1} \\ B_n &= Fib(t) \cdot A_{n-1} + Fib(t+1) \cdot B_{n-1} \end{aligned}$$

where $Fib(n)$ is the n^{th} Fibonacci number [58] and $t = 3$. We can then expand the recurrence for A:

$$\begin{aligned} A_n &= Fib(t-1) \cdot (Fib(t-1) \cdot A_{n-2} + Fib(t) \cdot B_{n-2}) + \\ &\quad Fib(t) \cdot (Fib(t) \cdot A_{n-2} + Fib(t+1) \cdot B_{n-2}) \\ &= (Fib(t-1) \cdot Fib(t-1) + Fib(t) \cdot Fib(t))A_{n-2} + \\ &\quad (Fib(t-1) \cdot Fib(t) + Fib(t) \cdot Fib(t+1))B_{n-2} \\ &= Fib(2t-1) \cdot A_{n-2} + Fib(2t) \cdot B_{n-2} \end{aligned}$$

where the last step uses the property [58]

$$Fib(a+b) = Fib(a) \cdot Fib(b+1) + Fib(a-1) \cdot Fib(b).$$

Similarly, we can expand the recurrence for B to obtain

$$B_n = Fib(2t) \cdot A_{n-2} + Fib(2t+1) \cdot B_{n-2}.$$

This process can be repeated $n-1$ times, which gives us

$$\begin{aligned} A_n &= Fib((n-1)t-1)A_1 + Fib((n-1)t)B_1 \\ B_n &= Fib((n-1)t)A_1 + Fib((n-1)t+1)B_1. \end{aligned}$$

Substituting $t = 3$ into these equations, we see that the recurrences for A and B can be written as

$$\begin{aligned} A_n &= Fib(3n-4) \cdot A_1 + Fib(3n-3) \cdot B_1 \\ B_n &= Fib(3n-3) \cdot A_1 + Fib(3n-2) \cdot B_1 \end{aligned} \tag{D.2}$$

and the recurrences for C and D are similar. Looking at Figure D.1, we see that we have the terminating cases $A_1 = 2$ and $B_1 = 3$. Substituting these values into Equation D.2 gives us

$$\begin{aligned} A_n &= 2Fib(3n-4) + 3Fib(3n-3) \\ &= 2(Fib(3n-4) + Fib(3n-3)) + Fib(3n-3) \\ &= 2Fib(3n-2) + Fib(3n-3) \\ &= (Fib(3n-2) + Fib(3n-3)) + Fib(3n-2) \\ &= Fib(3n-1) + Fib(3n-2) \\ &= Fib(3n) \end{aligned}$$

which can be derived from the Fibonacci property [58]

$$Fib(n) + Fib(n+1) = Fib(n+2).$$

Similarly, we can obtain expressions for the other recurrences:

$$\begin{aligned} B_n &= Fib(3n+1) \\ C_n &= Fib(3n-1) \\ D_n &= Fib(3n). \end{aligned}$$

Substituting these back into Equation D.1 gives us

$$|\mathcal{S}(K)| = 2Fib(3K-3) + 3Fib(3K-2) + Fib(3K-4) + 2Fib(3K-3)$$

which can be reduced to give us the surprising property that the number of states of the K-philosopher model is exactly $Fib(3K+1) + Fib(3K-1)$.

Bibliography

[1] G. M. Adel'son-Vel'skii and E. M. Landis. An algorithm for the organization of information. *Sov. Math. Dokl.*, 3:1259–1262, 1962.

[2] Tilak Agerwala. A complete model for representing the coordination of asynchronous processes. Hopkins Computer Research Report 32, Johns Hopkins University, Baltimore, Maryland, July 1974.

[3] Marco Ajmone Marsan, Gianfranco Balbo, Andrea Bobbio, Giovanni Chiola, Giovanni Conte, and Aldo Cumani. The effect of execution policies on the semantics and analyis of Stochastic Petri Nets. *IEEE Transactions on Software Engineering*, 15(7):832–846, July 1989.

[4] Marco Ajmone Marsan, Gianfranco Balbo, and Giovanni Conte. A class of Generalized Stochastic Petri Nets for the performance evaluation of multiprocessor systems. *ACM Transactions on Computer Systems*, 2(2):93–122, May 1984.

[5] David Aldous and Larry Shepp. The least variable phase type distribution is Erlang. *Comm. Stat. – Stochastic Models*, 3(3):467–473, 1987.

[6] Howard Anton. *Elementary Linear Algebra*. John Wiley & Sons, 1987.

[7] F Baskett, K. M. Chandy, R. R. Muntz, and F. Palacios-Gomez. Open, Closed, and Mixed networks of queues with different classes of customers. *Journal of the ACM*, 22(2):335–381, April 1975.

[8] Alex Blakemore. The cost of eliminating vanishing markings from generalized stochastic Petri nets. In *Proc. 3rd Int. Workshop on Petri Nets and Performance Models (PNPM'89)*, pages 85–92, Kyoto, Japan, December 1989. IEEE Comp. Soc. Press.

[9] B. Bollig, P. Savicky, and I. Wegener. On the improvement of variable orderings for OBDDs. In *IFIP workshop on Logic and architecture synthesis*, pages 71–80, Grenoble, 1994.

[10] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[11] Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[12] Peter Buchholz. Numerical solution methods based on structured descriptions of Markovian models. In *Computer performance evaluation*, G. Balbo and G. Serazzi, editors, pages 251–267. Elsevier Science Publishers B.V. (North-Holland), 1991.

[13] Peter Buchholz. Hierarchical structuring of superposed GSPNs. In *Proc. 7th Int. Workshop on Petri Nets and Performance Models (PNPM'97)*, pages 81–90, Saint Malo, France, June 1997. IEEE Comp. Soc. Press.

[14] Peter Buchholz. Structured analysis approaches for large Markov chains. *Applied Numerical Mathematics*, 31(4):375–404, 1999.

[15] Peter Buchholz, Gianfranco Ciardo, Susanna Donatelli, and Peter Kemper. Complexity of memory-efficient Kronecker operations with applications to the solution of Markov models. *INFORMS J. Comp.* To appear.

[16] Peter Buchholz and Peter Kemper. Numerical analysis of stochastic marked graphs. In *Proc. 6th Int. Workshop on Petri Nets and Performance Models (PNPM'95)*, pages 32–41, Durham, NC, October 1995. IEEE Comp. Soc. Press.

[17] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

[18] J. Campos, M. Silva, and S. Donatelli. Structured solution of stochastic DSSP systems. In *Proc. 7th Int. Workshop on Petri Nets and Performance Models (PNPM'97)*, pages 91–100, St. Malo, France, June 1997. IEEE Comp. Soc. Press.

[19] Giovanni Chiola. Compiling techniques for the analysis of stochastic Petri nets. *Performance Evaluation*, September 1988.

[20] Hoon Choi and Kishor S. Trivedi. Approximate performance models of polling systems using stochastic Petri nets. In *Proc. IEEE INFOCOM 1992*, pages 2306–2314, Florence, Italy, May 1992.

[21] Gianfranco Ciardo. *Analysis of large stochastic Petri net models.* PhD thesis, Duke University, Durham, NC, 1989.

[22] Gianfranco Ciardo. Petri nets with marking-dependent arc multiplicity: properties and analysis. In *Application and Theory of Petri Nets 1994 (Proc. 15th Int. Conf. on Applications and Theory of Petri Nets, Zaragoza, Spain)*, Robert Valette, editor, Lecture Notes in Computer Science 815, pages 179–198. Springer-Verlag, June 1994.

[23] Gianfranco Ciardo. Discrete-time Markovian stochastic Petri nets. In *Computations with Markov Chains*, William J. Stewart, editor, pages 339–358. Kluwer, Boston, MA, 1995.

[24] Gianfranco Ciardo, Reinhard German, and Christoph Lindemann. A characterization of the stochastic process underlying a stochastic Petri net. *IEEE Transactions on Software Engineering*, 20(7):506–515, July 1994.

[25] Gianfranco Ciardo, Joshua Gluckman, and David Nicol. Distributed state-space generation of discrete-state stochastic models. *INFORMS J. Comp.*, 10(1):82–98.

[26] Gianfranco Ciardo and Andrew S. Miner. SMART: Simulation and Markovian Analyzer for Reliability and Timing. In *Proc. IEEE International Computer Performance and Dependability Symposium (IPDS'96)*, page 60, Urbana-Champaign, IL, USA, September 1996. IEEE Comp. Soc. Press.

[27] Gianfranco Ciardo and Andrew S. Miner. Storage alternatives for large structured state spaces. In *Proc. 9th Int. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation*, R. Marie, B. Plateau, M. Calzarossa, and G. Rubino, editors, LNCS 1245, pages 44–57, Saint Malo, France, June 1997. Springer-Verlag.

[28] Gianfranco Ciardo and Andrew S. Miner. A data structure for the efficient Kronecker solution of GSPNs. In *Proc. 8th Int. Workshop on Petri Nets and Performance Models (PNPM'99)*, Peter Buchholz, editor, pages 22–31, Zaragoza, Spain, September 1999. IEEE Comp. Soc. Press.

[29] Gianfranco Ciardo and Marco Tilgner. On the use of Kronecker operators for the solution of generalized stochastic Petri nets. ICASE Report 96-35, Institute for Computer Applications in Science and Engineering, Hampton, VA, May 1996.

[30] Gianfranco Ciardo and Kishor S. Trivedi. A decomposition approach for stochastic reward net models. *Performance Evaluation*, 18(1):37–59, 1993.

[31] Erhan Çinlar. *Introduction to Stochastic Processes*. Prentice-Hall, Inc., 1975.

[32] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[33] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[34] P. J. Courtois and P. Semal. Computable bounds for conditional steady-state probabilities in large Markov chain and queueing models. *IEEE J. Sel. Areas in Comm.*, SAC-4(6):926–937, 1986.

[35] D.R. Cox. A use of complex probabilities in the theory of stochastic processes. *Proc. of the Cambridge Philosophical Society*, 51:313–319, 1955.

[36] Aldo Cumani. ESP - A package for the evaluation of stochastic Petri nets with phase-type distributed transitions times. In *Proc. Int. Workshop on Timed Petri Nets*, Torino, Italy, July 1985.

[37] Marc Davio. Kronecker products and shuffle algebra. *IEEE Transactions on Computers*, C-30(2):116–125, February 1981.

[38] D. D. Deavours and W. H. Sanders. An efficient disk-based tool for solving very large Markov models. In *Proc. 9th Int. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation*, R. Marie, B. Plateau, M. Calzarossa, and G. Rubino, editors, LNCS 1245, pages 58–71, Saint Malo, France, June 1997. Springer-Verlag.

[39] D. D. Deavours and W. H. Sanders. "On-the-fly" solution techniques for stochastic Petri nets and extensions. In *Proc. 7th Int. Workshop on Petri Nets and Performance Models (PNPM'97)*, pages 132–141, Saint Malo, France, June 1997. IEEE Comp. Soc. Press.

[40] E. W. Dijkstra, W. H. J. Feijen, and A. J. M. van Gasteren. Derivation of a termination detection algorithm for distributed computations. *Information Processing Letters*, 16(5):217–219, June 1983.

[41] Susanna Donatelli. Superposed Stochastic Automata: a class of stochastic Petri nets with parallel solution and distributed state space. *Performance Evaluation*, 18:21–26, 1993.

[42] Susanna Donatelli. Superposed generalized stochastic Petri nets: definition and efficient solution. In *Application and Theory of Petri Nets 1994 (Proc. 15th Int. Conf. on Applications and Theory of Petri Nets)*, Robert Valette, editor, Lecture Notes in Computer Science 815, pages 258–277, Zaragoza, Spain, June 1994. Springer-Verlag.

[43] R. Drechsler and B. Becker. Overview of decision diagrams. *IEEE Proc.-Comput. Digit. Tech.*, 144(3):187–193, May 1997.

[44] Paulo Fernandes, Brigitte Plateau, and William J. Stewart. Efficient descriptor-vector multiplication in stochastic automata networks. *Journal of the ACM*, 45(3):381–414, 1998.

[45] M. Fujita, Y. Matsunaga, and T. Kakuda. On the variable ordering of binary decision diagrams for the application of multi-level synthesis. In *European conference on Design automation*, pages 50–54, 1991.

[46] M. Fujita, P. McGeer, and J. C.-Y. Yang. Multi-terminal binary decision diagrams: An efficient data structure for matrix representations. *Formal Methods in System Design*, 10(2/3):149–169, 1997.

[47] Gene Golub and Charles Van Loan. *Matrix Computations*. The Johns Hopkins University Press, 1989.

[48] S. Haddad, P. Moreaux, and G. Chiola. Efficient handling of phase-type distributions in generalized stochastic Petri nets. In *Application and Theory of Petri Nets 1997 (Proc. 18th Int. Conf. on Applications and Theory of Petri Nets)*, P. Azéma and G. Balbo, editors, Lecture Notes in Computer Science 1248, pages 175–194. Springer-Verlag, June 1997.

[49] B.R. Haverkort, Alexander Bell, and Henrik Bohnenkamp. On the efficient sequential and distributed generation of very large Markov chains from stochastic Petri nets. In *Proc. 8th Int. Workshop on Petri Nets and Performance Models (PNPM'99)*, Peter Buchholz, editor, pages 12–21, Zaragoza, Spain, September 1999. IEEE Comp. Soc. Press.

[50] Philip Heidelberger and Kishor S. Trivedi. Queueing network models for parallel processing with asynchronous tasks. *IEEE Trans. Comp.*, C-31(11):1099–1109, November 1982.

[51] Philip Heidelberger and Kishor S. Trivedi. Analytic queueing models for programs with internal concurrency. *IEEE Transactions on Computers*, C-32(1):73–82, January 1983.

[52] H. Hermanns, J. Kayser, and M. Siegle. Multi terminal binary decision diagrams to represent and analyse continuous time Markov chains. In *Numerical Solution of Markov Chains*, Brigitte Plateau, William J. Stewart, and Manuel Silva, editors, pages 188–207, Zaragoza, Spain, June 1999. Prensas Universitarias de Zaragoza.

[53] O. C. Ibe, R. C. Howe, and K. S. Trivedi. Approximate availability analysis of VAXcluster systems. *IEEE Trans. Rel.*, 38(1):146–152, April 1989.

[54] T. Kam. *State Minimization of Finite State Machines using Implicit Techniques*. PhD thesis, University of California at Berkeley, 1995.

[55] John G. Kemeney and J. Laurie Snell. *Finite Markov Chains*. D. Van Nostrand Company, Inc., 1960.

[56] Peter Kemper. Numerical analysis of superposed GSPNs. *IEEE Transactions on Software Engineering*, 22(4):615–628, September 1996.

[57] Peter Kemper. Reachability analysis based on structured representations. In *Application and Theory of Petri Nets 1996, Lecture Notes in Computer Science 1091 (Proc. 17th Int. Conf. on Applications and Theory of Petri Nets, Osaka, Japan)*, J. Billington and W. Reisig, editors, pages 269–288. Springer-Verlag, June 1996.

[58] Donald Knuth. *The Art of Computer Programming*, volume 1. Addison-Wesley, third edition, 1997.

[59] Donald Knuth. *The Art of Computer Programming*, volume 2. Addison-Wesley, third edition, 1998.

[60] Donald Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley, third edition, 1998.

[61] Dexter Kozen. Results on the propositional μ-calculus. *Theor. Comput. Sci.*, 27:333–354, 1983.

[62] Leslie Lamport. *LaTeX: A Document Preparation System*. Addison-Wesley, second edition, 1994.

[63] C. Y. Lee. Representation of switching circuits by binary-decision programs. *Bell Syst. Techn. J.*, 38(4):985–999, July 1959.

[64] Varsha Mainkar and Kishor S. Trivedi. Fixed point iteration using Stochastic Reward Nets. In *Proc. 6th Int. Workshop on Petri Nets and Performance Models (PNPM'95)*, pages 21–30, Durham, NC, October 1995. IEEE Comp. Soc. Press.

[65] Z. Manna and P. Wolper. Synthesis of communicating processes from temporal logic specifications. In *Lecture Notes in Computer Science 131 (Logic of Programs: Workshop)*, D. Kozen, editor. Springer-Verlag, 1981.

[66] J. Martinez and M. Silva. A simple and fast algorithm to obtain all invariants of a generalised Petri net. In *Proc. 2nd European Workshop on Application and Theory of Petri Nets*, pages 411–422, Bad Honnef, Germany, 1981.

[67] Jean Mayo and Phil Kearns. Distributed termination detection with roughly synchronized clocks. *Information Processing Letters*, 52:105–108, 1994.

[68] Andrew S. Miner and Gianfranco Ciardo. Efficient reachability set generation and storage using decision diagrams. In *Application and Theory of Petri Nets 1999 (Proc. 20th Int. Conf. on Applications and Theory of Petri Nets, Williamsburg, VA, USA)*, H.C.M. Kleijn and Susanna Donatelli, editors, Lecture Notes in Computer Science 1639, pages 6–25. Springer-Verlag, June 1999.

[69] Andrew S. Miner, Gianfranco Ciardo, and Susanna Donatelli. Using the exact state space of a Markov model to compute approximate stationary measures. In *Proc. 2000 ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, pages 207–216, Santa Clara, CA, June 2000.

[70] D. Mitra and P. Tsoucas. Relaxations for the numerical solutions of some stochastic problems. *Communications in Statistics: Stochastic Models*, 4(3):387–419, 1988.

[71] Michael K. Molloy. *On the integration of delay and throughput measures in distributed processing models*. PhD thesis, UCLA, Los Angeles, CA, 1981.

[72] Michael K. Molloy. Performance analysis using stochastic Petri nets. *IEEE Transactions on Computers*, 31(9):913–917, September 1982.

[73] R. R. Muntz, E. de Souza e Silva, and A. Goyal. Bounding availability of repairable computer systems. *IEEE Transactions on Computers*, 12(38):1714–1723, December 1989.

[74] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[75] Marcel Neuts. *Matrix-Geometric Solutions in Stochastic Models*. The Johns Hopkins University Press, 1981.

[76] David Nicol and Gianfranco Ciardo. Automated parallelization of discrete state-space generation. *Journal of Parallel and Distributed Computing*, 47:153–167.

[77] David M. Nicol. Noncommittal barrier synchronization. *Parallel Computing*, 21:529–549, 1995.

[78] E. Pastor and J. Cortadella. Efficient encoding schemes for symbolic analysis of Petri nets. In *Proc. Design Automation and Test in Europe*, February 1998.

[79] E. Pastor and J. Cortadella. Structural methods applied to the symbolic analysis of Petri nets. In *Proc. IEEE/ACM International Workshop on Logic Synthesis*, June 1998.

[80] E. Pastor, O. Roig, J. Cortadella, and R. Badia. Petri net analysis using boolean manipulation. In *Application and Theory of Petri Nets 1994, Lecture Notes in Computer Science 815 (Proc. 15th Int. Conf. on Applications and Theory of Petri Nets, Zaragoza, Spain)*, Robert Valette, editor, pages 416–435. Springer-Verlag, June 1994.

[81] Enric Pastor, Jordi Cortadella, and Marco A. Peña. Structural methods to improve the symbolic analysis of Petri nets. In *Application and Theory of Petri Nets 1999 (Proc. 20th Int. Conf. on Applications and Theory of Petri Nets, Williamsburg, VA, USA)*, H.C.M. Kleijn and Susanna Donatelli, editors, Lecture Notes in Computer Science 1639, pages 26–45. Springer-Verlag, June 1999.

[82] Carl Petri. *Kommunikation mit Automaten*. PhD thesis, University of Bonn, Bonn, West Germany, 1962.

[83] Sergio Pissanetzky. *Sparse Matrix Technology*. Academic Press, Inc., 1984.

[84] Brigitte Plateau. On the stochastic structure of parallelism and synchronisation models for distributed algorithms. In *Proc. 1985 ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, pages 147–153, Austin, TX, USA, May 1985.

[85] Brigitte Plateau and Karim Atif. Stochastic Automata Network for modeling parallel systems. *IEEE Transactions on Software Engineering*, 17(10):1093–1108, October 1991.

[86] Brigitte Plateau and Jean-Michel Fourneau. A methodology for solving Markov models of parallel systems. *Journal of Parallel and Distributed Computing*, 12:370–387, 1991.

[87] S. P. Rana. A distributed solution of the distributed termination problem. *Information Processing Letters*, 17:43–46, July 1983.

[88] O. Roig, J. Cortadella, and E. Pastor. Verification of asynchronous circuits by BDD-based model checking of Petri nets. In *Application and Theory of Petri Nets 1995 (Proc. 16th Int. Conf. on Applications and Theory of Petri Nets, Turin, Italy)*, Giorgio De Michelis and Michel Diaz, editors, Lecture Notes in Computer Science 935, pages 374–391. Springer-Verlag, June 1995.

[89] Sheldon M. Ross. *Introduction to Probability Models*. Academic Press, 1993.

[90] Richard Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *International conference on CAD*, pages 42–47, 1993.

[91] Marco Scarpa and Andrea Bobbio. Kronecker representation of stochastic Petri nets with discrete PH distributions. In *Proc. IEEE International Computer Performance and Dependability Symposium (IPDS'98)*, Robert Geist, Guenter Haring, and James Westall, editors, pages 52–61. IEEE Comp. Soc. Press, September 1998.

[92] Matteo Sereno and Gianfranco Balbo. Computational algorithms for product form solution stochastic Petri nets. In *Proc. 5th Int. Workshop on Petri Nets and Performance Models (PNPM'93)*, pages 98–107, Toulouse, France, October 1993. IEEE Comp. Soc. Press.

[93] Kenneth C. Sevcik. Priority scheduling disciplines in queueing network models of computer systems. In *1977 IFIP Congr. Proc.*, 1977.

[94] Daniel D. Sleator and Robert E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, July 1985.

[95] Arvind Srinivasan, Timothy Kam, Sharad Malik, and Robert K. Brayton. Algorithms for discrete function manipulation. In *International Conference on CAD*, pages 92–95. IEEE Computer Society, 1990.

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

[97] William J. Stewart, Karim Atif, and Brigitte Plateau. The numerical solution of stochastic automata networks. *European Journal of Operations Research*, 86:503–525, 1995.

[98] Yukio Takahashi. Aggregate approximation for acyclic queuing networks with communication blocking. In *Queueing Networks with Blocking*, H. G. Perros and T. Altiok, editors, pages 33–47. Elsevier Science Publishers B.V., 1989.

[99] Marco Tilgner, Yukio Takahashi, and Gianfranco Ciardo. SNS 1.0: Synchronized Network Solver. In *1st International Workshop on Manufacturing and Petri Nets*, pages 215–234, Osaka, Japan, June 1996.

[100] Lorrie Tomek and Kishor S. Trivedi. Fixed-Point Iteration in Availability Modeling. In *Informatik-Fachberichte, Vol. 91: Fehlertolerierende Rechensysteme*, M. Dal Cin, editor, pages 229–240. Springer-Verlag, 1991.

[101] Rodney W. Topor. Termination detection for distributed computations. *Information Processing Letters*, 18(1):33–36, January 1984.

[102] Hendrik Vantilborgh. Exact aggregation in exponential queueing networks. *Journal of the ACM*, 25(4):620–629, October 1978.

[103] Richard Varga. *Matrix Iterative Analysis*. Prentice-Hall, Inc., 1962.

[104] N. Wirth. *Algorithm + Data Structures = Programs.* Prentice-Hall, 1976.

[105] C. Murray Woodside. Throughput calculation for basic Stochastic Rendezvous Networks. *Performance Evaluation*, 9:143–160, 1988/89.

[106] Denis Zampunièris. *The Sharing Tree Data Structure, Theory and Applications in Formal Verification.* PhD thesis, Department of Computer Science, University of Namur, Belgium, 1997.

[107] Peter Ziegler and Helena Szczerbicka. A structure based decomposition approach for GSPN. In *Proc. 6th Int. Workshop on Petri Nets and Performance Models (PNPM'95)*, pages 261–270, Durham, NC, October 1995. IEEE Comp. Soc. Press.

VITA

Andrew Stephen Miner

Andrew Miner was born during rush hour in Baltimore, Maryland on December 2, 1971. He graduated ΦBK from Randolph-Macon College in Ashland, Virginia, with a B.S. in Physics and Computer Science, in May 1993. Hoping to never again go through pages of `rigamarole` computing integrals, he began his graduate work in Computer Science at the College of William and Mary. He instead found himself spending long hours in computer labs until the software tool SMART was created. He completed his M.S. degree in May 1995. During the academic years 1998-1999 and 1999-2000 he was supported by NASA Graduate Student Researchers Program and Virginia Space Grant Consortium fellowships. After further improvements to SMART and a research trip to Torino, Italy, he found that he had accidentally completed his dissertation and was awarded the Ph.D. degree, despite his protests, in June 2000.

Dr. Miner currently resides in Colonial Williamsburg, where computers are still hand-crafted from wood and paper in the old colonial style of Jefferson.

DATA STRUCTURES FOR THE ANALYSIS OF LARGE STRUCTURED MARKOV MODELS

ABSTRACT

High-level modeling formalisms are increasingly popular tools for studying complex systems. Given a high-level model, we can automatically verify certain system properties or compute performance measures about the system. In the general case, measures must be computed using discrete-event simulations. In certain cases, exact numerical analysis is possible by constructing and analyzing the underlying stochastic process of the system, which is a continuous-time Markov chain (CTMC) in our case. Unfortunately, the number of states in the underlying CTMC can be extremely large, even if the high-level model is "small". In this thesis, we develop data structures and techniques that can tolerate these large numbers of states.

First, we present a multi-level data structure for storing the set of reachable states of a model. We then introduce the concept of event "locality", which considers the components of the model that an event may affect. We show how a state generation algorithm using our multi-level structure can exploit event locality to reduce CPU requirements.

Then, we present a symbolic generation technique based on our multi-level structure and our concept of event locality, in which operations are applied to sets of states. The extremely compact data structure and efficient manipulation routines we present allow for the examination of much larger systems than was previously possible.

The transition rate matrix of the underlying CTMC can be represented with Kronecker

algebra under certain conditions. However, the use of Kronecker algebra introduces several sources of CPU overhead during numerical solution. We present data structures, including our new data structure called *matrix diagrams*, that can reduce this CPU overhead. Using our techniques, we can compute measures for large systems in a fraction of the time required by current state-of-the-art techniques.

Finally, we present a technique for approximating stationary measures using aggregations of the underlying CTMC. Our technique utilizes exact knowledge of the underlying CTMC using our compact data structure for the reachable states and a Kronecker representation for the transition rates. We prove that the approximation is exact for models possessing a product-form solution.

Andrew Stephen Miner

Department of Computer Science

The College of William and Mary in Virginia

Advisor: Gianfranco Ciardo

DATA STRUCTURES FOR THE ANALYSIS OF LARGE STRUCTURED MARKOV MODELS