Efficient solution of GSPNs using Canonical Matrix Diagrams

Andrew S. Miner
Department of Computer Science
Iowa State University
asminer@iastate.edu

Abstract

The solution of a generalized stochastic Petri net (GSPN) is severely restricted by the size of its underlying continuous-time Markov chain. In recent work, matrix diagrams built from a Kronecker expression for the transition rate matrix of certain types of GSPNs were shown to allow for more efficient solution; however, the GSPN model requires a special form, so that the transition rate matrix has a Kronecker expression. In this paper, we extend the earlier results to GSPN models with partitioned sets of places. Specifically, we give a more restrictive definition for matrix diagrams and show that the new form is canonical. We then present an algorithm that builds a canonical matrix diagram representation for an arbitrary non-negative matrix, given encodings for the sets of rows and columns. Using this algorithm, a Kronecker expression is not required to construct the matrix diagram. The efficient matrix diagram algorithms for numerical solution presented earlier are still applicable. We apply our technique to several example GSPNs.

1. Introduction

Generalized stochastic Petri nets (GSPNs) [16] and related models (e.g., stochastic reward nets [19], stochastic activity networks [17]) are gaining increased acceptance as tools for analyzing complex systems. The popularity of such high-level formalisms is due to their ability to represent complex systems in a compact and convenient way, while still describing an underlying continuous-time Markov chain (CTMC). Performance measures can be computed by generating and analyzing the CTMC described by the model. This method suffers from the well-known *state explosion* problem: a compact GSPN can define an underlying CTMC with an enormous number of states. This problem severely limits the size of models for which an exact analysis can reasonably be attempted.

Much work has been done to cope with the state explosion problem. We consider techniques that attempt to tolerate the large number of states while still providing an "exact" solution (i.e., without approximation errors) for a fairly general class of models. These techniques must be capable of handling a very large state space, or reachability set, $\mathcal{S}$. While $\mathcal{S}$ is not required for numerical solution, it is needed to compute the measures of interest, and may be needed during the generation of the underlying CTMC. To analyze the underlying CTMC, we must also represent the infinitesimal generator matrix $\mathbf{Q}$ of the CTMC, which is a square matrix of size $|\mathcal{S}|$, usually extremely sparse. Finally, we must store at least one probability vector $\boldsymbol{\pi}$ of size $|\mathcal{S}|$, the solution to $\boldsymbol{\pi} \cdot \mathbf{Q} = \mathbf{0}$. For simplicity, we will only consider stationary solution of ergodic CTMCs in this paper; however, our work applies just as well to the analysis of absorbing CTMCs or to transient analysis of arbitrary CTMCs. Any technique for an exact solution must therefore be able to efficiently represent the three structures $\mathcal{S}$, $\mathbf{Q}$, and $\boldsymbol{\pi}$, each of which can easily require an excessive amount of memory. Indeed, while the computational cost of an exact solution may be quite high, it is ultimately the storage requirements that determine if an exact solution is possible.

There have been many techniques proposed to represent the matrix $\mathbf{Q}$. For instance, one can generate elements of $\mathbf{Q}$ as needed "on-the-fly" from the high-level model [11]. Another approach is to store $\mathbf{Q}$ on a fast disk, and to retrieve the elements as needed [10]. Some work has been done using multi-terminal binary decision diagrams (MTBDDs) [14] to encode the function $f(i_K, \ldots, i_1, j_K, \ldots, j_1) = \mathbf{Q}[i, j]$, where $(i_K, \ldots, i_1)$ and $(j_K, \ldots, j_1)$ are binary encodings of i and j. The efficiency of this approach depends in part on the number of distinct values in $\mathbf{Q}$. In the worst case, all non-zero entries in $\mathbf{Q}$ are unique and the MTBDD will require at least as much memory as sparse storage.

One approach that has received much attention is the use of Kronecker descriptions. After Plateau's initial work with synchronized automata networks [20], Kronecker representations have been applied to Petri nets and other formalisms [1, 3, 12, 15]. Using a Kronecker approach, the model is assumed to be composed of K interacting submodels, and

$\mathbf{Q}$ is written as the sum of several Kronecker products of K small matrices, where matrix k represents the change of state due to submodel k. The memory required to store the small matrices is usually a fraction of that required to store $\mathbf{Q}$ explicitly. However, the cost of numerical solution can increase by a factor of K in the worst case [2], since the elements of $\mathbf{Q}$ must be computed. Another source of overhead comes from "spurious" entries, since a Kronecker expression describes $\hat{\mathbf{Q}}$, a super-matrix of $\mathbf{Q}$. These entries are due to unreachable states, and must be ignored during numerical solution. Functional transitions can be used [13] to reduce the size of $\hat{\mathbf{Q}}$ and to increase modeling power.

In [7], we presented a new approach using a data structure called *matrix diagrams* (MDs), and showed how an MD representation of $\mathbf{Q}$ can be built from a Kronecker expression containing constant entries (i.e., ordinary Kronecker products only). One advantage of an MD representation over a Kronecker representation is that the MD encodes the matrix $\mathbf{Q}$ exactly, not a super-matrix $\hat{\mathbf{Q}}$. This completely eliminates the overhead of skipping the spurious entries. Another benefit of MDs is their ability to re-use previous results when computing a column of $\mathbf{Q}$, thus reducing the number of floating-point multiplications required for numerical solution. However, the technique described in [7] requires a Kronecker expression for $\mathbf{Q}$. In this paper, we show how MDs can be used to encode any non-negative matrix, given appropriate encodings for sets of rows and columns. This allows us to construct an MD representation for any GSPN whose set of places can be partitioned, without using a Kronecker representation.

The remainder of the paper is organized as follows. Section 2 briefly describes multi-valued decision diagrams (MDDs) and how they can encode sets of states (in particular the reachability set $\mathcal{S}$). In Section 3, we give a new definition of matrix diagrams, and prove that the new definition is a canonical representation of matrices, given MDD encodings for the sets of rows and columns. In Section 4, we present our algorithm for building the canonical matrix diagram representation for any non-negative matrix. In Section 5, we describe how to apply matrix diagrams towards the solution of GSPNs. Experimental results are presented in Section 6. Finally, Section 7 concludes our work.

2. Encoding sets of states

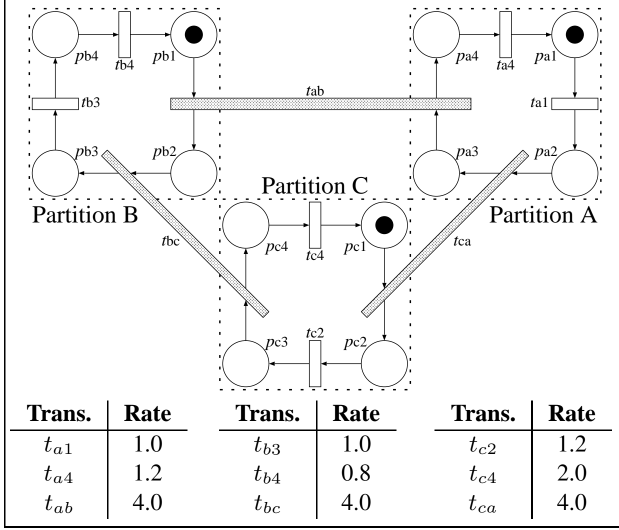
The use of matrix diagrams requires MDD encodings for sets of states. In this section, we review the definition of MDDs, and how they are used to encode sets of states.

2.1. Multi-valued decision diagrams

A multi-valued decision diagram (MDD) [21] is a data structure used to encode integer functions of the form $f : \mathcal{N}_K \times \cdots \times \mathcal{N}_1 \rightarrow \{0, \ldots, M-1\}$, where $\mathcal{N}_k = \{0, \ldots, N_k - 1\}$. Formally, an MDD is a directed, acyclic graph containing *terminal* and *non-terminal* nodes. Each terminal node is labeled with an integer $m \in \{0, \ldots, M-1\}$, and is written simply m. A non-terminal node is labeled with a variable identifier x_k, and contains N_k pointers to other nodes. Each pointer corresponds to a *cofactor* of f, defined as $f_{x_k=c} = f(x_K, \ldots, x_{k+1}, c, x_{k-1}, \ldots, x_1)$ for variable x_k and constant c. A non-terminal node labeled with variable x_k representing function f is then written as the (N_k+1)-tuple $(x_k, f_{x_k=0}, \ldots, f_{x_k=N_k-1})$, and we say it is a *level-k* node. We say the terminal nodes are level-0 nodes. An *ordered* MDD (OMDD) is an MDD such that every downward pointer from a level-k node goes to a node whose level is less than k. Every OMDD has a single *top-level* node with level larger than all other nodes. Two nodes are equivalent if they are both terminal nodes with the same label or they are both level-k non-terminal nodes with the same (N_k+1)-tuple $(x_k, f_{x_k=0}, \ldots, f_{x_k=N_k-1})$. We say a level-$k$ node is *redundant* if it is a non-terminal node with (N_k+1)-tuple $(x_k, d, \ldots, d)$; that is, all downward pointers are to the same node. A *reduced* OMDD (ROMDD) is an OMDD with no equivalent nodes and no redundant nodes. It has been shown [21] that ROMDDs are a canonical representation for integer functions: given any integer function and a variable ordering, there is exactly one ROMDD representing that function. To simplify the presentation in the rest of the paper, from now on we assume that all MDDs are ROMDDs, except that we require the top level to be level K, and downward pointers from a level-$(k+1)$ node must go either to node 0 or to a level-k node. This is achieved by adding redundant nodes, and it can be shown that this is also a canonical form.

2.2. Representing sets of states with MDDs

Using MDDs to encode sets of states is similar to the idea of using a multi-level structure [6]. We assume that the set of places of our GSPN model has been partitioned into K subsets: $\mathcal{P} = \mathcal{P}_K \cup \cdots \cup \mathcal{P}_1,\ i \neq j \Rightarrow \mathcal{P}_i \cap \mathcal{P}_j = \emptyset$. A marking of the GSPN can be considered a collection of submarkings, $\mathbf{m} = (\mathbf{m}_K, \ldots, \mathbf{m}_1)$, where submarking $\mathbf{m}_k$ describes the number of tokens in each place in $\mathcal{P}_k$. Given a set of markings $\mathcal{S}$, the set of possible submarkings for partition $\mathcal{P}_k$ is $\mathcal{S}_k = \{\mathbf{m}_k : \exists(\mathbf{m}_K, \ldots, \mathbf{m}_k, \ldots, \mathbf{m}_1) \in \mathcal{S}\}$. In some cases, we can generate the sets $\mathcal{S}_K, \ldots, \mathcal{S}_1$ in isolation (i.e., before generating $\mathcal{S}$); otherwise, we can generate the sets $\mathcal{S}_K, \ldots, \mathcal{S}_1$ during or after the generation of $\mathcal{S}$. Either way, we can maintain an index for each possible submarking. Formally, for each partition $\mathcal{P}_k$, we have an indexing bijection $\phi_k : \mathcal{S}_k \rightarrow \mathcal{N}_k$, where $\mathcal{N}_k = \{0, \ldots, |\mathcal{S}_k|-1\}$. A marking $(\mathbf{m}_K, \ldots, \mathbf{m}_1)$ can thus be represented as a collection of indices $(s_K, \ldots, s_1)$, where $s_k = \phi_k(\mathbf{m}_k)$. We



Trans.	Rate	Trans.	Rate	Trans.	Rate
t_{a1}	1.0	t_{b3}	1.0	t_{c2}	1.2
t_{a4}	1.2	t_{b4}	0.8	t_{c4}	2.0
t_{ab}	4.0	t_{bc}	4.0	t_{ca}	4.0

Figure 1. Running GSPN example

Partition A		Partition B		Partition C	
Index	$[p_{a1}, p_{a2}, p_{a3}, p_{a4}]$	Index	$[p_{b1}, p_{b2}, p_{b3}, p_{b4}]$	Index	$[p_{c1}, p_{c2}, p_{c3}, p_{c4}]$
0	[1,0,0,0]	0	[1,0,0,0]	0	[1,0,0,0]
1	[0,1,0,0]	1	[0,1,0,0]	1	[0,1,0,0]
2	[0,0,1,0]	2	[0,0,1,0]	2	[0,0,1,0]
3	[0,0,0,1]	3	[0,0,0,1]	3	[0,0,0,1]

Figure 2. Local states for the running example

will refer to collections of indices as *states*, and we shall use states and markings interchangeably with the understanding that conversion is done via functions ϕ_k and ϕ_k^{-1}.

As a running example, we use the simple GSPN model in Figure 1, and partition the set of places according to the dashed lines. In Figure 2, we show the sets of possible submarkings and their indices, assuming the initial marking in Figure 1. State $(1, 3, 3)$ refers to submarking 1 for partition A, submarking 3 for partition B, and submarking 3 for partition C. Thus, state $(1, 3, 3)$ corresponds to the marking with one token in places p_{a2}, p_{b4} and p_{c4}, and zero tokens in the remaining places.

Note that the set $\mathcal{S}$ is a subset of the *potential* markings $\hat{\mathcal{S}} = \mathcal{S}_K \times \cdots \times \mathcal{S}_1$, since for any marking in $\mathcal{S}$, submarking $\mathbf{m}_k$ for partition $\mathcal{P}_k$ is present in $\mathcal{S}_k$. One way to represent the set $\mathcal{S}$ is to specify which of the potential markings are in $\mathcal{S}$, using a function $\chi : \hat{\mathcal{S}} \to \{0, 1\}$, defined by $\chi(s_K, \ldots, s_1) = 1 \Leftrightarrow (s_K, \ldots, s_1) \in \mathcal{S}$ where $s_k \in \mathcal{N}_k$. Since χ is an integer function, it can be encoded by an MDD. We note in passing that another way to represent the function χ is to use a vector of bits [15].

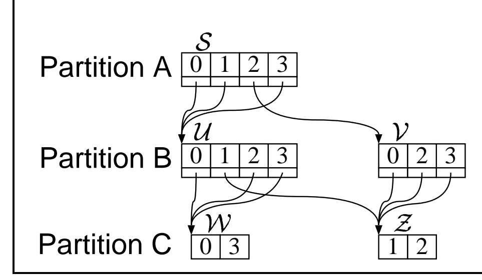


Figure 3. MDD representation of $\mathcal{S}$

For our work, we must construct an MDD encoding for the reachability set $\mathcal{S}$. MDDs can be used during generation of $\mathcal{S}$ except in certain cases when the model contains immediate transitions or marking-dependent arc cardinalities [4, 18]. When MDDs cannot be used, we can generate $\mathcal{S}$ using a traditional data structure, and then construct the MDD function χ once $\mathcal{S}$ is known. In this case, an excellent choice is the multi-level data structure presented in [6], which is equivalent to an unreduced MDD. For a discussion of several practical issues related to reachability set storage using MDDs, such as how to enumerate the reachable states efficiently, we refer the interested reader to [7].

Continuing our running example, the MDD encoding of the reachability set $\mathcal{S}$ for the GSPN of Figure 1 is shown in Figure 3. The MDD shown assumes that the places are partitioned as in Figure 1, and the MDD nodes are ordered accordingly. Note that, throughout this paper, we use a "sparse" MDD representation: for a given level-k node f, a value v is omitted if $f_{x_k=v} = 0$. For instance, in node $\mathcal{V}$ the value 1 is omitted. Also, pointers are not shown for level-1 nodes, since they must be to node 1. A state is reachable if there is a path from the top-level node to node 1 following the appropriate downward pointers. For instance, in Figure 3, state $(1, 3, 3)$ is reachable: in node $\mathcal{S}$ we follow the pointer $\mathcal{S}_{x_A=1}$ to reach node $\mathcal{U}$, we then follow the pointer $\mathcal{U}_{x_B=3}$ to reach node $\mathcal{W}$, finally we follow the pointer $\mathcal{W}_{x_C=3}$ (not shown) to node 1. The state $(2, 1, 1)$ is not reachable: $\mathcal{S}_{x_A=2} = \mathcal{V}$, and $\mathcal{V}_{x_B=1} = 0$.

For a level-k node f, the number of non-zero downward pointers is written $\eta(f)$. Each non-zero downward pointer $f_{x_k=v}$ is assigned an index, denoted $\eta(f, v)$, where $\eta(f, v)$ is the number of non-zero downward pointers for values less than v, and $\eta(f, v)$ is undefined if $f_{x_k=v} = 0$. For the example in Figure 3, we have $\eta(\mathcal{V}) = 3$, since node $\mathcal{V}$ has three non-zero downward pointers. Similarly, $\eta(\mathcal{Z})$ is 2, since node $\mathcal{Z}$ has two pointers to node 1. The downward pointers for node $\mathcal{V}$ are indexed: $\eta(\mathcal{V}, 0) = 0$, $\eta(\mathcal{V}, 2) = 1$, and $\eta(\mathcal{V}, 3) = 2$. Similarly, we have $\eta(\mathcal{Z}, 1) = 0$ and $\eta(\mathcal{Z}, 2) = 1$. Omitted values have undefined indices: $\eta(\mathcal{V}, 1) = \eta(\mathcal{Z}, 0) = \eta(\mathcal{Z}, 3) =$ *undefined*.

3. Matrix diagrams

A matrix diagram (MD) is a directed, acyclic graph used to represent matrices. The terminal nodes of the graph are labeled 0 and 1, and are considered level-0 nodes. The level-k non-terminal nodes of the graph are labeled with variable m_k, and each contain a matrix $\mathbf{M}$ of pairs (real-value, node pointer). We denote the real-value component of an element as $\mathbf{M}[x, y]_{\rm v}$, and the node-pointer component as $\mathbf{M}[x, y]_{\rm d}$. By definition, we have $\mathbf{M}[x, y]_{\rm v} = 0 \Leftrightarrow \mathbf{M}[x, y]_{\rm d} = 0$.

An *ordered* MD (OMD) is an MD such that every downward pointer from a level-k node goes to a node whose level is less than k. Every OMD has a single *top-level* node with level larger than all other nodes. To simplify our presentation, from now on we assume that all MDs are OMDs, the top level is level K, and downward pointers from a level-$(k+1)$ node go either to node 0 or to a level-k node. A matrix diagram is *reduced* if it contains no equivalent nodes. That is, for any two level-k nodes $(m_k, \mathbf{M})$ and $(m_k, \mathbf{N})$ we either have matrices $\mathbf{M}$ and $\mathbf{N}$ of different sizes, or $\mathbf{M}[x, y] \neq \mathbf{N}[x, y]$ for some row x and column y.

Note that our definition of matrix diagrams differs slightly from the one we originally presented in [7]; namely, the matrix elements in [7] are allowed to be *sets* of pairs (real-value, pointer). To obtain a canonical form, sets of pairs cannot be allowed, and we must introduce additional restrictions.

3.1. Canonical form

A *canonical* MD (CMD) consists of a triple $(M, \mathcal{R}, \mathcal{C})$, where M is a MD with variables $m_K, \ldots, m_1$, $\mathcal{R}$ and $\mathcal{C}$ are MDDs encoding the sets of rows and columns, with variables $r_K, \ldots, r_1$ and $c_K, \ldots, c_1$, and the following properties hold.

1. $\mathcal{R}$ and $\mathcal{C}$ have the same top level, and $\mathcal{R} \neq 0, \mathcal{C} \neq 0$.
2. If $M \neq 0$, then M has the same top level as $\mathcal{R}$ and $\mathcal{C}$.
3. If the top-level node of M is non-terminal, then its matrix $\mathbf{M}$ has $\eta(\mathcal{R})$ rows and $\eta(\mathcal{C})$ columns.
4. For every $\mathbf{M}[x, y]$, $0 \leq \mathbf{M}[x, y]_{\rm v} \leq 1$.
5. Matrix $\mathbf{M}$ contains an element with $\mathbf{M}[x, y]_{\rm v} = 1$.
6. For every $\mathbf{M}[x, y]$, if $\mathbf{M}[x, y] \neq (0, 0)$ then the triple $(M', \mathcal{R}', \mathcal{C}')$ is a CMD, where $M' = \mathbf{M}[x, y]_{\rm d}$, $\mathcal{R}' = \mathcal{R}_{r_k=i}$, $\mathcal{C}' = \mathcal{C}_{c_k=j}$, $x = \eta(\mathcal{R}, i)$, $y = \eta(\mathcal{C}, j)$.

A reduced CMD (RCMD) has a reduced MD M.

An example of a (reduced) CMD is shown in Figure 4. The blank spaces correspond to the entries $(0, 0)$. The nodes of $\mathcal{R}$ are drawn vertically to emphasize that they correspond to the number of rows in the matrices in M. Looking at

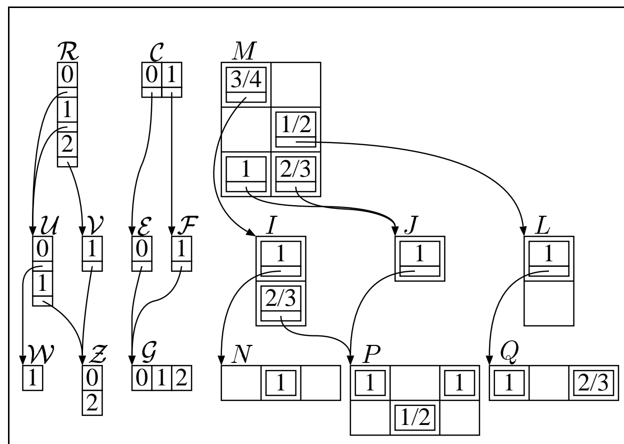


Figure 4. An example CMD

the CMD $(M, \mathcal{R}, \mathcal{C})$ we see node M contains matrix $\mathbf{M}$ with $\eta(\mathcal{R}) = 3$ rows and $\eta(\mathcal{C}) = 2$ columns. Following $\mathcal{R}_{r_3=0}$ brings us to node $\mathcal{U}$, $\mathcal{C}_{c_3=0}$ brings us to node $\mathcal{E}$, and $\mathbf{M}[\eta(\mathcal{R}, 0), \eta(\mathcal{C}, 0)] = \mathbf{M}[0, 0] = (\frac{3}{4}, I)$. Thus $(I, \mathcal{U}, \mathcal{E})$ is also a CMD. Note the matrix of node I contains $\eta(\mathcal{U}) = 2$ rows and $\eta(\mathcal{E}) = 1$ columns.

3.2. The encoded matrix

Given a CMD $(M, \mathcal{R}, \mathcal{C})$, we can compute an element of the matrix encoded by the CMD using the recurrence

$$\begin{aligned} Enc_K(M, \mathcal{R}, \mathcal{C})[\mathbf{i}, \mathbf{j}] &= \\ &\mathbf{M}[x_K, y_K]_{\rm v} \cdot Enc_{K-1}(M', \mathcal{R}', \mathcal{C}')[\mathbf{i}', \mathbf{j}'] \end{aligned}$$

where $\mathbf{i} = (i_K, \ldots, i_1) \in \mathcal{R}$, $\mathbf{j} = (j_K, \ldots, j_1) \in \mathcal{C}$, $x_K = \eta(\mathcal{R}, i_K)$, $y_K = \eta(\mathcal{C}, j_K)$, $M' = \mathbf{M}[x_K, y_K]_{\rm d}$, $\mathcal{R}' = \mathcal{R}_{r_K=i_K}$, $\mathcal{C}' = \mathcal{C}_{c_K=j_K}$, $\mathbf{i}' = (i_{K-1}, \ldots, i_1)$, and $\mathbf{j}' = (j_{K-1}, \ldots, j_1)$. The terminal case of the recurrence is $Enc_0(1, 1, 1) = 1$. We also define the special case $Enc_K(0, \mathcal{R}, \mathcal{C})[\mathbf{i}, \mathbf{j}] = 0$ so that the CMD with matrix diagram equal to node 0 encodes a matrix of all zeroes.

As an example, the matrix encoded by the CMD in Figure 4 is shown in Figure 5. The matrix to the far right shows the encoded values, and the rows and columns are labeled with the appropriate states from $\mathcal{R}$ and $\mathcal{C}$. The lines indicate the submatrices due to node M of the matrix diagram. Note in particular the blocks of zeroes, corresponding to the $(0, 0)$ entries of the matrix of node M. To compute element $(0, 1, 2), (0, 0, 1)$ of the matrix, we first compute $x_3 = \eta(\mathcal{R}, 0) = 0$ and $y_3 = \eta(\mathcal{C}, 0) = 0$. We then consider element $0, 0$ of the matrix of node M. The value is $\frac{3}{4}$, and we follow the appropriate pointers from nodes M, $\mathcal{R}$, and $\mathcal{C}$ to obtain CMD $(I, \mathcal{U}, \mathcal{E})$. Now we have $x_2 = \eta(\mathcal{U}, 1) = 1$ and $y_2 = \eta(\mathcal{E}, 0) = 0$, and element $1, 0$ of the matrix of node I gives us a value of $\frac{2}{3}$. Following the appropriate

	0	1
0	$\frac{3}{4}Enc_2(I,\mathcal{U},\mathcal{E})$	$\mathbf{0}$
1	$\mathbf{0}$	$\frac{1}{2}Enc_2(L,\mathcal{U},\mathcal{F})$
2	$Enc_2(J,\mathcal{V},\mathcal{E})$	$\frac{2}{3}Enc_2(J,\mathcal{V},\mathcal{F})$

=

	00	11
00	$\frac{3}{4}Enc_1(N,\mathcal{W},\mathcal{G})$	$\mathbf{0}$
01	$\frac{1}{2}Enc_1(P,\mathcal{Z},\mathcal{G})$	$\mathbf{0}$
10	$\mathbf{0}$	$\frac{1}{2}Enc_1(Q,\mathcal{W},\mathcal{G})$
11	$\mathbf{0}$	$\mathbf{0}$
21	$Enc_1(P,\mathcal{Z},\mathcal{G})$	$\frac{2}{3}Enc_1(P,\mathcal{Z},\mathcal{G})$

=

	000	001	002	110	111	112
001	0	$\frac{3}{4}$	0	0	0	0
010	$\frac{1}{2}$	0	$\frac{1}{2}$	0	0	0
012	0	$\frac{1}{4}$	0	0	0	0
101	0	0	0	$\frac{1}{2}$	0	$\frac{1}{3}$
110	0	0	0	0	0	0
112	0	0	0	0	0	0
210	1	0	1	$\frac{2}{3}$	0	$\frac{2}{3}$
212	0	$\frac{1}{2}$	0	0	$\frac{1}{3}$	0

Figure 5. Computing $Enc_3(M,\mathcal{R},\mathcal{C})$ from the CMD in Figure 4.

pointers from nodes I, $\mathcal{U}$, and $\mathcal{E}$ gives us CMD $(P,\mathcal{Z},\mathcal{G})$. Finally, we have $x_1 = \eta(\mathcal{Z},2) = 1$ and $y_1 = \eta(\mathcal{G},1) = 1$, and element $1,1$ of the matrix of node P gives us a value of $\frac{1}{2}$. The downward pointers all lead to terminal node 1, so we are finished, and the encoded element is $\frac{3}{4}\cdot\frac{2}{3}\cdot\frac{1}{2} = \frac{1}{4}$.

Note that, as defined, a CMD encodes a matrix whose elements are between zero and one (inclusive), and whose maximal element is one. To represent an arbitrary non-negative matrix $\mathbf{A}$, one possibility is to encode $\frac{1}{a}\cdot\mathbf{A}$, where $a \neq 0$ is the largest element of $\mathbf{A}$, and then multiply each element returned by Enc_K by a. Another possibility, which we adopt, is to relax the restriction for the values of the matrix of the top-level node. Thus our top-level matrix has a maximal value of a instead of one.

3.3. Proof of canonicity

To prove that RCMDs are a canonical representation, we must show that any non-negative matrix can be represented as a RCMD (given the MDDs $\mathcal{R}$ and $\mathcal{C}$), and that the RCMD is unique. We will only show that the RCMD obtained is unique; Section 4 discusses how to build the RCMD that encodes a given matrix, but we do not formally prove that the algorithm is correct due to space considerations.

Lemma 1 $Enc_k(M,\mathcal{R},\mathcal{C}) = \mathbf{0}^{|\mathcal{R}|\times|\mathcal{C}|} \Leftrightarrow M = 0$.

Proof: By definition of *Enc*, we have $M = 0 \Rightarrow Enc_k(M,\mathcal{R},\mathcal{C}) = \mathbf{0}^{|\mathcal{R}|\times|\mathcal{C}|}$. If $M \neq 0$, then either M is 1, and we have $Enc_0(1,1,1) = 1$, or M is a non-terminal node. By definition, the matrix associated with a non-terminal node contains an element with value 1 whose pointer is not to terminal node zero. Thus there exists a path from the level-k node of M to node 1, where the product of the values along the path is one. Thus, $Enc_k(M,\mathcal{R},\mathcal{C})$ will produce at least one nonzero element if $M \neq 0$. □

Theorem 1 $Enc_k(M,\mathcal{R},\mathcal{C}) = Enc_k(M',\mathcal{R},\mathcal{C}) \Leftrightarrow M = M'$

Proof: Trivially, $Enc_k(M,\mathcal{R},\mathcal{C}) = Enc_k(M',\mathcal{R},\mathcal{C})$ when $M = M'$. We show that $Enc_k(M,\mathcal{R},\mathcal{C}) = Enc_k(M',\mathcal{R},\mathcal{C}) \Rightarrow M = M'$ by induction on k.

The base case is $k = 0$, where we either have $Enc_0(1,\mathcal{R},\mathcal{C}) = 1$ or $Enc_0(0,\mathcal{R},\mathcal{C}) = 0$. Thus the property holds in the base case.

Now we assume the property holds for $0 \le k \le K$, and prove it must also hold for $k = K+1$. Given $Enc_{K+1}(M,\mathcal{R},\mathcal{C}) = Enc_{K+1}(M',\mathcal{R},\mathcal{C})$, we must show that $M = M'$. If $M = 0$ then from Lemma 1 we trivially get $M' = 0$. Otherwise, M and M' are both level $K+1$ nodes, and we know that the matrices $\mathbf{M}$ and $\mathbf{M}'$ both have $\eta(\mathcal{R})$ rows and $\eta(\mathcal{C})$ columns. Since the MD is reduced, $M = M'$ if and only if $\mathbf{M}[x,y] = \mathbf{M}'[x,y]$ for all elements. From the definition of *Enc*, we know that

$$\mathbf{M}[x,y]_{\mathrm{v}} \cdot Enc_K(\mathbf{M}[x,y]_{\mathrm{d}},\mathcal{R}_{r_{K+1}=i},\mathcal{C}_{c_{K+1}=j}) = \mathbf{M}'[x,y]_{\mathrm{v}} \cdot Enc_K(\mathbf{M}'[x,y]_{\mathrm{d}},\mathcal{R}_{r_{K+1}=i},\mathcal{C}_{c_{K+1}=j})$$

for $x = \eta(\mathcal{R},i)$ and $y = \eta(\mathcal{C},j)$. If $\mathbf{M}[x,y]_{\mathrm{v}} = 0$, then we have $\mathbf{M}[x,y]_{\mathrm{d}} = 0$, and from Lemma 1 we get $\mathbf{M}'[x,y] = (0,0)$, and thus $\mathbf{M}[x,y] = \mathbf{M}'[x,y]$. Otherwise, we have $\mathbf{M}[x,y]_{\mathrm{v}} \neq 0$, and we obtain

$$Enc_K(\mathbf{M}[x,y]_{\mathrm{d}},\mathcal{R}_{r_{K+1}=i},\mathcal{C}_{c_{K+1}=j}) = \frac{\mathbf{M}'[x,y]_{\mathrm{v}}}{\mathbf{M}[x,y]_{\mathrm{v}}} \cdot Enc_K(\mathbf{M}'[x,y]_{\mathrm{d}},\mathcal{R}_{r_{K+1}=i},\mathcal{C}_{c_{K+1}=j}).$$

Since $\mathbf{M}[x,y]_{\mathrm{v}} \neq 0$, we know that $\mathbf{M}[x,y]_{\mathrm{d}} \neq 0$. Thus by Lemma 1, $Enc_K(\mathbf{M}[x,y]_{\mathrm{d}},\mathcal{R}_{r_{K+1}=i},\mathcal{C}_{c_{K+1}=j})$ is not a zero matrix, and thus has a maximal value of exactly 1. This implies that $Enc_K(\mathbf{M}'[x,y]_{\mathrm{d}},\mathcal{R}_{r_{K+1}=i},\mathcal{C}_{c_{K+1}=j})$ is not a zero matrix, and thus has a maximal value of exactly 1. Thus, $\mathbf{M}'[x,y]_{\mathrm{v}} = \mathbf{M}[x,y]_{\mathrm{v}}$, otherwise one matrix cannot have maximal value of 1. This gives us

$$Enc_K(\mathbf{M}[x,y]_{\mathrm{d}},\mathcal{R}_{r_{K+1}=i},\mathcal{C}_{c_{K+1}=j}) = Enc_K(\mathbf{M}'[x,y]_{\mathrm{d}},\mathcal{R}_{r_{K+1}=i},\mathcal{C}_{c_{K+1}=j})$$

and so by the inductive hypothesis we have $\mathbf{M}'[x,y]_{\mathrm{d}} = \mathbf{M}[x,y]_{\mathrm{d}}$. Thus, $\mathbf{M}[x,y] = \mathbf{M}'[x,y]$. □

4. Building CMDs

In this section, we develop algorithms to build a RCMD, given MDDs encoding the sets of rows and columns, and a desired matrix to encode.

4.1. A Simple Algorithm

One way to construct a RCMD is to construct a MD, then manipulate the MD until we have a RCMD. This is the approach used in algorithm Build1, shown in Figure 6. Given a set of rows $\mathcal{R}$ and a set of columns $\mathcal{C}$, both encoded as MDDs, and a matrix $\mathbf{E}$ to encode, procedure Build1 will build a RCMD $(M, \mathcal{R}, \mathcal{C})$ such that $Enc_K(M, \mathcal{R}, \mathcal{C}) = \mathbf{E}$. First, we add each entry of matrix $\mathbf{E}$ to an unreduced MD M using procedure AddEntry, which creates the appropriate path from the top node to node 1, with the encoded element as the value at the bottom level, and values of one at all other levels. As an example, Figure 7 shows the unreduced MD obtained from the matrix in Figure 5 after all entries have been added (i.e., before line 5 of Build1 executes).

Once we have obtained our unreduced MD, we must adjust the values so that we have a CMD. This requires *normalizing* the matrix values so that each value is between zero and one, and the maximal value is one (properties 4 and 5 in Section 3.1). This is done with procedure Normalize, shown in Figure 6. Note that Normalize allows arbitrary values in the level-K node, so that we can encode non-negative matrices whose maximal values are not necessarily one.

Finally, we must reduce our CMD using procedure Reduce, which eliminates duplicate nodes in a bottom-up fashion. Procedure UniqueInsert, not shown, inserts a node into a hash table; if an identical node (i.e. a node at the same level with an equal matrix) is found in the hash table, the duplicate node is deleted and the node in the hash table is returned. Due to floating-point roundoff errors, we cannot do a strict comparison of elements when considering if two matrices are equal; instead, we consider the values of elements to be equal if their relative difference is less than ϵ. Thanks to theorem 1, we know that the node returned by Reduce is the only (appropriately-sized) node that encodes a given submatrix. When the MD in Figure 7 is normalized and reduced, we obtain the CMD in Figure 4.

```
AddEntry(M, R, C, i, j, val)
 1: k ← top level of R;
 2: if M = 0 then
 3:    M ← new node(k, #rows = η(R), #cols = η(C));
 4: end if
 5: x ← η(R, i[k]);   y ← η(C, j[k]);
 6: if k = 1 then     • Bottom level, terminate recursion
 7:    M[x, y] ← (val, 1);
 8: else
 9:    M' ← M[x, y]_d;  R' ← R_{r_k = i[k]};  C' ← C_{c_k = j[k]};
10:    M[x, y] ← (1, AddEntry(M', R', C', i, j, val));
11: end if
12: return M;
```

```
MultiplyBy(M, scalar)
 1: for each x, y such that M[x, y] ≠ (0, 0) do
 2:    M[x, y]_v ← M[x, y]_v · scalar;
 3: end for
```

```
Normalize(M)
 1: if M = 1 then              • Terminate recursion
 2:    return 1;
 3: end if
 4: k ← top level of M;   scale ← 0;
 5: for each x, y such that M[x, y] ≠ (0, 0) do
 6:    M[x, y]_v ← M[x, y]_v · Normalize(M[x, y]_d);
 7:    scale ← max(scale, M[x, y]_v);
 8: end for
 9: if k ≠ K then     • Allow arbitrary values at level K
10:    MultiplyBy(M, 1/scale);
11: end if
12: return scale;
```

```
Reduce(M)
 1: if M = 0 or M = 1 then      • Terminate recursion
 2:    return M;
 3: end if
 4: for each x, y such that M[x, y] ≠ (0, 0) do
 5:    M[x, y]_d ← Reduce(M[x, y]_d);
 6: end for
 7: return UniqueInsert(M);
```

```
Build1(R, C, E)
 1: M ← 0;
 2: for each i ∈ R, j ∈ C such that E[i, j] ≠ 0 do
 3:    M ← AddEntry(M, R, C, i, j, E[i, j]);
 4: end for
 5: Normalize(M);
 6: return Reduce(M);
```

Figure 6. Simple CMD construction

4.2. A Better Algorithm

The problem with Build1 is that we must build an entirely unreduced MD of our matrix $\mathbf{E}$, which may require a substantial amount of memory. A better approach is to reduce our matrix diagram periodically, to save memory. This is the idea behind Build2, shown in Figure 8. The last parameter, *freq*, determines the frequency of the reductions: when the row to be inserted differs from the previous row at a component $k >$ *freq*, we perform a reduction. Thus, when *freq* $= K$, Build2 is the same as Build1; when *freq* $= 0$, we

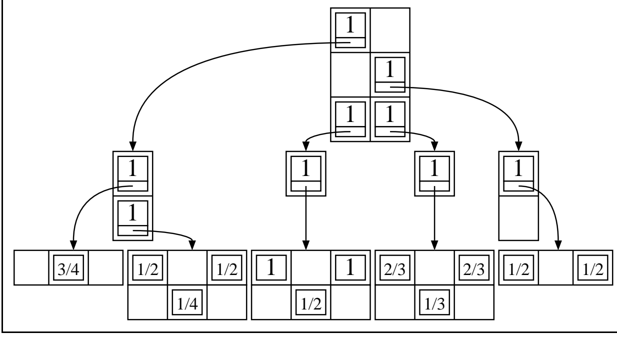
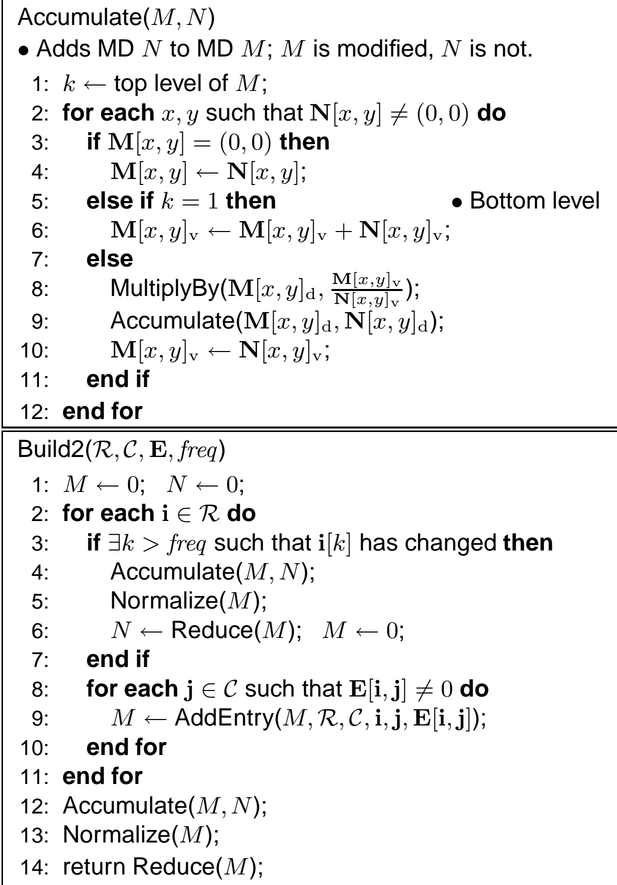


Figure 7. Matrix diagram before normalization



```
Accumulate(M, N)
• Adds MD N to MD M; M is modified, N is not.
 1: k ← top level of M;
 2: for each x, y such that N[x, y] ≠ (0, 0) do
 3:    if M[x, y] = (0, 0) then
 4:       M[x, y] ← N[x, y];
 5:    else if k = 1 then                    • Bottom level
 6:       M[x, y]_v ← M[x, y]_v + N[x, y]_v;
 7:    else
 8:       MultiplyBy(M[x, y]_d, M[x,y]_v / N[x,y]_v);
 9:       Accumulate(M[x, y]_d, N[x, y]_d);
10:       M[x, y]_v ← N[x, y]_v;
11:    end if
12: end for
```

```
Build2(R, C, E, freq)
 1: M ← 0;   N ← 0;
 2: for each i ∈ R do
 3:    if ∃k > freq such that i[k] has changed then
 4:       Accumulate(M, N);
 5:       Normalize(M);
 6:       N ← Reduce(M);   M ← 0;
 7:    end if
 8:    for each j ∈ C such that E[i, j] ≠ 0 do
 9:       M ← AddEntry(M, R, C, i, j, E[i, j]);
10:    end for
11: end for
12: Accumulate(M, N);
13: Normalize(M);
14: return Reduce(M);
```

Figure 8. Better CMD construction

reduce after each row is added.

Reduction is performed in lines 4-6 and 12-14. We maintain a RCMD N and an unreduced MD M which contains all the rows that have been added since we created N. During reduction, we must create an RCMD that encodes the matrix $Enc_K(M, \mathcal{R}, \mathcal{C}) + Enc_K(N, \mathcal{R}, \mathcal{C})$. The addition is performed by procedure Accumulate, which adds N to M without modifying N. Since the RCMD nodes of N have already been normalized and reduced, we prefer to modify nodes of M instead. The main part of the addition (lines 8-10 of Accumulate) is based on the property $c\mathbf{A} + d\mathbf{B} = d(\frac{c}{d}\mathbf{A} + \mathbf{B})$. For level-1 nodes, Accumulate is the same as matrix addition (line 6). We then normalize and reduce MD M, which becomes the new RCMD N. After calling Accumulate, M may contain both MD nodes and RCMD nodes. To reduce computational costs, we use modified versions of Normalize and Reduce that return immediately when called with RCMD nodes.

5. Numerical solution of GSPNs with CMDs

We now summarize the steps that must be taken to use CMDs for solution of GSPNs, and discuss practical issues.

Partition the set of places. It is important to note that, in theory, *any* partitioning will work and lead to correct results. At one extreme, each partition consists of a single place, and our CMD will have $|\mathcal{P}|$ levels. Since the number of floating-point multiplications required to determine a matrix element from a CMD depends on the number of levels, this can lead to a high computational overhead during numerical solution. At the other extreme, we have a single partition. In this case, our MDD contains a single node with $|\mathcal{S}|$ pointers to node 1 (which is equivalent to explicit storage of $\mathcal{S}$), and our CMD contains a single node whose matrix corresponds to the transition rate matrix $\mathbf{R}$, where the downward pointers are to node 1 for the non-zero values (which is equivalent to sparse storage for $\mathbf{R}$). Ultimately, the efficiency of our approach depends on the choice of partition. While we give a brief example in Section 6, a thorough discussion of how to choose a partition is beyond the scope of this paper.

Build an MDD that encodes $\mathcal{S}$. If possible, we can generate $\mathcal{S}$ using MDDs [4, 18]. Otherwise, we can generate $\mathcal{S}$ using traditional methods, and then construct an MDD.

Build an RCMD that encodes $\mathbf{R}$. We build a RCMD representation for the transition rate matrix $\mathbf{R}$, and store the row sums of $\mathbf{R}$ (used to compute the diagonal elements of $\mathbf{Q}$ as needed) either explicitly in a full vector or in a second matrix diagram, as done in [7]. Our RCMD is constructed using a modified version of Build2, using $\mathbf{R}$ for $\mathbf{E}$ and $\mathcal{S}$ for both $\mathcal{R}$ and $\mathcal{C}$. However, we do not explicitly build $\mathbf{R}$ and then call Build2; instead, we generate the rows of $\mathbf{R}$ as needed from our GSPN model within Build2.

The CMD encoding of $\mathbf{R}$ for our running GSPN example is shown in Figure 9. The MDD for the set $\mathcal{S}$, which is used for the sets of rows and columns, is shown in Figure 3 so we do not duplicate it here.

Compute stationary probabilities. The efficient solution techniques presented in [7] apply also to CMDs, since

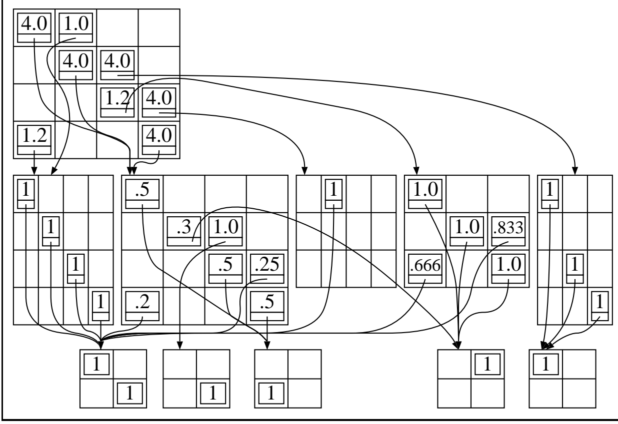


Figure 9. CMD for the running GSPN example

a CMD is a special case of an MD. For numerical solution, we must be able to access a specific column of $\mathbf{R}$; to do so, we use a recursive function *GetColumn*, similar to the function *Enc*. Since each CMD node may have multiple incoming pointers, a CMD node may receive multiple requests to compute the same (partial) column. To avoid duplicate computation, each node saves the last partial column it computed in a *cache*. Each cache "hit" may result in at most $K-1$ floating-point multiplications saved per column entry. See [7] for more details about obtaining a column and using the cache, in particular how to order the column accesses to better utilize the cache.

6. Experimental Results

A prototype of our technique is implemented in the tool SMART [5]. All reported results are on a 933 Mhz Pentium III machine running Linux. We test our approach on several models, including our running example (denoted "Simple") with N initial tokens in places p_{a1}, p_{b1}, and p_{c1}. We modified the running example to contain marking-dependent arc cardinalities: the input and output arcs of transition t_{ab} have cardinality $\min(\#p_{a3}, \#p_{b1})$, the arcs of t_{bc} have cardinality $\min(\#p_{b2}, \#p_{c3})$, and the arcs of t_{ca} have cardinality $\min(\#p_{c1}, \#p_{a2})$. This change allows for multiple synchronizations to occur with a single firing of transitions t_{ab}, t_{bc} and t_{ca}. The modified version of "Simple" is denoted "Var. Arc". Note that these models have the same reachable states and the same number of arcs in their reachability graphs.

We also examine models from the literature: a kanban model and a flexible manufacturing system (FMS) model. The kanban model, presented in [8], is partitioned into four submodels [6, 7, 8]. We denote this model by "Kan-timed" if the synchronizing transitions are timed, and by "Kan-imm" if the synchronizing transitions are immediate. We use the FMS model presented in [9] (with flushing arcs, marking-dependent rates, and immediate transitions) and partition the places as in [6]. Both of these models have an input parameter N, which specifies the initial number of tokens in certain places. Note that, using the MD technique described in [7], model "Kan-imm" requires a Kronecker implementation that can handle immediate synchronizing transitions [8], and models "Var. Arc" and "FMS" cannot be solved at all, since they contain transitions that cannot be expressed as ordinary Kronecker products.

First, we look at how our choice of partition affects our CMD. In Table 1, we consider three different partitions for our running example, and look at the number of nodes per level in our CMD and the memory required for the resulting CMD. When reporting memory usage, we use the suffix "b" to mean bytes, "kb" to mean 1024 bytes, and "mb" to mean 1024^2 bytes. The partition "ABC" corresponds to the one depicted in Figure 1, and gives us a 3-level CMD. The partition "Finest" assigns each place to its own partition: $\mathcal{P}_{12} = \{Pa_1\}, \mathcal{P}_{11} = \{Pa_2\}$, etc. Note that we get a 12-level CMD with a fairly large number of nodes, but the matrix for each node is quite small, so the overall memory requirements are not too large. The partition "Bad" uses $\mathcal{P}_k = \{Pa_k, Pb_k, Pc_k\}$ and illustrates how a bad choice of partition can result in a CMD with many large nodes.

In Table 2, we show the sizes of the state spaces and reachability graphs, the number of MD nodes for each level in the CMD constructed for the transition rate matrix $\mathbf{R}$, and the memory required for the CMD for numerical solution (i.e., including the MD representation of the row sums and the cache memory). For comparison, we also show the memory required for sparse storage of $\mathbf{R}$ (the dashes indicate cases where main memory was exceeded). Note that the matrix for each node is stored in a sparse format, so the memory required is proportional to the total number of matrix elements, not to the number of nodes. Note the CMDs for the "Var. Arc" model require significantly more nodes than the corresponding "Simple" model. This is because the "Var. Arc" model has a more complex structure, and some pointers that went to the same node in the CMD for "Simple" now must go to different nodes in the CMD for "Var. Arc". Similarly, the structure for "Kan-imm" is more complex than "Kan-timed". Also, note that while the size of each node may increase with N, the number of nodes for the kanban models is fixed except at level 2. Thus, for this model we can efficiently encode very large matrices.

In Table 3, we show the CPU and memory requirements of Build2 for different values of *freq*. For comparison, we show the CPU time required when explicit storage is used; the difference between this value and the time required for Build2 is essentially the overhead of constructing the CMD. In the CPU columns, "s" stands for seconds, "m" for min-

Partition	Number of CMD Nodes per level												CMD
	12	11	10	9	8	7	6	5	4	3	2	1	Mem
A B C										1	25	23	71.5kb
Finest	1	19	25	25	25	211	217	26	26	25	1	1	84.1kb
Bad									1	1,740	403	1	2.5mb

Table 1. Effects of different partitions, Simple model, $N = 6$

GSPN Model	N	#States	#Arcs	Nodes/level				CMD Mem	Explicit Mem
				4	3	2	1		
Simple	1	30	55		1	5	5	1.7kb	604b
	12	7,503,405	51,820,626		1	49	44	719.8kb	452.6mb
	16	56,137,077	412,022,676		1	65	58	2.0mb	—
Var. Arc	12	7,503,405	51,820,626		1	247	308	2.6mb	452.6mb
	16	56,137,077	412,022,676		1	425	523	9.1mb	—
Kan-timed	5	2,546,432	24,460,016	1	3	22	3	21.2kb	206.0mb
	8	133,865,325	1,507,898,700	1	3	34	3	62.8kb	—
Kan-imm	6	4,785,536	47,943,168	1	8	63	8	56.6kb	402.3mb
	9	106,153,300	1,177,449,900	1	8	93	8	141.9kb	—
FMS	8	4,459,455	38,533,968	1	133	186	38	538.0kb	328.0mb
	11	54,682,992	518,030,370	1	232	335	53	1.8mb	—

Table 2. Matrix and CMD sizes

utes, and "h" for hours. We see that when *freq* is zero, the CPU requirements are quite high. When *freq* is one, the CPU requirements are reasonable, and the memory requirements are usually the lowest. When *freq* is large, the MD is allowed to expand quite a bit during construction (leading to high memory requirements) before it is reduced. While the number of reductions decreases as *freq* increases, the cost of each reduction will increase if there are more nodes to reduce. In some cases (for instance, FMS), this causes an increase in the CPU time when *freq* increases.

7. Conclusion

We have presented a new definition of matrix diagrams, which are a proper subset of the matrix diagrams defined in [7]. We show that our matrix diagrams are a canonical form, and give a detailed algorithm for constructing a canonical matrix diagram for any non-negative matrix, given MDDs for the sets of rows and columns. This allows us to construct a CMD representation for a GSPN with a partitioned set of places in a general way, unlike our previous work which was tied to a Kronecker representation. Despite the differences between MDs and CMDs, the efficient solution techniques presented earlier for MDs can still be applied to CMDs. Using an MDD representation for $\mathcal{S}$ and a CMD representation for $\mathbf{R}$, the only memory bottleneck for GSPN solution is the probability vector $\boldsymbol{\pi}$. Thus, GSPNs with extremely large state spaces can be analyzed.

The new canonical form of matrix diagrams expands their applicability, since they apply to matrices in general, and opens up new directions of research. We are currently investigating efficient algorithms to build CMDs directly from the GSPN model specification, instead of explicitly adding each arc of the underlying CTMC.

References

[1] P. Buchholz. Numerical solution methods based on structured descriptions of Markovian models. In G. Balbo and G. Serazzi, editors, *Computer performance evaluation*, pages 251–267. Elsevier Science Publishers B.V. (North-Holland), 1991.

[2] P. Buchholz, G. Ciardo, S. Donatelli, and P. Kemper. Complexity of memory-efficient Kronecker operations with applications to the solution of Markov models. *INFORMS J. Comp.*, 12(3):203–222, SUMMER 2000.

[3] P. Buchholz and P. 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, Oct. 1995. IEEE Comp. Soc. Press.

[4] G. Ciardo, G. Luettgen, and R. Siminiceanu. Saturation: an efficient iteration strategy for symbolic state space generation. In T. Margaria and W. Yi, editors, *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Lecture Notes in Computer Science 2031, pages 328–342, Genova, Italy, Apr. 2001. Springer-Verlag.

GSPN Model	N	$freq = 0$ CPU	Mem	$freq = 1$ CPU	Mem	$freq = 2$ CPU	Mem	$freq = 3$ CPU	Mem	$freq = 4$ CPU	Mem	Explicit CPU
Simple	6	45.9s	94.8kb	22.7s	94.4kb	20.9s	274.0kb	21.4s	12.2mb			17.7s
	12	5.5h	842.1kb	55.6m	839.6kb	47.9m	4.1mb	—	—			41.3m
Var. Arc	6	77.6s	167.8kb	48.3s	174.5kb	45.8s	355.8kb	46.6s	13.1mb			39.4s
	12	12.1h	2.3mb	2.0h	2.3mb	1.7h	6.1mb	—	—			1.5h
Kan-timed	3	29.7s	20.6kb	20.7s	19.6kb	20.4s	67.0kb	20.5s	706.0kb	20.6s	10.9mb	16.4s
	5	42.9m	64.1kb	23.0m	61.0kb	23.1m	375.3kb	23.5m	11.9mb	—	—	20.0m
Kan-imm	3	21.2s	27.7kb	15.1s	26.4kb	14.9s	72.6kb	14.6s	619.5kb	15.1s	8.0mb	12.4s
	6	1.7h	124.8kb	53.1m	119.0kb	52.3m	747.0kb	—	—	—	—	45.4m
FMS	4	67.0s	60.2kb	61.1s	59.2kb	60.9s	89.5kb	61.2s	371.6kb	75.0s	6.0mb	56.6s
	8	5.5h	562.9kb	4.2h	559.8kb	4.2h	886.5kb	4.6h	13.1mb	—	—	3.5h

Table 3. CPU and storage requirements for CMD construction

[5] G. Ciardo and A. 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, Sept. 1996. IEEE Comp. Soc. Press.

[6] G. Ciardo and A. S. Miner. Storage alternatives for large structured state spaces. In R. Marie, B. Plateau, M. Calzarossa, and G. Rubino, editors, *Proc. 9th Int. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation*, LNCS 1245, pages 44–57, Saint Malo, France, June 1997. Springer-Verlag.

[7] G. Ciardo and A. S. Miner. A data structure for the efficient Kronecker solution of GSPNs. In P. Buchholz, editor, *Proc. 8th Int. Workshop on Petri Nets and Performance Models (PNPM'99)*, pages 22–31, Zaragoza, Spain, Sept. 1999. IEEE Comp. Soc. Press.

[8] G. Ciardo and M. 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.

[9] G. Ciardo and K. S. Trivedi. A decomposition approach for stochastic reward net models. *Performance Evaluation*, 18(1):37–59, 1993.

[10] D. D. Deavours and W. H. Sanders. An efficient disk-based tool for solving very large Markov models. In R. Marie, B. Plateau, M. Calzarossa, and G. Rubino, editors, *Proc. 9th Int. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation*, LNCS 1245, pages 58–71, Saint Malo, France, June 1997. Springer-Verlag.

[11] 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.

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

[13] P. Fernandes, B. Plateau, and W. J. Stewart. Efficient descriptor-vector multiplication in stochastic automata networks. *Journal of the ACM*, 45(3):381–414, 1998.

[14] H. Hermanns, J. Kayser, and M. Siegle. Multi terminal binary decision diagrams to represent and analyse continuous time Markov chains. In B. Plateau, W. J. Stewart, and M. Silva, editors, *Numerical Solution of Markov Chains*, pages 188–207, Zaragoza, Spain, June 1999. Prensas Universitarias de Zaragoza.

[15] P. Kemper. Numerical analysis of superposed GSPNs. *IEEE Transactions on Software Engineering*, 22(4):615–628, Sept. 1996.

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

[17] J. F. Meyer, A. Movaghar, and W. H. Sanders. Stochastic activity networks: structure, behavior, and application. In *Proc. Int. Workshop on Timed Petri Nets*, Torino, Italy, July 1985.

[18] A. S. Miner and G. Ciardo. Efficient reachability set generation and storage using decision diagrams. In H. Kleijn and S. Donatelli, editors, *Application and Theory of Petri Nets 1999 (Proc. 20th Int. Conf. on Applications and Theory of Petri Nets, Williamsburg, VA, USA)*, Lecture Notes in Computer Science 1639, pages 6–25. Springer-Verlag, June 1999.

[19] J. K. Muppala, G. Ciardo, and K. S. Trivedi. Modeling using Stochastic Reward Nets. In *Proc. 1st Int. Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'93)*, pages 367–372, San Diego, CA, USA, Jan. 1993. IEEE Comp. Soc. Press.

[20] B. 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.

[21] A. Srinivasan, T. Kam, S. Malik, and R. K. Brayton. Algorithms for discrete function manipulation. In *International Conference on CAD*, pages 92–95. IEEE Computer Society, 1990.