Implicit GSPN reachability set generation using decision diagrams

Andrew S. Miner
Department of Computer Science, Iowa State University
E-mail: asminer@iastate.edu

Abstract

Implicit techniques for representing and generating the reachability set of a high-level model have become quite efficient. However, such techniques are usually restricted to models whose events have equal priority. Models containing events with differing classes of priority or complex priority structure, in particular models with immediate events, have thus been required to use less-efficient explicit *reachability set generation techniques. In this paper, we present an efficient implicit technique, based on multi-valued decision diagram representations for sets of states and matrix diagram representations for next-state functions, that can handle models with complex priority structure. We adapt an efficient Kronecker-based reachability set generation algorithm to work with matrix diagrams. If the model contains immediate events, the vanishing states can be eliminated either during generation, by manipulating the matrix diagram, or after generation, by manipulating the multi-valued decision diagram. We apply both techniques to several models and give detailed experimental results.*

1. Introduction

The generation of the reachability set for a discrete-state model such as a stochastic Petri net (SPN), generalized stochastic Petri net (GSPN) or queuing network, is a necessary step in many types of studies. Formal verification techniques, such as model checking, may require generation of the reachability set to verify that the reachable states all satisfy some desirable property, such as the absence of a deadlock. Performance evaluation, based on Markov chain analysis, often uses the reachability set of a high-level model while constructing its underlying Markov chain and while computing reward measures once the Markov chain has been analyzed. In either case, generation of the reachability set is often difficult in practice due to the state explosion problem: the reachability set can be extremely large, even for a "small" model.

Traditional approaches to generate the reachability set $\mathcal{S}$ are *explicit*: reachable states are examined and generated one at a time, using an efficient data structure to store the discovered states (for example, [9, 15, 20]). Determination of the states that can be reached from the current state is done by examining the high-level model. This allows for maximum flexibility: the behavior of events can depend on the system state in complex ways.

Explicit techniques have been developed that exploit structured high-level representations, such as superposed GSPNs and stochastic automata networks. For these systems, the *next-state* function $\mathcal{N}$, which specifies the states that can be reached from the current state in a single step, can be represented in terms of Kronecker products and sums of several small matrices [7]. Explicit data structures, based on the Kronecker representation, can be used to describe the reachability set efficiently [23]. Given current technology, explicit techniques are limited to about 10^8 states at best.

Recent advancements in reachability set generation based instead on *implicit* techniques have been quite promising. Unlike explicit techniques, implicit techniques do not necessarily require space or time proportional to the size of the reachability set. This allows implicit techniques to handle reachability set sizes of 10^{20} states or more. The use of *decision diagrams*, binary decision diagrams (BDDs) [6] in particular, has been embraced by many researchers following [8]. Multi-terminal BDDs have also been used for various formalisms; these are used to store both the reachability set and the underlying Markov chain [21]. Reachability set generation for *safe* Petri nets using BDDs has been investigated in [27] and later works, where the reachability set and the next-state function are represented implicitly using BDDs. The authors suggest that non-safe nets can be handled by encoding the integer number of tokens in a place using several binary variables, although this significantly complicates the next-state function.

Alternate approaches to handle non-safe Petri nets have been developed [12, 26]. These implicit approaches use Kronecker representations for the next-state function, and multi-valued decision diagrams (MDDs) [22] to represent the reachability set, allowing state variables to be bounded integers. To use the Kronecker representation, a model must

be partitioned so that each event's contribution to the next-state function can be expressed as the cross-product of "local" functions. Separation of the events according to their affected submodels allows for an efficient iteration strategy called *saturation* [12], which can dramatically reduce the time and memory required to generate the reachability set.

A substantial limitation of previous implicit techniques for reachability set generation for Petri nets is that all transitions are assumed to have equal priority. This implies that all transitions must be timed, since immediate transitions have priority over timed transitions. Immediate transitions can be handled only if they are completely *local* to a submodel: local immediate transitions are removed and the submodel is re-defined appropriately. The focus and contribution of this paper is an implicit technique that can handle transitions with complex priority structure, and thus can handle immediate transitions in general.

The remainder of the paper is organized as follows. Additional background information about MDDs is given in Section 2. In Section 3 we propose an alternate representation of the next-state functions, based on *matrix diagrams* [16] instead of Kronecker representations or BDDs. We then show in Section 4 how the next-state functions can be constructed using matrix diagrams when the model contains events with different priorities, including immediate events. We show in Section 5 how the reachability set can be efficiently generated, once we have a matrix diagram representation for the next-state function. In Section 6 we describe two methods for eliminating vanishing states to obtain the tangible states only: elimination "on-the-fly" and elimination after generation. Experimental results, based on our implementation in SMART [14], are given in Section 7. Finally, Section 8 concludes the work.

2. Background and related work

The reachability set is formally defined as follows. Given a finite set of possible states $\hat{\mathcal{S}}$, an initial state $s \in \hat{\mathcal{S}}$ (or a set of initial states), and a next-state function $\mathcal{N} : \hat{\mathcal{S}} \rightarrow 2^{\hat{\mathcal{S}}}$ which describes the (possibly empty) set of states that may be reached from a given state in a single step, the reachability set $\mathcal{S}$ is the smallest set containing the initial state(s) that is closed over $\mathcal{N}$. A common technique to determine $\mathcal{S}$ is to compute the sequence $\mathcal{S}^n = \mathcal{S}^{n-1} \cup \mathcal{N}(\mathcal{S}^{n-1})$, where $\mathcal{S}^n$ is the set of states reachable from the initial state in n steps or fewer, with $\mathcal{S}^0 = \{s\}$. Since the set $\hat{\mathcal{S}}$ is finite and $\mathcal{S}^n \subseteq \mathcal{S}^{n+1} \subseteq \hat{\mathcal{S}}$ for all $n \in \mathbb{N}$, the sequence will eventually converge with $\lim_{n\rightarrow\infty} \mathcal{S}^n = \mathcal{S}$. Note that for this paper, we consider finite reachability sets only.

The practical goal of any reachability set generation algorithm is to efficiently represent $\hat{\mathcal{S}}$ and $\mathcal{N}$, while allowing for efficient computation of $\mathcal{S}$. In the case of SPNs, the set $\hat{\mathcal{S}}$ can be implicitly specified by determining (or placing) an upper bound on the number of tokens allowed in each place. Another approach is to decompose the model into K *submodels*, for some $K \in \mathbb{N}$. This is done for SPNs by partitioning the set of places $\mathcal{P}$. The finest possible decomposition is obtained when each submodel consists of a single place, $K = |\mathcal{P}|$; we can also group places together so that submodels contain multiple places. Given a decomposition, the set $\hat{\mathcal{S}}$ can be written as $\hat{\mathcal{S}} = \mathcal{S}_K \times \cdots \times \mathcal{S}_1$, where $\mathcal{S}_k$ is the set of possible substates for submodel k. A model state can thus be expressed as a $K-$tuple $(s_K, \ldots, s_1)$ with $s_k \in \mathcal{S}_k$. The sets $\mathcal{S}_k$, often referred to as the "local" state spaces, are required to be finite, and in this paper we require that the sets $\mathcal{S}_k$ can be generated *a priori* in isolation (it may be necessary to add implicit places to do so).

While the set $\mathcal{S}$ is assumed to be finite, it is often extremely large. One way to combat this problem is through the use of an efficient data structure. In particular, a set of states can be represented using a multi-valued decision diagram (MDD) [22]. If the model is composed of K submodels, then an MDD representing a set of states is a directed, acyclic graph consisting of *terminal* nodes zero and one, and *non-terminal* nodes, each corresponding to a submodel. We say a non-terminal node is a *level-k* node if it corresponds to submodel k, and we say that terminal nodes are level-0 nodes. A level-k non-terminal node contains $|\mathcal{S}_k|$ downward pointers, one for each substate in $\mathcal{S}_k$. A state is contained in the set represented by the MDD if and only if the path from the root node, following the pointers corresponding to the appropriate substates, reaches terminal node one. Non-terminal nodes are said to be *duplicates* if they are both level-k nodes and their corresponding pointers are equal. A non-terminal node is said to be *redundant* if all of its pointers are to the same node. A *quasi-reduced, ordered* MDD (QROMDD) does not contain duplicate nodes and requires all pointers from a level-k node to lead either to a level-$(k-1)$ node or to terminal node zero. A *reduced, ordered* MDD (ROMDD) does not contain duplicate nodes or redundant nodes, and requires all pointers from a level-k node to lead to a level less than k. Thus, ROMDDs may require fewer nodes than QROMDDs, but require more complex algorithms for manipulation (because we must handle cases where levels are "skipped"). For consistency, we follow the convention of [12] and assume in the remainder of the paper that all MDDs are QROMDDs; however, the algorithms we present could be adapted for use with ROMDDs. An example MDD and the set of states it encodes is shown in Figure 1.

For traditional reachability set generation, the function $\mathcal{N}$ is specified by the high-level model. Many types of formalisms, including SPNs, specify $\mathcal{N}$ in terms of distinct *events*:

$$\mathcal{N}(s) = \bigcup_{e \in \mathcal{E}} \mathcal{N}^e(s) \tag{1}$$

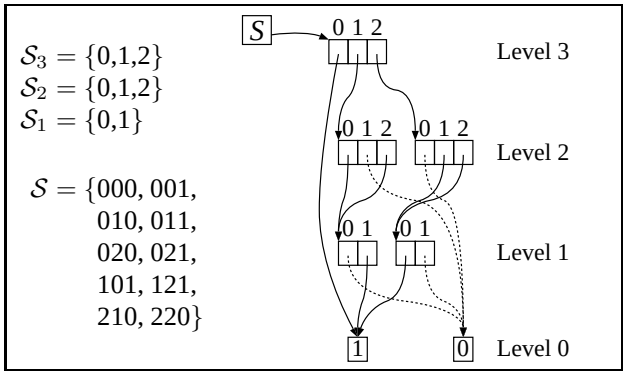


Figure 1. An MDD and its represented states

where $\mathcal{E}$ is a finite set of events. In the case of SPNs, events correspond to transitions. For implicit techniques, it is usually assumed that each $\mathcal{N}^e$ function can be expressed as the cross-product of local functions [12, 26]:

$$\mathcal{N}^e(s_K, \ldots, s_1) = \mathcal{N}^e_K(s_K) \times \cdots \times \mathcal{N}^e_1(s_1) \qquad (2)$$

where $\mathcal{N}^e_k : \mathcal{S}_k \rightarrow 2^{\mathcal{S}_k}$. If this property does not hold for some event, we can either re-partition the model so that it does hold, or split the event into sub-events such that the property holds for the sub-events. The function $\mathcal{N}$ can thus be represented by tables for each of the $\mathcal{N}^e_k$ functions. Note that this is equivalent to using a Kronecker representation for the next-state function [23]: the tables used to represent the $\mathcal{N}^e_k$ functions are the same as the small matrices used in a Kronecker representation. When the local state spaces $\mathcal{S}_k$ can be generated *a priori*, the table for each $\mathcal{N}^e_k$ function can also be constructed *a priori*, by examining the effect of event e on each local state in $\mathcal{S}_k$, considering submodel k only.

Previous work has already addressed the issue of events with priority when a Kronecker representation is used for the underlying stochastic process: [17] deals with models containing immediate synchronizing events; [19] deals with multiple levels of priorities (including immediate events). However, the focus of this earlier work is on the representation of the underlying process and on numerical solution; it is unclear if the results in [17] and [19] can be directly applied to implicit reachability set generation techniques. It is however interesting to note that the basic idea to deal with priorities is the same in this work and in both [17] and [19]: a matrix "mask" is used to restrict events with lower priority.

3. Representing the next-state function with matrix diagrams

Matrix diagrams were initially introduced as an alternative to Kronecker representations for the transition rate matrix, and have since been modified to obtain a canonical representation [25]. In this paper, we require *boolean* matrices instead of real-valued matrices. This allows for some simplifications to the structure; thus, we present here a simplified version of the canonical matrix diagrams presented in [25].

A *matrix diagram* is similar to an MDD, except the non-terminal nodes are matrices of pointers instead of vectors of pointers. Matrix diagrams consist of two level-0 terminal nodes, zero and one, and level-k non-terminal nodes, for $k \in \{1, \ldots, K\}$. Each level-k non-terminal node contains a $|\mathcal{S}_k| \times |\mathcal{S}_k|$ matrix of pointers to other nodes. A matrix diagram represents a boolean matrix of size $|\hat{\mathcal{S}}| \times |\hat{\mathcal{S}}|$, which can encode a next-state function: the non-zero entries in a given row correspond to the states that can be reached from that row. We denote a next-state function in script, as $\mathcal{N}$, and its corresponding matrix diagram representation in boldface, as $\mathbf{N}$. In matrix diagram $\mathbf{N}$, state $(j_K, \ldots, j_1)$ is contained in $\mathcal{N}(i_K, \ldots, i_1)$ if the path from the root node, following pointer $[i_k, j_k]$ from level k, reaches terminal node one.

We extend the MDD concepts of ordering and reducing nodes to matrix diagrams as follows. Two matrix diagram nodes are said to be duplicates if they are both level-k nodes and their corresponding pointers are equal (just like MDD nodes). We say that a matrix diagram node is an *identity node* if its matrix contains equal pointers along the diagonal and all off-diagonal pointers are to terminal node zero. Identity nodes capture an important characteristic of a model: independence of an event from a given submodel. The matrix diagram representing the next-state function for an event will contain identity nodes at level k if the event is independent of submodel k, because the event does not change the local state for submodel k. This type of behavior naturally occurs in a Petri net when a transition is not connected to any place within submodel k.

We say that a matrix diagram is *quasi-reduced* and *ordered* if it does not contain duplicate nodes and if all pointers from a level-k node lead either to a level-$(k-1)$ node or to terminal node zero. We say that a matrix diagram is *reduced* and *ordered* if it does not contain duplicate nodes or identity nodes, and if all pointers from a level-k node lead to a level less than k.

We note that a quasi-reduced, ordered matrix diagram is similar in structure to a QROMDD with twice as many levels: each level in the matrix diagram has a corresponding "row" and "column" level in the MDD. Indeed, this is the structure often used by MTBDDs to represent a matrix

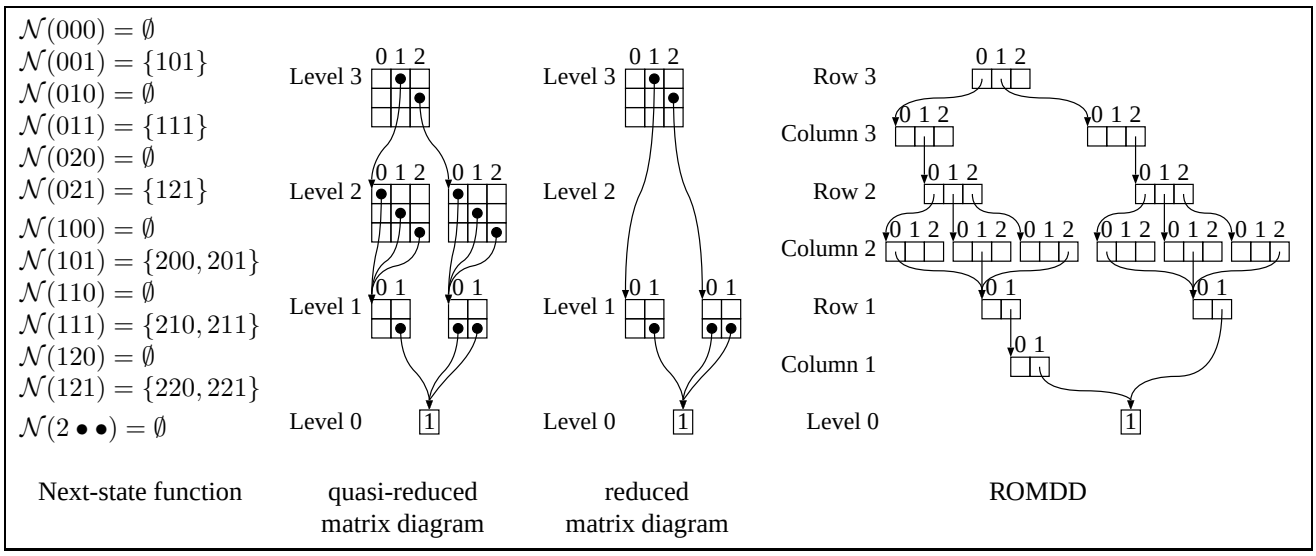


Figure 2. Representing a next-state function with a matrix diagrams and ROMDDs

[21]. However, the same is not true for the reduced case, due to the fundamental difference between the reduction rules: a skipped level in a ROMDD corresponds to a redundant node, while a skipped level in a reduced, ordered matrix diagram corresponds to an identity node. Thus, when an ROMDD is used to represent a boolean matrix, terminal node zero corresponds to a matrix of zeroes, and terminal node one corresponds to a matrix of ones. In contrast, when a reduced, ordered matrix diagram is used, terminal node zero corresponds to a matrix of zeroes, and terminal node one corresponds to an identity matrix.

An example next-state function $\mathcal{N}$ is shown in Figure 2 to illustrate the differences between quasi-reduced matrix diagram, reduced matrix diagram, and ROMDD representations. For clarity, pointers to terminal node zero are omitted. Note that, in this example, the next-state function is not affected by, and does not change, the local state of submodel 2. As a result, the quasi-reduced matrix diagram contains identity nodes at level 2, and the reduced matrix diagram pointers skip level 2. For the remainder of the paper, we assume that all matrix diagrams are ordered and reduced.

3.1. Constructing next-state functions

If the next-state function is represented by a function for each event (as in Equation 1), and the event functions are expressible as the cross-product of local functions (as in Equation 2), then the algorithm to construct a matrix diagram $\mathbf{N}^e$ for the next-state function $\mathcal{N}^e$ is straightforward. The local functions $\mathcal{N}_K^e, \ldots, \mathcal{N}_1^e$ are stored as tables and can be constructed directly from the high-level model. Note that if event e has no effect on submodel k then the function $\mathcal{N}_k^e$ will be the identity function; this can be detected by examining the structure of the high-level model. Once we have obtained the local functions $\mathcal{N}_K^e, \ldots, \mathcal{N}_1^e$, we can construct a matrix diagram using algorithm BuildNext, shown in Figure 3. The matrix diagram node at level k is based on $\mathcal{N}_k^e$. If $\mathcal{N}_k^e$ is the identity function, then the matrix diagram $\mathbf{N}^e$ will skip level k, since the node would be an identity node. Otherwise, row i_k of the matrix diagram node at level k will contain a pointer in column j_k if $j_k \in \mathcal{N}_k^e(i_k)$. All pointers for a given node will be equal, and will point to the next non-terminal node or terminal node one.

```
BuildNext(N_K^e, ..., N_1^e)
 1: N^e ← one;
 2: for each k ∈ {1, ..., K} do
 3:    if N_k^e is not identity then
 4:       C ← New level-k node of all zeroes;
 5:       for each i ∈ S_k, j ∈ N_k^e(i) do
 6:          C[i, j] ← N^e;
 7:       end for
 8:       N^e ← C;
 9:    end if
10: end for
11: return N^e;
```

Figure 3. Constructing a next-state function for a given event

```
ElementWise(Md A, op, Md B)
 1: if op = + then                    • Terminal conditions for +
 2:   if A = B ∨ A = zero then
 3:     return B;
 4:   else if B = zero then
 5:     return A;
 6:   end if
 7: else if op = − then               • Terminal conditions for -
 8:   if A = B ∨ A = zero then
 9:     return zero;
10:   else if B = zero then
11:     return A;
12:   end if
13: else if op = · then              • Terminal conditions for ·
14:   if A = B then
15:     return A;
16:   else if A = zero ∨ B = zero then
17:     return zero;
18:   end if
19: end if
20: if cache contains(A, op, B, C) then
21:   return C;
22: end if
23: kA ←Level(A);   kB ←Level(B);
24: k ← MAX(kA, kB);
25: C ← New level-k node;
26: if kA > kB then
27:   ∀i, C[i, i] ← ElementWise(A[i, i], op, B);
28:   ∀i ≠ j, C[i, j] ← ElementWise(A[i, j], op, zero);
29: else if kB > kA then
30:   ∀i, C[i, i] ← ElementWise(A, op, B[i, i]);
31:   ∀i ≠ j, C[i, j] ← ElementWise(zero, op, B[i, j]);
32: else
33:   ∀i, j, C[i, j] ← ElementWise(A[i, j], op, B[i, j]);
34: end if
35: Reduce C;
36: Add cache entry (A, op, B, C);
37: return C;
```

Figure 4. Element-wise operations on matrix diagrams

3.2. Manipulating matrix diagrams

The main strength of matrix diagrams over Kronecker-based representations is their ability to be manipulated. Binary operations on MDDs and matrix diagrams have complexity that depends on the number of nodes in the graphs, rather than the number of items encoded by the graph. Thus, the compactness in size of MDDs and matrix diagrams leads not only to obvious memory savings, but also to computational savings.

A recursive algorithm for element-wise matrix operations is given in Figure 4. Useful element-wise operations for this work are element-wise addition, subtraction, and multiplication, which are used to compute the union, difference, and intersection of two next-state functions, respectively. For instance, the next-state function $\mathcal{C}(s_K, \ldots, s_1) = \mathcal{A}(s_K, \ldots, s_1) \cup \mathcal{B}(s_K, \ldots, s_1)$ can be constructed from matrix diagrams $\mathbf{A}$ and $\mathbf{B}$ by calling $\mathbf{C} = \mathsf{ElementWise}(\mathbf{A}, +, \mathbf{B})$. The recursion terminates when one of the operator-specific terminal conditions holds. Terminal cases for element-wise addition are

$$\begin{aligned} \mathsf{ElementWise}(\mathbf{A}, +, \mathbf{A}) &= \mathbf{A} \\ \mathsf{ElementWise}(\mathbf{A}, +, \mathbf{0}) &= \mathbf{A} \\ \mathsf{ElementWise}(\mathbf{0}, +, \mathbf{A}) &= \mathbf{A} \end{aligned}$$

which are handled in lines 2-6. Terminal cases for element-wise subtraction are

$$\begin{aligned} \mathsf{ElementWise}(\mathbf{A}, -, \mathbf{A}) &= \mathbf{0} \\ \mathsf{ElementWise}(\mathbf{A}, -, \mathbf{0}) &= \mathbf{A} \\ \mathsf{ElementWise}(\mathbf{0}, -, \mathbf{A}) &= \mathbf{0} \end{aligned}$$

which are handled in lines 8-12. Terminal cases for element-wise multiplication are

$$\begin{aligned} \mathsf{ElementWise}(\mathbf{A}, \cdot, \mathbf{A}) &= \mathbf{A} \\ \mathsf{ElementWise}(\mathbf{A}, \cdot, \mathbf{0}) &= \mathbf{0} \\ \mathsf{ElementWise}(\mathbf{0}, \cdot, \mathbf{A}) &= \mathbf{0} \end{aligned}$$

which are handled in lines 14-18. Since matrix diagram nodes can have multiple incoming pointers, it is possible that several of the recursive calls to ElementWise have the same parameters. To avoid duplication of work, MDD and matrix diagram manipulation algorithms often make use of a *computation table* or *cache* to save previously-computed results [6]. Lines 20-21 check that the current computation has not been computed already, and line 36 adds the result of the current computation to the table.

Since ElementWise operates on reduced matrix diagrams, we must handle skipped levels, which are treated as identity nodes; this is done in lines 26-31: lines 27 and 30 handle the equal diagonal pointers and lines 28 and 31 handle the non-diagonal pointers to terminal node zero. The resulting matrix diagram must be reduced; this is handled in line 35 by checking if the matrix diagram node just created is a duplicate of some other node or is an identity node. A *uniqueness table* (usually a hash table) is maintained to discover duplicates; this is a common technique used by decision diagram manipulation algorithms [5].

As with recursive BDD and MDD algorithms, the use of the computation table guarantees that the number of recursive calls generated to ElementWise is limited to one for each distinct pair A, B of matrix diagram nodes. Thus, a top-level call to ElementWise for matrix diagrams $\mathbf{A}$ and $\mathbf{B}$

```
Multiply(Md A, Md B)
 1: if (A = zero) ∨ (B = zero) then
 2:    return zero;
 3: else if (A = one) then
 4:    return B;
 5: else if (B = one) then
 6:    return A;
 7: else if Multiply Cache contains(A, B, C) then
 8:    return C;
 9: end if
10: kA ←Level(A);   kB ←Level(B);
11: k ← MAX(kA, kB);
12: C ← New level-k node;
13: if kA > kB then
14:    ∀i, j, C[i, j] ← Multiply(A[i, j], B);
15: else if kB > kA then
16:    ∀i, j, C[i, j] ← Multiply(A, B[i, j]);
17: else
18:    for each i, j, l ∈ S_k do
19:       T ← Multiply(A[i, l], B[l, j]);
20:       C[i, j] ← ElementWise(C[i, j], +, T);
21:    end for
22: end if
23: Reduce C;
24: Insert Multiply Cache entry (A, B, C);
25: return C;
```

Figure 5. Matrix diagram multiplication

will produce at most $|\mathbf{A}| \cdot |\mathbf{B}|$ recursive calls that require computation, where $|\mathbf{A}|$ is the number of nodes contained in matrix diagram $\mathbf{A}$. As shown, the computational cost for a single call to ElementWise is at most $O(|\mathcal{S}_k|^2)$; this can be reduced in practice to $O(\eta(\mathbf{C}))$, where $\eta(\mathbf{C})$ is the number of non-zero downward pointers of node $\mathbf{C}$. Assuming constant-time complexity to check for duplicates and to check the computation table, the overall complexity of ElementWise is thus $O(w \cdot |\mathbf{A}| \cdot |\mathbf{B}|)$, where the computation required for a single call is at worst $w = \max_k(|\mathcal{S}_k|^2)$.

Another important operation for this work is matrix multiplication: the composition $\mathcal{C}$ of functions $\mathcal{A}$ and $\mathcal{B}$ (i.e., $\mathcal{C}(x) = \mathcal{B}(\mathcal{A}(x))$) can be determined by multiplying matrix diagrams $\mathbf{A}$ and $\mathbf{B}$. A recursive algorithm for matrix diagram multiplication is shown in Figure 5, where Multiply$(\mathbf{A}, \mathbf{B})$ computes $\mathbf{A} \cdot \mathbf{B}$ for boolean matrices. The algorithm is based on the multiplication of two block-structured matrices. The recursion terminates if the appropriate terminal conditions hold, or if the answer has already been computed (and is present in the cache). The terminal conditions for multiplication are $\mathbf{A} \cdot \mathbf{0} = \mathbf{0} \cdot \mathbf{B} = \mathbf{0}$ (line 1), $\mathbf{I} \cdot \mathbf{B} = \mathbf{B}$ (line 3), and $\mathbf{A} \cdot \mathbf{I} = \mathbf{A}$ (line 5). Lines 13-16 of Multiply handle skipped levels. Similar to algorithm ElementWise, the number of recursive calls to Multiply that require computation is limited to $|\mathbf{A}| \cdot |\mathbf{B}|$. The cost of a single recursive call is $O(|\mathcal{S}_k|^3)$ due to the matrix-matrix multiplication in lines 18-21, multiplied by the cost of ElementWise for the addition. However, the worst-case number of distinct calls to ElementWise (in line 20) is difficult to characterize: while the number of possible values of $\mathbf{T}$ (in line 19) is equal to the number of possible distinct outcomes of Multiply, which is $|\mathbf{A}| \cdot |\mathbf{B}|$, these can be summed in different orders, leading to different intermediate results. In practice, if the matrix diagram nodes are sparse, then $\mathbf{T}$ will often be zero, so the number of non-trivial calls to ElementWise should be quite small. In particular, if $\mathbf{A}$ and $\mathbf{B}$ represent diagonal matrices, then the cost of lines 18-21 can be reduced to $O(|\mathcal{S}_k|)$, and all calls to ElementWise will be trivial. Thus, when Multiply is applied to diagonal matrices $\mathbf{A}$ and $\mathbf{B}$, its complexity is at worst $O(|\mathbf{A}| \cdot |\mathbf{B}| \cdot \max_k(|\mathcal{S}_k|))$. Note that this is essentially the same computation (and the same complexity) as MDD-based set intersection.

Another operation that we will utilize is determining which rows of a matrix are empty. In terms of a next-state function, empty rows correspond to states that do not have a next state. This information is used to determine the states that disable a given event. The algorithm to perform this operation is shown in Figure 6. Specifically, given a matrix diagram $\mathbf{A}$, EmptyRows builds a second matrix diagram that encodes a diagonal matrix, where the diagonal element for row i is one if and only if row i of the matrix encoded by $\mathbf{A}$ is empty. Like the previous algorithms, it is recursive and terminates if the appropriate terminal condition holds or if the answer has already been computed. Since the identity matrix does not contain any empty rows, and a matrix of zeroes contains all empty rows, the terminal conditions are EmptyRows$(\mathbf{I}) = \mathbf{0}$ and EmptyRows$(\mathbf{0}) = \mathbf{I}$. Thanks to the compute table, the number of recursive calls generated to EmptyRows that require computation is $|\mathbf{A}|$. The computational cost of a single call is $O(\eta(\mathbf{A}))$, multiplied by the cost of Multiply. Note that the call to Multiply (line 14) is non-trivial only when $\mathbf{A}$ contains more than one non-terminal pointer for a given row. Since the calls to Multiply generated by EmptyRows will always be for diagonal matrices, they can be computed efficiently.

4. Handling priorities

We are now ready to handle high-level models whose events are allowed to have *priorities*. Specifically, we allow the model to specify, for pairs of events b and c, that event b always has priority over event c. Thus, whenever event b is enabled, event c effectively becomes disabled. In addition to providing an increase in modeling power, priorities are necessary to handle *immediate* events (events which occur in zero time), since immediate events always have priority over timed events [1].

```
EmptyRows(Md A)
 1: if (A =zero) then
 2:    return one;
 3: else if (A =one) then
 4:    return zero;
 5: else if RowsCache contains(A, C) then
 6:    return C;
 7: end if
 8: k ←Level(A);
 9: C ← New level-k node;
10: for each i ∈ S_k do
11:    C[i, i] ← one;
12:    for each j ∈ S_k do
13:       T ← EmptyRows(A[i, j]);
14:       C[i, i] ← Multiply(C[i, i], T);
15:    end for
16: end for
17: Reduce C;
18: Insert RowsCache entry (A, C);
19: return C;
```

Figure 6. Determining empty rows of a matrix diagram

```
BuildNextPriority(event e)
 1: if we have computed N^e then
 2:    return;
 3: else if we are already computing N^e then
 4:    Error, cyclic dependency;
 5: end if
 6: N^e ← BuildNext(N^e_K, ..., N^e_1);
 7: E ← one;            • States enabling e, initially all
 8: for each event e' with higher priority than e do
 9:    BuildNextPriority(e');
10:    E ← Multiply(E, EmptyRows(N^e'));
11: end for
12: N^e ← Multiply(E, N^e);
```

```
BuildNextPriorityClass
 1: E ← one;
 2: F ← one;
 3: for each priority p, from high to low do
 4:    for each event e with priority p do
 5:       N^e ← BuildNext(N^e_K, ..., N^e_1);
 6:       N^e ← Multiply(F, N^e);
 7:       E ← Multiply(E, EmptyRows(N^e));
 8:    end for
 9:    F ← E;
10: end for
```

Figure 7. Constructing next-state functions with event priorities

To construct matrix diagrams for the next-state functions for each event when events have a priority structure, we first build the next-state functions in isolation, ignoring the priorities, and then modify the functions appropriately. If our model contains only timed events with equal priority, the next-state functions computed using BuildNext (shown in Figure 3) can be used without modification. If event b has priority over event c, we must modify function $\mathcal{N}^c$ so that any state s that enables event b has no next-states: $\mathcal{N}^c(s) = \emptyset$. This can be done by manipulating the matrix diagrams $\mathbf{N}^b$ and $\mathbf{N}^c$ as follows.

1. Determine the states that do not enable event b by computing EmptyRows($\mathbf{N}^b$).
2. Restrict event c to these states only, using Multiply.

Thus, Multiply(EmptyRows($\mathbf{N}^b$),$\mathbf{N}^c$) constructs the proper next-state function for event c, taking into account that event c is disabled whenever event b is enabled.

Complex priority structures can also be handled with the same technique, if care is taken to perform the steps in the correct order. For instance, suppose event a has priority over event b, and event b has priority over event c, and event a has the same priority as event c (i.e., a does not have priority over c and c does not have priority over a). Then b is disabled whenever a is enabled, and c is disabled whenever b is enabled (and a is disabled). Steps (1) and (2) above will correctly construct the next-state function for c, provided $\mathbf{N}^b$ has already been adjusted. Thus, adjustment of the next-state functions to incorporate priority should proceed from high priority events to low priority events.

Given any acyclic priority structure [11], the next-state function for each event can be determined using the algorithm BuildNextPriority in Figure 7. Priority cycles are not allowed (e.g., if event b has priority over c, then event c cannot have priority over b). Existence of a cycle will cause BuildNextPriority(e) to eventually call itself with the same parameter, which can be detected. If a priority class structure is used instead, in which each event is assigned an integer priority [10], then the next-state function for each event can be determined using the simpler algorithm BuildNextPriorityClass, also shown in Figure 7. Note that integer priorities are insufficient to capture the example above: it is impossible to assign integer priorities to events a, b, and c such that a has priority over b, b has priority over c, and a and c have equal priority.

An example illustrating the construction of the next-state functions for events a, b and c, where a has priority over b and b has priority over c, is shown in Figure 8. Event a is enabled in states $(1 \bullet 0)$ and $(2 \bullet 0)$. To determine the next-state function for b, we compute $\mathbf{E}^a$ = EmptyRows($\mathbf{N}^a$) and multiply it by $\mathbf{N}^b$. Note that, before adjustment (matrix diagram $\mathbf{N}^b$ in the figure), event b is enabled in states

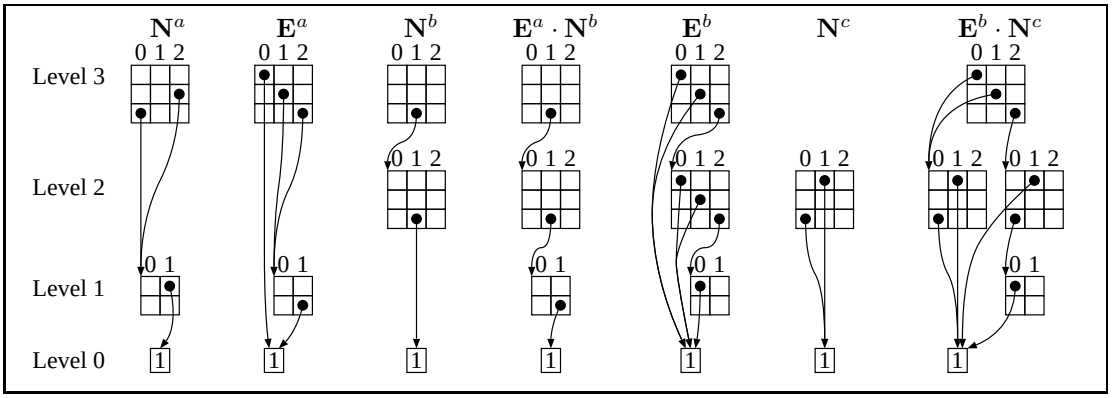


Figure 8. Example next-state construction for events a, b, c with priorities

$(22\bullet)$, but after adjusting for priorities, b is enabled in state (221) only. Event c depends only on submodel 2, and is enabled in states $(\bullet 0\bullet)$ and $(\bullet 2\bullet)$. To adjust its next-state function to include the priority relationship with b, we compute $\mathbf{E}^b = \mathsf{EmptyRows}(\mathbf{E}^a \cdot \mathbf{N}^b)$ and multiply $\mathbf{E}^b$ by $\mathbf{N}^c$. After adjustment, event c is disabled in state (221).

5. Saturation using matrix diagrams

Once the proper next-state functions have been determined, the reachability set can be generated. We must adapt one of the Kronecker-based implicit techniques [12, 26] to work using matrix diagrams instead of a Kronecker representation for the next-state function.

An extremely efficient implicit reachability set generation algorithm for asynchronous systems is that of *node saturation* [12]. The idea is to exploit the locality of events in the system: an event e may affect only a few submodels. In terms of a product-form decomposition (as in Equation 2), such an event will have identities for many of the local functions. For example, if the polling model in Figure 11(c) contained only timed transitions, then transition Ta_i would only affect queue i; assuming the model is partitioned so that each queue is a submodel, we would then have $\mathcal{N}^{Ta_i}(s_K, \ldots, s_1) = \{s_K\} \times \cdots \times \{s_{i+1}\} \times \mathcal{N}_i^{Ta_i}(s_i) \times \{s_{i-1}\} \times \cdots \times \{s_1\}$. In terms of a Kronecker representation, the matrices for all submodels except submodel i would be identity matrices; the matrix diagram representation $\mathbf{N}^{Ta_i}$ would consist of a single level-i node.

Two key observations lead to the saturation algorithm. First, if submodel k is the "first" submodel affected by event e (i.e., the top node of the matrix diagram for event e is a level-k node), then submodels K through $k+1$ cannot disable event e. Thus, during reachability set generation, we can "apply" event e to any level-k MDD node, since level k is the first level affected by event e. The second observation is that whenever a level-k node is created, we can *saturate* the node by repeatedly considering events whose first level is k. The Kronecker version of saturation, presented in [12], sorts the events by first level, so that saturation of a level-k node can cycle through all events whose first level is k. The matrix-diagram version of saturation, shown in Figure 9, instead maintains a single matrix diagram $\mathbf{N}_k$ for each level k, which can be considered a "macro-event" that combines all events whose first level is k. Thus, instead of maintaining several different events, as is done with a Kronecker representation, with matrix diagrams we represent K collections of events. This eliminates the need to cycle through several events at each level k.

For saturation to work efficiently, we *must* be able to collect state-to-state transitions into the appropriate $\mathbf{N}_k$ matrix diagram, according to their first affected submodels. Otherwise, the saturation algorithm reduces to the "traditional" symbolic generation algorithm [26, 27] which considers one event at a time from the top level. One way to achieve this with matrix diagrams is to maintain $\mathbf{N}_k$ for each k during construction of the next-state functions. When the matrix diagram is constructed for the next-state function for event e, we can easily determine the first level affected by e from the top node of the matrix diagram. The matrix diagram is then added (using algorithm ElementWise) to the overall matrix diagram for the appropriate level. However, this method may not obtain the best possible splitting of the state-to-state transitions.

For instance, after adjusting event c to account for priorities in Figure 8, we obtain matrix diagram $\mathbf{N}$ in Figure 10. Since the top node is a level-3 node, we would add it to the overall matrix diagram for level 3. However, by inspecting matrix diagram $\mathbf{N}$ we see that it is "almost" independent of submodel 3: the level-3 node is "almost" an identity node. We can remove the state-to-state transitions that do not depend on submodel 3 (i.e., the transitions that occur regardless of the state of submodel 3 and do not change submodel 3). This gives us matrix diagram $\mathbf{N}_3$, which contains the

```
Saturate(level k, MDD n)
 1: L ← {i : n[i] ≠ zero ∧ ∃N_k[i, j] ≠ zero};
 2: while L ≠ ∅ do
 3:    i ← SelectElement(L);
 4:    L ← L \ {i};
 5:    for each j : N_k[i, j] ≠ zero do
 6:       f ← RecFire(N_k[i, j], n[i]);
 7:       if f ≠ zero then
 8:          u ← Union(f, n[j]);
 9:          if u ≠ n[j] then
10:             n[j] ← u;
11:             if ∃r, N_k[j, r] ≠ zero then
12:                L ← L ∪ {j};
13:             end if
14:          end if
15:       end if
16:    end for
17: end while
```

```
Extract(Md A; out: Md B, Md C)
1: k ← Top(A);
2: C ← A[0, 0];
3: for each i ∈ {1, ..., |S_k| − 1} do
4:    C ← ElementWise(C, ·, A[i, i]);
5: end for
6: B ← ElementWise(A, −, C);
```

```
RecFire(Md N, MDD n)
 1: if N = one then
 2:    return n;
 3: else if N = zero ∨ n = zero then
 4:    return zero;
 5: else if Cache contains (N, n, s) then
 6:    return s;
 7: end if
 8: k ← MAX(Top(N), Top(n));
 9: s ← new level-k MDD node;
10: if Top(N) < Top(n) then
11:    for each i do
12:       s[i] ← RecFire(N, n[i]);
13:    end for
14: else
15:    for each i, j : N[i, j] ≠ zero do
16:       f ← RecFire(N[i, j], n[i]);
17:       s[j] ← Union(s[j], f);
18:    end for
19: end if
20: if s ≠ [zero, ..., zero] then
21:    Saturate(k, s);
22: end if
23: Reduce s;
24: Add (N, n, s) to Cache;
25: return s;
```

Figure 9. Node saturation algorithm using matrix diagrams

transitions that depend on submodel 3, and matrix diagram $\mathbf{N}_2$, which contains the transitions that do not depend on submodel 3. Note that $\mathbf{N}$ is the sum of $\mathbf{N}_3$ and $\mathbf{N}_2$.

The use of matrix diagrams allows us to easily perform this splitting. Given a level-k matrix diagram node $\mathbf{N}$, we can extract *all* state-to-state transitions that do not depend on submodel k by finding the "largest common" diagonal element using element-wise multiplication. This is detailed in procedure Extract, shown in Figure 9, which splits a matrix diagram node $\mathbf{A}$ into $\mathbf{B} + \mathbf{C}$, where $\mathbf{C}$ represents the state-to-state transitions that do not depend on submodel k. If such a splitting is impossible (which occurs when all state-to-state transitions in matrix diagram $\mathbf{A}$ depend on the top-level submodel), then Extract will produce matrix diagrams $\mathbf{B} = \mathbf{A}$ and $\mathbf{C} = \text{zero}$.

In summary, to use saturation with matrix diagrams we first construct a single overall next-state matrix diagram $\mathbf{N}$ by summing the matrix diagrams for each event (using algorithm ElementWise). We then compute the matrix diagram $\mathbf{N}_k$ for each level k by first initializing $\mathbf{N}_K$ to $\mathbf{N}$, and then computing $\text{Extract}(\mathbf{N}_k, \mathbf{N}_k, \mathbf{N}_{k-1})$ for all k from K down to 2. These level-wise next state functions are then used in the saturation algorithm. The reachability set is generated by constructing the MDD representing the initial state(s) in a bottom-up fashion. As each node n in the MDD is created, it is saturated by calling $\text{Saturate}(k, n)$, where k is the level of node n. When the level-K node has been saturated, it will represent the reachability set.

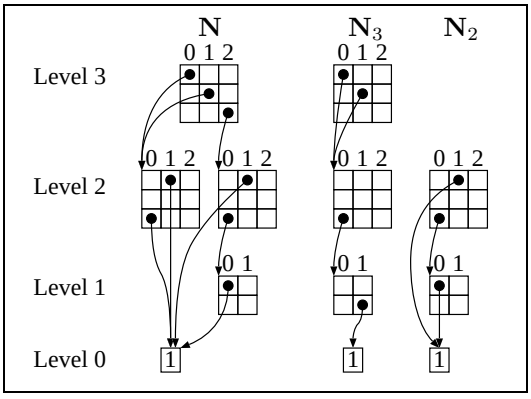


Figure 10. Splitting a next-state function

6. Eliminating vanishing states

The techniques described in the previous sections can be applied to high-level models with event priorities, including GSPNs and other formalisms with immediate events. Reachability set generation will correctly produce the set of reachable states. However, the reachability set $\mathcal{S}$ can be partitioned into the set of *vanishing* states $\mathcal{V}$, in which an immediate event is enabled, and *tangible* states $\mathcal{T}$, in which no immediate events are enabled. For many types of analysis, especially generation and solution of an underlying Markov chain, only the tangible states are of interest, and the vanishing states can be eliminated [3, 4]. As with traditional reachability set generation algorithms, elimination of vanishing states can be done either *during* generation (sometimes called "on-the-fly") or *after* generation. That is, we can either construct the set $\mathcal{T}$ directly, or we can construct the entire reachability set $\mathcal{S}$, partition it into $\mathcal{T}$ and $\mathcal{V}$, and keep only $\mathcal{T}$.

6.1. Elimination during generation

To eliminate vanishing states during generation, we must maintain only the set $\mathcal{T}$ and ensure that only the tangible reachable states are added to the set $\mathcal{T}$. This is done by using a modified next-state function $\mathcal{N}'$ that, given a tangible state, returns the set of tangible states that can be reached from the given state when a single timed event occurs, possibly via a sequence of vanishing states. If a tangible state reaches a vanishing state via a timed event, then the sequences of vanishing states are collapsed until another tangible state is reached. If the initial state is tangible, then generating the reachability set using $\mathcal{N}'$ will produce exactly the set $\mathcal{T}$. A vanishing initial state can be handled by determining all the tangible states that can be reached from the initial state via immediate events only, and using those as the initial states.

If the next-state function is represented as a matrix $\mathbf{N}$, then we can classify the rows and columns as either tangible or vanishing, and obtain the block structure

$$\mathbf{N} = \begin{bmatrix} \mathbf{N}_{\mathcal{T}} \\ \mathbf{N}_{\mathcal{V}} \end{bmatrix} = \begin{bmatrix} \mathbf{N}_{\mathcal{T}\mathcal{T}} & \mathbf{N}_{\mathcal{T}\mathcal{V}} \\ \mathbf{N}_{\mathcal{V}\mathcal{T}} & \mathbf{N}_{\mathcal{V}\mathcal{V}} \end{bmatrix}$$

where $\mathbf{N}_{\mathcal{T}\mathcal{T}}$ describes transitions from tangible states to tangible states, $\mathbf{N}_{\mathcal{T}\mathcal{V}}$ describes transitions from tangible states to vanishing states, and so on. The matrix representation $\mathbf{N}'$ is exactly

$$\mathbf{N}' = \mathbf{N}_{\mathcal{T}\mathcal{T}} + \mathbf{N}_{\mathcal{T}\mathcal{V}} \cdot \mathbf{N}^*_{\mathcal{V}\mathcal{V}} \cdot \mathbf{N}_{\mathcal{V}\mathcal{T}} \qquad (3)$$

where "*" denotes transitive and reflexive closure. Thus, reaching a tangible state from another tangible state via a single timed event is possible directly, by a transition from a tangible state to a tangible state, or via vanishing states, by a transition from a tangible state to a vanishing state, followed by zero or more transitions from a vanishing state to another vanishing state, followed by a transition from a vanishing state to a tangible state.

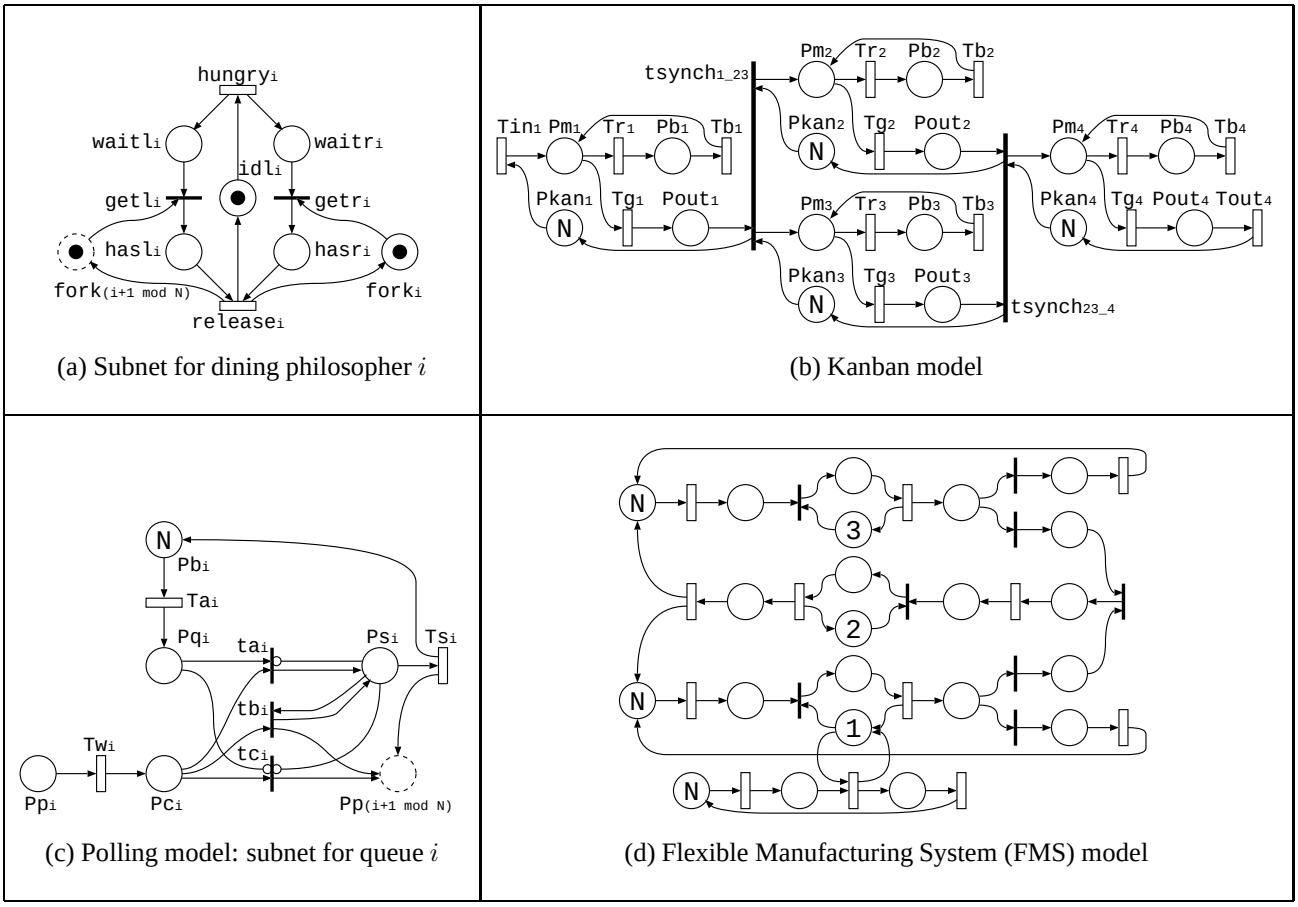
To construct the matrix diagram $\mathbf{N}'$, we use Equation 3. Given the next-state function for each event, modified to account for priorities (as described earlier), we can compute $\mathbf{N}_{\mathcal{T}}$ by summing the next-state functions for the timed events, and $\mathbf{N}_{\mathcal{V}}$ by summing the next-state functions for the immediate events. The potential tangible states can be found by computing $\mathbf{P}_{\mathcal{T}} = \mathsf{EmptyRows}(\mathbf{N}_{\mathcal{V}})$, and the potential vanishing states can be found by computing $\mathbf{P}_{\mathcal{V}} = \mathsf{EmptyRows}(\mathbf{P}_{\mathcal{T}})$. The tangible (vanishing) columns of a matrix can be selected by multiplying the matrix by $\mathbf{P}_{\mathcal{T}}$ ($\mathbf{P}_{\mathcal{V}}$). For instance, we compute $\mathbf{N}_{\mathcal{T}\mathcal{T}} = \mathsf{Multiply}(\mathbf{N}_{\mathcal{T}}, \mathbf{P}_{\mathcal{T}})$ and $\mathbf{N}_{\mathcal{T}\mathcal{V}} = \mathsf{Multiply}(\mathbf{N}_{\mathcal{T}}, \mathbf{P}_{\mathcal{V}})$. To compute the transitive and reflexive closure of $\mathbf{N}_{\mathcal{V}\mathcal{V}}$, we compute $\mathbf{N}^*_{\mathcal{V}\mathcal{V}} = (\mathbf{I} + \mathbf{N}_{\mathcal{V}\mathcal{V}})^\infty$ by iterative squaring: we initialize $\mathbf{N}^*_{\mathcal{V}\mathcal{V}}$ to $\mathbf{I} + \mathbf{N}_{\mathcal{V}\mathcal{V}}$, and repeatedly perform the computation $\mathbf{N}^*_{\mathcal{V}\mathcal{V}} \leftarrow \mathsf{Multiply}(\mathbf{N}^*_{\mathcal{V}\mathcal{V}}, \mathbf{N}^*_{\mathcal{V}\mathcal{V}})$ until $\mathbf{N}^*_{\mathcal{V}\mathcal{V}}$ has converged. Note that, since the above computation involves *boolean* matrices, it is guaranteed to converge.

6.2. Elimination after generation

Elimination of the vanishing states after the reachability set has been generated is fairly straightforward. The next-state functions are constructed as described in Section 4, with the immediate events having priority over the timed events, and are summed to obtain the overall next-state function $\mathcal{N}$ using ElementWise. The overall next-state function is then split into functions for each level using Extract, and the set of reachable states $\mathcal{S} = \mathcal{T} \cup \mathcal{V}$ is generated using the saturation algorithm. The set of tangible, reachable states $\mathcal{T}$ can be determined using a simple query, as described in [26]. First, we determine the set of potential states that are tangible, by determining the states that do not enable any immediate events. This is done by computing $\mathbf{P}_{\mathcal{T}} = \mathsf{EmptyRows}(\mathbf{N}_{\mathcal{V}})$, as described earlier. The non-empty rows of $\mathbf{P}_{\mathcal{T}}$ correspond to the potential tangible states. We then compute the intersection of the reachability set with the set of potential tangible states, which gives us the set of tangible, reachable states.

7. Experimental results

Prototypes of both "elimination after generation" and "elimination during generation" are implemented in the SMART tool [14]. All results are obtained from 933 MHz Pentium III workstations with 512Mb of RAM, running Linux. No reported results made use of virtual memory.



(a) Subnet for dining philosopher i

(b) Kanban model

(c) Polling model: subnet for queue i

(d) Flexible Manufacturing System (FMS) model

Figure 11. Models used in the experiments

We examine several models taken from the literature, each parameterized by an integer N. The dining philosopher model, taken from [26, 27], is composed of N subnets, one for each philosopher. The subnet for philosopher i is shown in Figure 11(a). Our model differs from the models in [26, 27]: we use immediate transitions instead of timed transitions for the action of picking up a fork. The dining philosophers net is broken into subnets so that each subnet consists of two adjacent philosophers; thus, the GSPN model for N philosophers contains $K = \frac{N}{2}$ submodels. We also study a kanban network and a flexible manufacturing system (FMS), taken from [15] and [18]. These models are fixed in size but have N tokens initially in certain places, corresponding to parts or jobs that circulate in the system. We use the "immediate" version of the kanban model, which has immediate synchronizing transitions. The FMS model is modified from its original presentation [18]: all arc cardinalities are one in our version. The kanban network and FMS model are shown in Figure 11(b) and Figure 11(d), respectively. Finally, we consider a multiserver polling system, described in [2], using the $1 \times Q$ server utilization policy. The system consists of multiple queues, and servers that move from queue to queue. The polling model is composed of a subnet for each queue, and contains multiple tokens corresponding to the numbers of jobs and servers in the system. The subnet for a single queue is shown in Figure 11(c). The number of servers is set by initializing the number of tokens in place Pp_0. The number of jobs per queue is set by initializing the number of tokens in places Pb_i. Our model uses N queues, jobs, and servers; thus, both the model size and initial numbers of tokens increase with N. For the kanban, FMS, and polling models, we use the finest possible decomposition: one place per subnet. For all models, we assume that immediate transitions have equal priority.

Table 1 compares the costs, in terms of computational and storage requirements, of constructing the tangible reachability set $\mathcal{T}$ for elimination during generation versus elimination after generation. When vanishing states are eliminated during generation, we construct the next-state function $\mathcal{N}'$ using Equation 3 and then generate $\mathcal{T}$ directly. Conversely, eliminating vanishing states after generation re-

Model	$\|\mathcal{T}\|$	Elimination during generation					Elimination after generation				
		Generating $\mathcal{N}'$			Generating $\mathcal{T}$		Generating $\mathcal{N}$			Generating $\mathcal{T}$	
		CPU (sec)	Memory (Kb) Final	Peak	CPU (sec)	Memory Peak (Kb)	CPU (sec)	Memory (Kb) Final	Peak	CPU (sec)	Memory Peak (Kb)
Phils 16	4.87×10^{6}	0.2	74	174	0.8	1,051	0.04	43	133	1.3	2,382
Phils 30	3.46×10^{12}	1.2	150	631	19	16,637	0.08	85	262	10	16,603
Phils 60	1.20×10^{25}	9.5	314	2,508	389	166,520	0.21	176	539	69	91,972
Phils 90	4.15×10^{37}	—	—	—	—	—	0.39	266	815	204	228,561
Kanban 8	4.23×10^{7}	0.3	42	188	1.1	878	0.00	23	40	0.5	511
Kanban 30	2.36×10^{12}	96	147	22,275	2,157	193,920	0.02	80	144	67	35,594
Kanban 40	2.86×10^{13}	—	—	—	—	—	0.03	106	191	280	99,346
Kanban 50	2.01×10^{14}	—	—	—	—	—	0.04	132	238	979	224,295
FMS 8	4.59×10^{6}	2	23	1,758	0.1	153	0.01	15	33	0.2	228
FMS 20	8.83×10^{9}	356	51	63,036	1.5	1,376	0.01	33	74	2.5	2,194
FMS 40	4.97×10^{12}	—	—	—	—	—	0.03	64	144	29	14,311
FMS 80	3.71×10^{15}	—	—	—	—	—	0.05	124	282	447	102,070
Poll 5	5.91×10^{6}	0.1	18	62	0.2	220	0.01	12	31	0.4	390
Poll 10	9.34×10^{16}	9.4	68	1,784	4.6	4,677	0.03	44	111	13	8,795
Poll 15	2.28×10^{28}	297	147	15,284	36	29,288	0.07	96	240	113	56,180
Poll 20	3.20×10^{40}	—	—	—	—	—	0.14	167	418	540	214,350

Table 1. Time and space requirements to construct the tangible reachability set $\mathcal{T}$

quires construction of the next-state function $\mathcal{N}$, generation of the reachable states $\mathcal{S}$, and determination of $\mathcal{T}$ using a query on $\mathcal{S}$. For both techniques, we show the costs of constructing the appropriate next-state function, and the costs of constructing the set $\mathcal{T}$. Since MDDs and matrix diagrams may expand and contract during manipulation, both the peak and final memory usage is reported. For next-state function construction, the memory usage reported is the total memory required for the matrix diagram. For the generation of $\mathcal{T}$, the memory usage reported is the total memory required for MDD nodes. Since the resulting MDD encoding $\mathcal{T}$ is the same for both techniques, we do not report the final memory usage for $\mathcal{T}$.

As we expect, constructing the next-state function $\mathcal{N}'$ is more expensive than constructing $\mathcal{N}$, since the construction of $\mathcal{N}'$ requires the construction of $\mathcal{N}$ and then application of Equation 3. However, we see that, as the model size increases, the space and time requirements to compute $\mathcal{N}'$ grow much faster than those to compute $\mathcal{N}$. This rapid increase is due to the transitive closure computation in Equation 3. The dashes in the table correspond to cases in which the storage requirements to construct $\mathcal{N}'$ exceeded available memory.

Next, we consider the computational and storage requirements to generate $\mathcal{T}$, once the appropriate next-state function has been constructed. While our intuition may suggest that elimination during generation will require less memory than elimination after generation, this is not always the case. This is due to the property of MDDs that the computational and storage complexities for manipulation depend on the number of *nodes* in the MDD, not the number of states encoded by the MDD. Removal of states from an MDD may actually increase its storage requirements! For example, computational and storage requirements for reachability set generation for the dining philosophers model are significantly lower when all transitions are timed [12]. As both the computational and storage complexities depend on the number of nodes in the MDD, there is a correlation between the peak memory and CPU time for generation of $\mathcal{T}$. However, there is no clear trend as to when the generation of $\mathcal{T}$ is more efficient if elimination during generation is used versus elimination after generation. When a comparison is possible, we see that elimination during generation is more efficient for the FMS and polling models, and elimination after generation is more efficient for the kanban model. For the dining philosophers model, elimination during generation is more efficient for small models, while elimination after generation is more efficient for large models.

Considering the total computational and storage requirements to construct $\mathcal{T}$ (i.e., the requirements to construct $\mathcal{N}$ or $\mathcal{N}'$ plus the requirements to construct $\mathcal{T}$), we see that the expense of constructing $\mathcal{N}'$ tends to make elimination during generation more expensive and less practical. This is due to the high cost of computing the transitive and reflexive closure in Equation 3. Iterative squaring is not the best way to compute transitive closure (partly because the matrix diagram nodes become dense, which increases the

computational cost of Multiply); we are currently developing more efficient implicit techniques to be used with matrix diagrams and comparing them with other implicit techniques (such as [24] for BDDs). Substantial reduction in the cost of the transitive and reflexive closure computation might make elimination during generation more competitive; until this occurs, elimination after generation appears to be a better solution in practice.

The first row for each model in Table 1 corresponds to the largest tangible reachability set that could be generated using explicit techniques. For a rough comparison of times with explicit approaches, generation of the tangible states for the FMS model with $N = 8$ required about 2.1 seconds and 0.21 seconds using our implicit approaches; a straightforward explicit approach in SMART (using a hash table to store states) required about 762 seconds, and we estimate that the specialized hashing technique presented in [20] would require about 390 seconds (if CPU times are adjusted for the difference in processor speeds). Our results are fairly typical with respect to other comparisons between explicit and implicit approaches, and illustrate the efficiency of implicit techniques in terms of both computational and storage requirements. In practice, implicit techniques often allow for the analysis of much larger systems than would otherwise be possible with explicit techniques.

8. Conclusion

Implicit techniques for reachability set generation are quite promising, as they are often able to handle extremely large sets, but are usually limited to handling restricted classes of models. We addressed one such limitation of current implicit reachability set generation techniques: their inability to handle events with priorities and immediate events. We presented an alternate representation of the next-state function of a model, based on matrix diagrams. Manipulations of the matrix diagrams allow for construction of next-state functions that can handle models with an acyclic event priority structure, including models with immediate events. We adapted the saturation algorithm, originally developed for Kronecker-based representations of the next-state function, to work with matrix diagrams. We showed how vanishing states can be eliminated either during generation or after generation, when only the tangible states are of interest.

Empirical results indicate that, while the costs of state generation can be lower when vanishing states are eliminated during generation, the approach is often impractical due to the extra computation required to construct the adjusted next-state function. This is mainly due to the cost of computing the transitive and reflexive closure of the vanishing-to-vanishing next-state function. One possible direction for further research is the development of improved techniques that reduce this cost. Another extension that we are investigating is to construct the local state spaces $\mathcal{S}_k$ *during* implicit generation of $\mathcal{S}$, following the ideas of [13].

References

[1] M. Ajmone Marsan, G. Balbo, and G. 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.

[2] M. Ajmone Marsan, S. Donatelli, F. Neri, and A. Scalia. Approximate GSPN analysis of multiserver polling systems. In *Proceedings of the International Conference on Telecommunication, Distribution and Parallelism (TDP'96)*, La Londe Les Maures, France, June 1996.

[3] G. Balbo, G. Chiola, G. Franceschinis, and G. Molinari Roet. On the efficient construction of the tangible reachability graph of generalized stochastic Petri nets. In *2nd Int. Workshop on Petri Nets and Performance Models (PNPM'87)*, pages 136–145, Madison, Wisconsin, Aug. 1987. IEEE Comp. Soc. Press.

[4] A. Blakemore. The cost of eliminating vanishing markings from generalized stochastic Petri nets. In *3rd Int. Workshop on Petri Nets and Performance Models (PNPM'89)*, pages 85–92, Kyoto, Japan, Dec. 1989. IEEE Comp. Soc. Press.

[5] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a BDD package. In *Conference proceedings on 27th ACM/IEEE design automation conference*, pages 40–45. ACM Press, 1990.

[6] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comp.*, C-35(8):677–691, Aug. 1986.

[7] 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.

[8] 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.

[9] G. Chiola. Compiling techniques for the analysis of stochastic Petri nets. *Perf. Eval.*, Sept. 1988.

[10] G. Chiola, M. Ajmone Marsan, G. Balbo, and G. Conte. Generalized stochastic Petri nets: a definition at the net level and its implications. *IEEE Trans. Softw. Eng.*, 19(2):89–107, Feb. 1993.

[11] G. Ciardo. Discrete-time Markovian stochastic Petri nets. In W. J. Stewart, editor, *Computations with Markov Chains*, pages 339–358. Kluwer, Boston, MA, 1995.

[12] 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)*, LNCS 2031, pages 328–342, Genova, Italy, Apr. 2001. Springer-Verlag.

[13] G. Ciardo, R. Marmorstein, and R. Siminiceanu. Saturation unbound. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Warsaw, Poland, Apr. 2003. Springer-Verlag. To appear.

[14] 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.

[15] 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.

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

[17] 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.

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

[19] S. Donatelli and P. Kemper. Integrating synchronization with priority into a Kronecker representation. *Perf. Eval.*, 44(1-4):73–96, Apr. 2001.

[20] B. Haverkort, A. Bell, and H. Bohnenkamp. On the efficient sequential and distributed generation of very large Markov chains from stochastic Petri nets. In P. Buchholz, editor, *8th Int. Workshop on Petri Nets and Performance Models (PNPM'99)*, pages 12–21, Zaragoza, Spain, Sept. 1999. IEEE Comp. Soc. Press.

[21] H. Hermanns, M. Kwiatkowska, G. Norman, D. Parker, and M. Siegle. On the use of MTBDDs for performability analysis and verification of stochastic systems. *Journal of Logic and Algebraic Programming: Special Issue on Probabilistic Techniques for the Design and Analysis of Systems*, 2003. To appear.

[22] T. Kam, T. Villa, R. Brayton, and A. Sangiovanni-Vincentelli. Multi-valued decision diagrams: theory and applications. *Multiple-Valued Logic*, 4(1–2):9–62, 1998.

[23] P. Kemper. Numerical analysis of superposed GSPNs. *IEEE Trans. Softw. Eng.*, 22(9):615–628, Sept. 1996.

[24] Y. Matsunaga, P. C. McGeer, and R. K. Brayton. On computing the transitive closure of a state transition relation. In *Proc. 30th International Design Automation Conference*, pages 260–265. ACM Press, 1993.

[25] A. S. Miner. Efficient solution of GSPNs using Canonical Matrix Diagrams. In R. German and B. Haverkort, editors, *9th Int. Workshop on Petri Nets and Performance Models (PNPM'01)*, pages 101–110, Aachen, Germany, Sept. 2001. IEEE Comp. Soc. Press.

[26] 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)*, LNCS 1639, pages 6–25, Williamsburg, VA, USA, June 1999. Springer-Verlag.

[27] E. Pastor, O. Roig, J. Cortadella, and R. Badia. Petri net analysis using boolean manipulation. In R. Valette, editor, *Application and Theory of Petri Nets 1994 (Proc. 15th Int. Conf. on Applications and Theory of Petri Nets)*, LNCS 815, pages 416–435, Zaragoza, Spain, June 1994. Springer-Verlag.