Saturation for a general class of models

Andrew S. Miner
Department of Computer Science
Iowa State University, Ames, Iowa, 50011
Phone: +1-515-294-2392
FAX: +1-515-294-0258
email: asminer@iastate.edu

***Abstract*—Implicit techniques for construction and representation of the reachability set of a high–level model have become quite efficient for certain types of models. In particular, previous work developed a "saturation" algorithm that exploits asynchronous behavior to efficiently construct the reachability set using multiway decision diagrams, but uses a Kronecker product expression to represent each model event. For models whose events do not naturally fall into this category, use of the saturation algorithm requires adjusting the model by combining components or splitting events into sub-events until a Kronecker product expression is possible. In practice, this can lead to additional overheads during reachability set construction. This article presents a new version of the saturation algorithm that works for a general class of models: models whose events are not necessarily expressible as Kronecker products, models containing events with complex priority structures, and models whose state variables have unknown bounds. Experimental results are given for several examples.**

***Keywords*-Model checking [D.2.4.e]; Stochastic analysis [D.4.8.g]**

I. Introduction

The generation of the reachability set for a discrete–state model is a necessary step and a significant challenge in many types of studies. Formal verification techniques, including model checking [15], may require reachability set construction to verify some desirable property, such as absence of deadlock. In performance evaluation studies, the reachability set is often required to generate a Markov reward model, which in turn is analyzed to produce performance measures of interest. In either case, reachability set generation can be difficult in practice due to the state explosion problem: the reachability set can be extremely large, even for a "small" model. Discovery of the reachability set by enumeration, using a straightforward breadth–first search and an explicit representation for states, quickly becomes impractical for such complex systems. Partial order reductions [22], [36] can alleviate state explosion in some cases by pruning certain paths during reachability set exploration.

Approaches based instead on implicit methods (also known as *symbolic* methods), which often have space and time complexity much less than the size of the reachability set, have been embraced by many researchers. Decision diagrams, in particular binary decision diagrams (BDDs) [1] and multi–way decision diagrams (MDDs) [21], are widely accepted as efficient and compact data structures for the representation of extremely large sets. Set operations can be performed with complexity dependent on the size of the decision diagram representations, rather than the cardinality of the sets. While decision diagrams degrade to an explicit representation in the worst case, they have been used successfully in practice to generate extremely large reachability sets [6], [7], [10], [30], [32].

To represent sets of states with decision diagrams, each state is decomposed into a collection of K substates, corresponding to levels of nodes in the decision diagram. In an asynchronous system, a given state–to–state transition usually affects only a few of the K substates. The *saturation* algorithm exploits this property to generate reachability sets extremely efficiently for asynchronous systems [10], but originally imposed three constraints: first, that the possible substates for component k (or a reasonable superset) are known *a priori*; second, that actions have equal priority; third, that the effect of an action on a component k depends only on substate k. The first and second requirements are addressed (in isolation) in [11] and [28], respectively. The third requirement, known as the "Kronecker product" requirement, is the subject of this article.

The remainder of the article is organized as follows. Section II briefly reviews background material. Section III presents a variant of the saturation algorithm that eliminates the Kronecker product requirement, assuming the component substates and next–state function are already known. Section IV extends this to generate the reachability set, the next–state function, and the

component substates simultaneously. Section V discusses implementation details. Experimental results for several examples are given in Section VI. Related work is discussed in Section VII. Finally, Section VIII concludes the article.

II. Background and definitions

Reachability set construction is useful for a variety of discrete–state models, including software, which can be represented using Petri nets [31], process algebras [20], or source code itself. Rather than use a specific high–level formalism, we specify a discrete–state model as a triple $(\hat{\mathcal{S}}, \mathcal{N}, \mathcal{S}^0)$, where

- $\hat{\mathcal{S}}$ is the set of possible states of the model;
- $\mathcal{N} : \hat{\mathcal{S}} \to 2^{\hat{\mathcal{S}}}$ is the next–state function, where $\mathcal{N}(\mathbf{i})$ specifies the set of states that can be reached from state $\mathbf{i} \in \hat{\mathcal{S}}$ in one step;
- $\mathcal{S}^0 \subseteq \hat{\mathcal{S}}$ is the non-empty set of initial states.

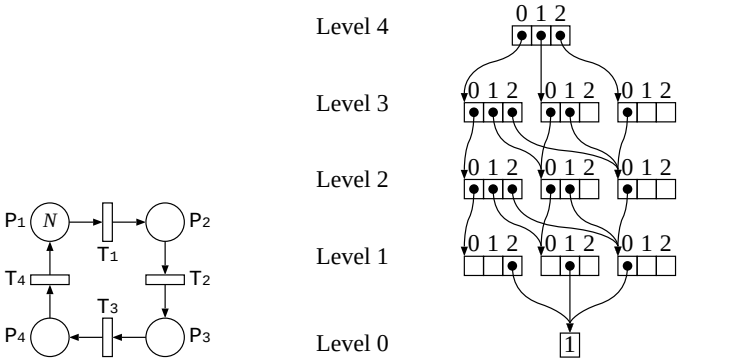
For this article, we assume that the set $\hat{\mathcal{S}}$ is finite, but we do not require advance knowledge of $\hat{\mathcal{S}}$. The reachability set $\mathcal{S}$ is the reflexive and transitive closure of $\mathcal{N}$, starting from the initial states:

$$\mathcal{S} = \mathcal{S}^0 \cup \mathcal{N}(\mathcal{S}^0) \cup \mathcal{N}(\mathcal{N}(\mathcal{S}^0)) \cup \cdots = \mathcal{N}^*(\mathcal{S}^0),$$

where we extend the next–state function to apply also to sets of states, $\mathcal{N}(\mathcal{I}) = \bigcup_{\mathbf{i} \in \mathcal{I}} \mathcal{N}(\mathbf{i})$.

A. MDDs to represent subsets of $\hat{\mathcal{S}}$

We require that the model is composed of $K > 1$ components (when $K = 1$, our approach is equivalent to explicit representation and generation), with $\hat{\mathcal{S}} = \mathcal{S}_K \times \cdots \times \mathcal{S}_1$ and $n_k = |\mathcal{S}_k|$. The set $\mathcal{S}_k$ is usually referred to as the set of *local states* for component k. For notational convenience, we assume that $\mathcal{S}_k = \{0, \ldots, n_k - 1\}$. Any subset of $\hat{\mathcal{S}}$ can then be represented using an MDD, defined as follows [21]. An MDD is a directed, acyclic graph consisting of m *terminal* nodes, which are labeled from integers $\{0, \ldots, m-1\}$, and *non-terminal* nodes p, which are labeled with a variable v_k, and contain a vector of n_k pointers to other nodes. We use the shorthand $p[i]$ to denote element i of this vector. In this article, we assume all MDDs are *ordered*: all nodes with variable v_k are said to belong to *level* k, terminal nodes are said to belong to level 0, and all pointers from a level–k node are to nodes at a level less than k. The level of a node p is denoted $Top(p)$. A given state $\mathbf{i} = (i_K, \ldots, i_1)$ is contained in a subset represented by a K–level MDD if and only if the path through the MDD, starting at the level–K node, following downward pointer i_k at level k, reaches terminal node 1.



(a) Simple model (b) MDD encoding of $\mathcal{S}$, $N = 2$

Fig. 1. A model and its reachability set

As a simple example, a small Petri net model is shown in Fig. 1(a). The model has $K = 4$ components, where the local state of component k corresponds to the number of tokens in place p_k. The reachability set, encoded as an MDD, is shown in Fig. 1(b) for the case $N = 2$, which has initial state $(0, 0, 0, 2)$. Paths that lead only to terminal node 0 are omitted for clarity. From the MDD, it can be seen that state $(0, 1, 0, 1)$ is reachable: the path starting from the level–4 root node, and following the pointers for $p_4 = 0$ at level 4, $p_3 = 1$ at level 3, $p_2 = 0$ at level 2, and $p_1 = 1$ at level 1, reaches terminal node 1. Conversely, state $(0, 1, 1, 1)$ is not reachable, because the corresponding path reaches terminal node 0.

MDDs remain compact through the use of reductions. Two level–k nodes p and p' are said to be *duplicates* if $p[i] = p'[i]$ for all i. A level–k node p is said to be *redundant* if $p[i_k] = p[0]$ for all $0 \leq i_k < n_k$. An MDD is *reduced* if it contains no duplicates or redundant nodes. An MDD is *quasi–reduced* if it contains no duplicates, and every path from the level–K node visits all other levels. It can be shown that both reduced and quasi–reduced (ordered) MDDs are canonical representations for a given variable ordering. Set operations such as union, difference, and intersection can be performed directly on their MDD representations [30]; the complexity of these operations is based on the number of nodes in the MDDs, rather than the number of elements in the sets encoded by the MDDs. For the remainder of the article, we assume all MDDs are quasi–reduced and ordered. Note that the MDD shown in Fig. 1(b) is both quasi–reduced and reduced.

B. Matrix diagrams for next–state functions

Like MDDs, a matrix diagram (MxD) is a directed, acyclic graph with terminal and non-terminal nodes; however, each non-terminal node p with variable v_k contains a *matrix* of $n_k \times n_k$ pointers to other nodes [28].

As with MDDs, we assume that all MxDs are ordered. Matrix diagrams can represent a next–state function $\mathcal{N}$ as follows. Given a pair of states $\mathbf{i} = (i_K, \ldots, i_1)$ and $\mathbf{j} = (j_K, \ldots, j_1)$, then $\mathbf{j} \in \mathcal{N}(\mathbf{i})$ if and only if the path through the MxD, starting at the level–K node, following downward pointer $[i_k, j_k]$ at level k, reaches terminal node 1.

Matrix diagrams use a slightly different set of reduction rules than MDDs. A level–k node p is said to be an *identity* node if $p[i_k, i_k] = p[0, 0]$ for all $0 \leq i_k < n_k$, and $p[i_k, j_k] = 0$ for all $i_k \neq j_k$, while the definition of *duplicate* nodes is analogous to the MDD case. A matrix diagram is reduced if it contains no duplicates or identity nodes. Like reduced MDDs, reduced matrix diagrams are a canonical representation for a given variable ordering. Set union and set intersection of two next–state functions encoded as MxDs can be performed similarly to MDDs [28]. It is important to note that an identity node at level k corresponds to the case where component k does not affect and is unchanged in the next state reached along a particular path in the MxD.

An alternate definition of matrix diagrams is sometimes used in the literature (e.g., [12], [18]), in which each node pointer has an associated real value, allowing matrix diagrams to represent matrices of reals. In this article, these will be referred to instead as "*edge–valued* matrix diagrams", for consistency with decision diagram terminology (cf. edge–valued MDDs [13]). The matrix diagrams used in this article can be seen as a special case of edge–valued matrix diagrams where every edge value is either 0.0 or 1.0, where edge values of 0.0 are used only for pointers directly to terminal node 0. As each edge value can be determined from its corresponding pointer, there is no need to explicitly store the edge values for this special case.

C. Kronecker algebra

A well–accepted compact representation for models, especially Markov chains, is based on Kronecker algebra. In particular, the Kronecker product of two square matrices $\mathbf{M}_2$ and $\mathbf{M}_1$, with dimensions N_2 and N_1, respectively, produces a square matrix

$$\mathbf{M}_2 \otimes \mathbf{M}_1 = \begin{bmatrix} \mathbf{M}_2[1,1] \cdot \mathbf{M}_1 & \cdots & \mathbf{M}_2[1,N_2] \cdot \mathbf{M}_1 \\ \vdots & \ddots & \vdots \\ \mathbf{M}_2[N_2,1] \cdot \mathbf{M}_1 & \cdots & \mathbf{M}_2[N_2,N_2] \cdot \mathbf{M}_1 \end{bmatrix}$$

with dimension $N_2 N_1$, in which every element of $\mathbf{M}_2$ is multiplied by every element of $\mathbf{M}_1$. Since the Kronecker product is associative, the above generalizes to the Kronecker product of K matrices: $\mathbf{M} = \mathbf{M}_K \otimes \cdots \otimes \mathbf{M}_1$. For additional properties of Kronecker products, we refer the interested reader to [17]. Kronecker algebra has been used quite successfully in the analysis of large Markov chains, and is closely related to the use of edge–valued matrix diagrams for the same purpose. The interested reader is referred to recent survey articles [5], [26] for more information.

In the context of reachability set generation, Kronecker algebra can be used to represent a next–state function $\mathcal{N}$ if it can be expressed as $\mathcal{N}(\mathbf{i}) = \bigcup_{e \in \mathcal{E}} \mathcal{N}^e(\mathbf{i})$, where each $\mathcal{N}^e$ can be expressed as a Kronecker product for a particular *event* $e \in \mathcal{E}$ of the model. For many types of formalisms, the set $\mathcal{E}$ corresponds naturally to formalism features; for instance, $\mathcal{E}$ might be the set of transitions of a Petri net. A function $\mathcal{N}^e$ can be expressed as a Kronecker product if it can be written as $\mathcal{N}^e(\mathbf{i}) = \mathcal{N}^e_K(i_K) \times \cdots \times \mathcal{N}^e_1(i_1)$, where the cross products become Kronecker products if the functions $\mathcal{N}^e_k$ are instead represented as boolean matrices $\mathbf{N}^e_k$. We say that event e has a *Kronecker–product form* in this case. Note that the MxD representation of a Kronecker product has a particular shape, in which there is at most one node per level, and all downward pointers from a non-terminal node are either to terminal node 0 or to one other node. This type of next–state function representation has been used successfully in many reachability set approaches, including [3], [10], [24], [28], [30].

In theory, it is always possible to obtain a Kronecker–product form for each event $e \in \mathcal{E}$ by re-defining the set $\mathcal{E}$ as necessary. Note that this may cause a single formalism event to be represented by several events in $\mathcal{E}$; in the worst case, each function $\mathcal{N}^e$ describes exactly one state–to–state transition: $\mathcal{N}^e(\mathbf{i}) = \{\mathbf{j}\}$, and $\mathcal{N}^e(\mathbf{i}') = \emptyset$ for all $\mathbf{i}' \neq \mathbf{i}$. Alternatively, a Kronecker–product form may be obtained by combining components, causing K to decrease, until each $\mathcal{N}^e$ has Kronecker–product form; in the worst case, all components must be combined into $K = 1$ component. In the Markovian setting, the "generalized" Kronecker product [19], which allows matrix entries to be functions, is sometimes used when events do not have an (ordinary) Kronecker–product form. We are unaware, however, of any implicit reachability set generation algorithms to make use of this.

Example MxDs corresponding to next–state functions for the simple model of Fig. 1(a) are shown in Fig. 2. Again, paths to terminal node 0 are omitted for clarity. Note that the next–state functions $\mathcal{N}^{t_4}$, $\mathcal{N}^{t_3}$, $\mathcal{N}^{t_2}$ and $\mathcal{N}^{t_1}$ are Kronecker products, while $\mathcal{N}^{t_4} \cup \mathcal{N}^{t_3}$ is not. In all cases, the "skipped levels" are due to omitted identity nodes for components not affected by the event.

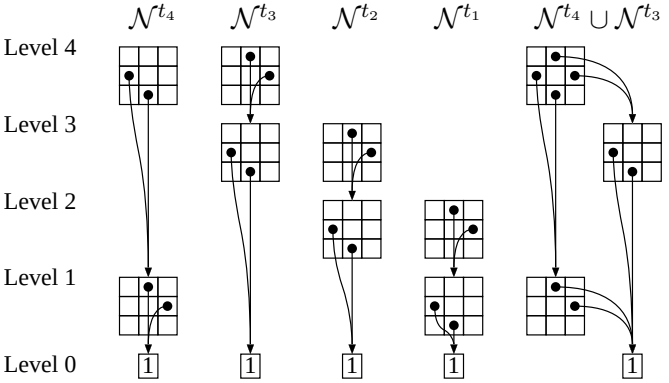


Fig. 2. Next-state functions for events

D. Saturation

The saturation algorithm [10] exploits the property of asynchronous systems that some state changes do not affect particular components. Given a next–state function for event e, any components unaffected by event e will produce identity nodes in the MxD representation of $\mathcal{N}^e$. In particular, if an event affects component k, but does not affect components $(k+1)$ through K, then the top–most MxD node for $\mathcal{N}^e$ will be a level–k node. The saturation algorithm [10] partitions the next–state function $\mathcal{N}$ into $\mathcal{N}_K, \ldots, \mathcal{N}_1$, where $\mathcal{N}_k$ is the union of all $\mathcal{N}^e$ that affect component k but do not affect components $k+1$ through K. During reachability set generation, any newly–created level–k MDD node is "saturated", by repeatedly considering all transitions in $\mathcal{N}_k$ until no new states are reached.

As originally presented in [10], the saturation algorithm required each $\mathcal{N}_k$ be expressed as a union of several $\mathcal{N}^e$, with each $\mathcal{N}^e$ expressible as a Kronecker product. The functions $\mathcal{N}_K, \ldots, \mathcal{N}_1$ and the initial state of the model are provided as inputs, implying:

1) Sets $\mathcal{S}_k$ and next–state functions $\mathcal{N}^e$ are known.
2) All events have equal priority.
3) For all $e \in \mathcal{E}$, $\mathcal{N}^e$ is a Kronecker product.

The saturation algorithm was modified in [11] to address requirement (1) by constructing the sets $\mathcal{S}_k$ and the next–state functions $\mathcal{N}^e$ during generation of the reachability set. Requirement (2) was addressed in [28], for cases where the functions $\mathcal{N}^e$ would be Kronecker products if priorities are ignored, by constructing MxDs for each $\mathcal{N}^e$ and then correcting the MxDs to take priorities into account. However, until now, the use of saturation has not been possible without requirement (3), except by splitting events or merging components until the requirement holds. Note that this requirement has been addressed for other implicit reachability set generation algorithms; these are discussed briefly in Section VII.

Note that the simple model of Fig. 1(a) satisfies the Kronecker product requirement using $K = 4$ components and events $\mathcal{E} = \{t_4, t_3, t_2, t_1\}$, since the next–state functions $\mathcal{N}^{t_4}, \mathcal{N}^{t_3}, \mathcal{N}^{t_2}, \mathcal{N}^{t_1}$ are Kronecker products. If all transitions have equal priority, and the model parameter N is fixed, then the model satisfies all three requirements. However, consider an example where the neighboring elements of an array may be exchanged in a nondeterministic fashion. If the dimension of the array is K, there are K state variables and $K-1$ events, where event e_k exchanges the values of state variables v_k and v_{k+1}. If each state variable is a model component, then this model does not satisfy the Kronecker product requirement: if event e_k occurs, the new value of state variable v_k depends on the value of state variable v_{k+1}. To satisfy the Kronecker product requirement, one must either merge components (i.e., group state variables v_k and v_{k+1} into the same component), which will ultimately produce one large component with $K = 1$, or split each event e_k into $e_{k,m,n}$, where (sub)event $e_{k,m,n}$ exchanges the values of state variables v_k and v_{k+1} when $v_k = m$ and $v_{k+1} = n$. Splitting the events in this way allows for the use of the saturation algorithm with the Kronecker product requirement, since each event $e_{k,m,n}$ can be expressed as a Kronecker product.

III. Lifting the Kronecker requirement

A. Constructing the next–state function

For simplicity, assume (for now) that local state spaces $\mathcal{S}_k$ are known, and events have equal priority. For each event e, we express the next–state function $\mathcal{N}^e$ as

$$\mathcal{N}^e = \mathcal{N}^e_{\mathcal{G}^e_n} \times \cdots \times \mathcal{N}^e_{\mathcal{G}^e_1} \tag{1}$$

where $\mathcal{G}^e_g$ is a group of components, $\mathcal{H}^e = \{\mathcal{G}^e_n, \ldots, \mathcal{G}^e_1\}$ is a partition of the set of components, and function $\mathcal{N}^e_{\mathcal{G}^e_g}$ cannot be broken into a cross product of functions with smaller groups. A component not affected by event e will be in its own group $\mathcal{G}$, and $\mathcal{N}^e_{\mathcal{G}}$ will be the identity function. If $\mathcal{N}^e$ satisfies the Kronecker product requirement, then each group will consist of a single component. Otherwise, components that violate the requirement are grouped together (as suggested in [10], [30]); however, different groupings may be used for different events. The partition $\mathcal{H}^e$ for event e can be determined automatically by examining the high–level model: initially, there are K groups, with $\mathcal{G}_k = \{k\}$; if a dependency exists between components k_1 and k_2 that is inexpressible as a Kronecker product, then groups containing k_1 and k_2 are merged. In the worst case, $\mathcal{H}^e$ consists of a single group containing every component;

```
void thread1()
 0: if (v3 > 0) {
 1:   v4 += v2;
  : }
 2: ...
```

Fig. 3. Source code segment

this occurs if an event affects every component in a way that cannot be broken into a cross product.

As a simple example model, consider the segment of code shown in Fig. 3. Events correspond to the execution of each line of code (assuming a statement executes atomically). The state variables include v_4, v_3, v_2, and the program counter for the thread, which we assign to level 1. When statement 0 occurs, the program counter is modified, depending on the value of v_3; no other variables are affected. Thus, the partition for statement 0 is $\mathcal{H}^{s0} = \{\{4\}, \{3,1\}, \{2\}\}$, where unaffected variables v_4 and v_2 produce identity functions. When statement 1 occurs, v_4 is modified depending on the value of v_2, and the program counter is advanced independently of the variable values. Thus, the partition for statement 1 is $\mathcal{H}^{s1} = \{\{4,2\}, \{3\}, \{1\}\}$.

Once the partition $\mathcal{H}^e$ for a given event e has been determined, we construct the MxD for functions $\mathcal{N}^e_{\mathcal{G}}$ for each $\mathcal{G} \in \mathcal{H}^e$, by enumerating the partial states in the set $\mathcal{S}_{\mathcal{G}} = \mathcal{S}_{k_g} \times \cdots \times \mathcal{S}_{k_1}$, where $\mathcal{G} = \{k_1, \ldots, k_g\}$. Clearly, the cost of enumerating $\mathcal{S}_{\mathcal{G}}$ grows dramatically with the number of components in the group $\mathcal{G}$; in the worst case, $\mathcal{G} = \{1, \ldots, K\}$, $\mathcal{S}_{\mathcal{G}} = \hat{\mathcal{S}}$, and the approach becomes explicit. For each partial state $\mathbf{i} \in \mathcal{S}_{\mathcal{G}}$ that does not disable event e, we must determine the partial states $\mathbf{j} \in \mathcal{S}_{\mathcal{G}}$ that may be reached from $\mathbf{i}$ when event e occurs, considering only the components in $\mathcal{G}$. Each such pair $(\mathbf{i}, \mathbf{j})$ is a partial transition that must be represented by the function $\mathcal{N}^e_{\mathcal{G}}$. These pairs are inserted into the MxD for $\mathcal{N}^e_{\mathcal{G}}$, one at a time, using algorithm Insert shown in Fig. 5, which is a simplified version of the AddEntry algorithm presented in [27] for edge–valued matrix diagrams. The purpose of line 8 will be explained in Section III-B.

Continuing with the simple example from Fig. 3, assume for simplicity that the local state spaces are known to be $\mathcal{S}_4 = \mathcal{S}_3 = \mathcal{S}_2 = \mathcal{S}_1 = \{0, 1, 2\}$, i.e., all variables are bounded by 2. Note $\mathcal{S}_1$ corresponds to the possible values for the program counter. Statement 1 (event $s1$) might occur in all partial states $\mathbf{i} \in \mathcal{S}_4 \times \mathcal{S}_2$. For a particular partial state such as $\mathbf{i} = (1,1)$, it can be determined that the occurrence of $s1$ might lead to partial state $\mathbf{j} = (2,1)$. Note that the pair $(1,1),(2,1)$ is represented by the MxD $\mathcal{N}^{s1}_{\{4,2\}}$ shown in Fig. 4, which can be seen by following pointer $[1,2]$ from the level–4 matrix and pointer $[1,1]$ from the level–2 matrix to reach terminal node 1. Similarly, the behavior of $s1$ with respect to partial states $\mathbf{i} \in \mathcal{S}_1$ is captured by MxD $\mathcal{N}^{s1}_{\{1\}}$ in Fig. 4: $s1$ might occur when the program counter for the thread is equal to one, and its occurrence will advance the program counter to two.

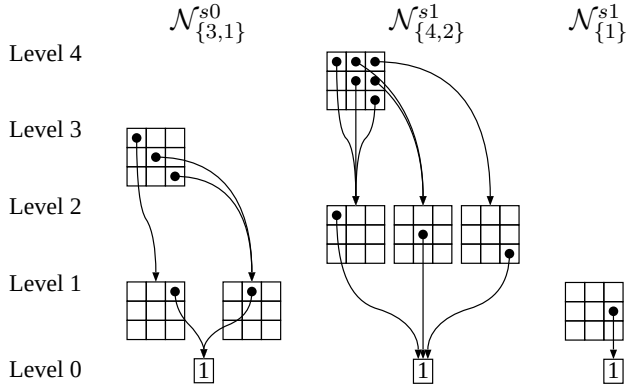


Fig. 4. Next–state functions by group

A cross product operator is necessary to construct the overall next–state function for a given event, after the group–wise next–state functions have been obtained. Given two functions A and B represented as MxDs, such that the sets of levels in the MxD representations of A and B are disjoint, their cross product $C = A \times B$ is simple to construct if the top–most level of B is below the bottom–most level of A: all pointers to terminal node 1 in the MxD representation of A are directed to the root node of the MxD representation of B. If the levels of A and B are instead "interleaved" (but still disjoint sets), then the cross product $C = A \times B$ can be constructed recursively, using algorithm CrossProd shown in Fig. 5, which works as follows. Assuming $Top(A) > Top(B)$ and $Top(A) = k$, then $C = A \times B$ is obtained by constructing a new level–k node whose downward pointers are to the cross product of B and the corresponding downward pointers in the level–k node in A (line 8). The recursion terminates when A or B is a terminal node (lines 2 and 3).

As is common with binary operations on decision diagrams [2], algorithm CrossProd generates a reduced graph and utilizes a *compute table*. Reduction is performed in line 10 (procedure Reduce) by checking if newly–created node C is a duplicate of an existing node: if so, node C is deleted and its duplicate is returned; otherwise, node C is returned. The compute table $opTable$ prevents duplicate computations: after computing $C = A \times B$, the result is added to the compute table (line 11); future requests to compute $A \times B$ will discover this result (line 4). Use of a compute table this way can significantly reduce the computational cost [1].

```
CrossProd(in : MxD A, MxD B)
 1: if Top(A) < Top(B) then Swap(A, B);
 2: if B = 0 then return 0;
 3: if B = 1 then return A;
 4: if ∃C : (A, ×, B, C) ∈ opTable then return C;
 5: k ← Top(A);
 6: C ← new level-k MxD node;
 7: for each i_k, j_k ∈ S_k do
 8:     C[i_k, j_k] ← CrossProd(A[i_k, j_k], B);
 9: end for
10: C ← Reduce(C);
11: opTable ← opTable ∪ (A, ×, B, C);
12: return C;
```

```
Insert(in : MxD N, group G, state i, j)
 1: while G ≠ ∅ do
 2:     k ← max(G);    G ← G \ {k};
 3:     if G = ∅ then N[i_k, j_k] ← 1;
 4:     if N[i_k, j_k] = 0 then
 5:         N[i_k, j_k] ← new level-max(G) node;
 6:     end if
 7:     N ← N[i_k, j_k];
 8:     Ŝ_k ← Ŝ_k ∪ {j_k};
 9: end while
```

```
UpdateNext(in : event e, group G, level k)
  • Update N^e_G for new local states in component k.
 1: G' ← G \ {k};
 2: for each i ∈ S_G' × U_k do
 3:     for each j : MightReach(e, i, j) do
 4:         Insert(N^e_G, G, i, j);
 5:     end for
 6: end for
```

```
UpdateLevel(in : level k)
  • Explore U_k and update next–state functions.
 1: for each event e ∈ E that affects component k do
 2:     Find g such that k ∈ G^e_g;
 3:     UpdateNext(e, G^e_g, k);
 4:     N^e_{G^e_g} ← Reduce(N^e_{G^e_g});
 5: end for
 6: S_k ← S_k ∪ U_k;
```

```
BuildAll(out : MxD N_K, ..., N_1)
 1: for each e ∈ E do
 2:     N^e ← 1;
 3:     for each G ∈ H^e do
 4:         N^e ← CrossProd(N^e, N^e_G);
 5:     end for
 6: end for
 7: Adjust each N^e for priorities;
 8: N = ⋃_{e∈E} N^e;
 9: Split N into N_K, ..., N_1
```

```
PreGenerate(out : set S_K, ..., S_1, MxD N_K, ..., N_1)
 1: for each k ∈ {1, ..., K} do
 2:     Ŝ_k ← S^0_k;    U_k ← S^0_k;    S_k ← ∅;
 3: end for
 4: while ∃k' ∈ {1, ..., K} : U_k' ≠ ∅ do
 5:     UpdateLevel(k');
 6:     for each k ∈ {1, ..., K} do
 7:         U_k ← Ŝ_k \ S_k;
 8:     end for
 9: end while
10: BuildAll(N_K, ..., N_1);
```

Fig. 5. Generating next–state functions and local state spaces simultaneously

B. Constructing local state spaces

The above assumption that the local state spaces $\mathcal{S}_k$ can be constructed in isolation, before constructing the next–state function or the reachability set, is not always realistic, especially considering models that violate the Kronecker product requirement. Local state spaces can be constructed during construction of the next–state functions by allowing the addition of any newly–discovered local states. The initial local states $\mathcal{S}_k^0$ can be determined as the subset of local states $\mathcal{S}_k$ appearing in $\mathcal{S}^0$, so that $\mathcal{S}^0 \subseteq \mathcal{S}_K^0 \times \cdots \times \mathcal{S}_1^0$. We then explicitly explore the groups for each event, adding local states as they are discovered. To this end, we maintain two additional sets of local states: $\hat{\mathcal{S}}_k$, which includes the initial states $\mathcal{S}_k^0$ and any local states that might be reached due to the occurrence of an event; and $\mathcal{U}_k$, which is the set of local states for component k that must still be explored. The set $\mathcal{S}_k$ refers to local states for component k that have been explored already: transitions out of those states are present in the next–state functions. More specifically, for each level–k matrix diagram node within a next–state function, the rows corresponding to $\mathcal{S}_k$ are complete, and the rows corresponding to $\mathcal{U}_k$ have yet to be added.

An algorithm to simultaneously construct local state spaces and next–state functions is shown in Fig. 5, in procedure PreGenerate. Initially, the next–state functions are constructed using the sets $\mathcal{S}_k^0$ as the local states. Newly–discovered local states are added to sets $\hat{\mathcal{S}}_k$ in procedure Insert, via line 8. Whenever new local states are discovered for component k, the next–state functions must be updated for all events that depend on component k. This is done by procedure UpdateLevel, which updates MxDs to include local states in $\mathcal{U}_k$ for every event e and group $\mathcal{G}$ such that $\mathcal{N}_{\mathcal{G}}^e$ depends on component k. The set of explored local states $\mathcal{S}_k$ is then updated. This process repeats until no new local states are discovered.

This algorithm may generate a larger set $\mathcal{S}_k$ than can actually be reached, since the local states are generated considering only partial states within $\mathcal{S}_\mathcal{G}$ for some group $\mathcal{G}$ of components. Due to synchronizations between groups of components, some of these generated local states might never occur within $\mathcal{S}$. This can pose problems since the generated sets $\mathcal{S}_k$ can be significantly larger than the actual set of local states that can occur, and are not guaranteed to be finite even for finite reachability sets $\mathcal{S}$. This problem can often be addressed by adjusting the model appropriately (e.g., by adding inhibitor arcs to a Petri net), but this may be difficult, even for an expert. The alternative is to use a more dynamic method of constructing sets $\mathcal{S}_k$; this approach is considered in Section IV.

After the local state spaces $\mathcal{S}_k$ and the next–state functions for each group of each event have been constructed, procedure BuildAll is called, which first constructs the next–state function $\mathcal{N}^e$ for each event e by building the cross product of the functions for each group of that event (lines 1–6). Once the MxDs are obtained for each next–state function $\mathcal{N}^e$, the techniques described in [28] can be applied to construct the necessary MxDs for use with the saturation algorithm, as follows. First, the next–state functions for each event are corrected for priorities (line 7): if event f has priority over e, then $\mathcal{N}^e$ must be modified so that event e cannot occur whenever event f is enabled. Second, a single overall next–state function $\mathcal{N}$ is constructed as the union of each $\mathcal{N}^e$ (line 8). Finally, the next–state function $\mathcal{N}$ is split into $\mathcal{N}_K, \ldots, \mathcal{N}_1$ (line 9), where $\mathcal{N}_K \cup \cdots \cup \mathcal{N}_1 = \mathcal{N}$ and $\mathcal{N}_k$ describes the state–to–state transitions (if any) that do not depend on components K through $(k+1)$, but do depend on component k.

C. Constructing the reachability set

An algorithm to perform saturation to construct the reachability set $\mathcal{S}$, given next–state functions $\mathcal{N}_K, \ldots, \mathcal{N}_1$ encoded as matrix diagrams, is presented in [28]; this algorithm can be used without modification after procedure BuildAll is called. If the model contains immediate events, the set $\mathcal{S}$ will include both tangible and vanishing states. If the set of tangible reachable states $\mathcal{T}$ is desired, it can be determined as the subset of $\mathcal{S}$ in which no immediate events are enabled, using a simple query [28], [30].

IV. On-the-fly construction

In the previous section, the local state spaces and next–state functions were constructed in a separate step before proceeding with the saturation algorithm. However, there may be cases where it is impossible to construct the local state spaces *a priori*, in particular if synchronizations with other components must be considered. The pregeneration algorithm must therefore be modified so that a newly–discovered local state is not explored (i.e., its row is not added to the next–state functions) until it is known that the local state appears in at least one reachable (global) state. Construction of the local state spaces, next–state functions, and reachability set must therefore be done simultaneously.

A. Updating local state spaces

Conceptually, our approach is similar to [11], which works for a model satisfying the Kronecker product requirement; however, the removal of the Kronecker product requirement leads to additional challenges. Consistent with [11], we write $\mathcal{S}_k$ to denote the set of *confirmed* local states for component k, which refers to local states that are known to appear in at least one reachable (global) state; and $\hat{\mathcal{S}}_k$ to denote the set of potentially–reachable local states for component k, which includes both confirmed and unconfirmed states. Our on-the-fly algorithm confirms a local state $s_k \in \hat{\mathcal{S}}_k$ only when a global state containing s_k has been discovered. Newly–confirmed local states s_k are added to $\mathcal{U}_k$, so that they will be explored when the next–state functions are updated.

B. Updating next–state functions

During reachability set construction, the next–state functions describe possible transitions from states in $\mathcal{S}_K \times \cdots \times \mathcal{S}_1$ to states in $\hat{\mathcal{S}}_K \times \cdots \times \hat{\mathcal{S}}_1$. Level–$k$ MxD nodes consider transitions out of confirmed locals $\mathcal{S}_k$ only, but these transitions can lead to confirmed or unconfirmed locals $\hat{\mathcal{S}}_k$. These MxD nodes can be viewed as having dimension $|\hat{\mathcal{S}}_k| \times |\hat{\mathcal{S}}_k|$ with rows $\hat{\mathcal{S}}_k \setminus \mathcal{S}_k$ containing pointers to terminal node zero, because as discussed earlier, those rows cannot be added until their corresponding local states are known to appear in at least one reachable state.

When a local state s_k is confirmed, the next–state functions must be updated. Specifically, row s_k of all level–k MxD nodes, which previously contained pointers to terminal node zero, must be updated to capture transitions (if any) out of local state s_k. In [11], this is done by updating the appropriate matrix within the Kronecker product for each event. In our case, many levels of MxD nodes may need to be updated: if next–state function $\mathcal{N}^e_\mathcal{G}$ depends on component k (i.e., $k \in \mathcal{G}$), then all levels in $\mathcal{G}$ must be checked in the MxD for $\mathcal{N}^e_\mathcal{G}$;

this can be done using the same procedure UpdateLevel (in Fig. 5) that was used for *a priori* generation of $\mathcal{N}$.

After the individual next–state functions have been updated, functions $\mathcal{N}_K, \ldots, \mathcal{N}_1$ must be reconstructed by calling procedure BuildAll shown in Fig. 5. Unlike [11], which determines each $\mathcal{N}_k$ *a priori* based on the event structure in the high–level model, our approach determines each $\mathcal{N}_k$ dynamically based on the next–state function itself. As such, the addition of a confirmed local state s_k can change $\mathcal{N}_l$ for any l by both adding new transitions and removing transitions. This can occur when s_k is the first discovered local state to affect an event e: before confirming s_k, the next–state function $\mathcal{N}^e$ contains identity nodes at level k; afterward, $\mathcal{N}^e$ contains at least one non–identity node at level k. In this case, the update can remove transitions from $\mathcal{N}_l$ and add them to $\mathcal{N}_k$, for some $l < k$. In contrast, updates in [11] are guaranteed not to remove transitions from any $\mathcal{N}_k$. Thus, unlike in [11], a local state s_k cannot be confirmed in isolation, since its addition may remove transitions from some other $\mathcal{N}_l$ that has already been used to (incorrectly) saturate a level–l node below.

To illustrate this, consider statement 0 from the simple code model in Fig. 3. If the only confirmed value for v_3 is zero, then the behavior of statement 0 will be seen as simply "change the program counter from zero to two", and this will be part of the level–1 next–state function $\mathcal{N}_1$. If some other statement occurs and changes the value of v_3, this will be discovered when a reachable (sub)state for $v_3 \neq 0$ is added to a level–3 MDD node. After this local state is confirmed, the behavior of statement 0 will depend on both level 3 and level 1, and thus will not be part of (new) function $\mathcal{N}_1$. We must prevent the nodes added beneath the level–3 MDD node that triggered this confirmation from being saturated at level 1 using the old, incorrect function $\mathcal{N}_1$.

To address this problem, our algorithm delays exploration of newly–confirmed local state s_k by adding it to $\mathcal{U}_k$. The algorithm first confirms local states along a given MxD path (this is discussed in greater detail below); if any local states are confirmed, then the group–wise next–state functions $\mathcal{N}^e_{\mathcal{G}}$ are updated for any level k with $\mathcal{U}_k \neq \emptyset$ (using procedure UpdateLevel), and functions $\mathcal{N}_K, \ldots, \mathcal{N}_1$ are reconstructed using BuildAll. In addition to solving the above problem, this approach also improves efficiency by reducing the number of times the overall MxDs must be reconstructed.

C. Generating the reachability set

Our modified version of saturation, which updates the next–state function and the local state spaces as necessary, is shown in procedure Saturate in Fig. 6. Given a set of states p encoded as a level–k MDD node, procedure Saturate updates the node to encode $\mathcal{N}^*_{\leq k}(p)$, where "$*$" denotes transitive closure and $\mathcal{N}_{\leq k}$ denotes $\mathcal{N}_k \cup \cdots \cup \mathcal{N}_1$. This is done recursively, by calling RecFire, which determines $\mathcal{N}(p)$ for a given next–state function $\mathcal{N}$ and set of states p and saturates any newly–created MDD nodes. In [11], any unconfirmed local state that is reached by Saturate or RecFire is immediately confirmed. In our case, we instead use a separate preprocessing step to confirm local states: for a given set of states represented as an MDD p and a next–state function $\mathcal{N}$, procedure Enabled determines if the set $\mathcal{N}(p)$ is empty, and if not, adds any unconfirmed locals present in $\mathcal{N}(p)$ to the appropriate $\mathcal{U}_k$ set. If any $\mathcal{U}_k$ is nonempty, then the next–state functions are reconstructed by calling Update$(1, k)$, which updates levels 1 through k (via procedure UpdateLevel) and then rebuilds the next–state functions (via procedure BuildAll). Essentially, any local states that would have been confirmed by RecFire are instead confirmed by procedure Enabled ahead of time, allowing RecFire to saturate nodes with correct, current next–state functions. Procedure RecFire is essentially the same as presented in [28]; it is shown in Fig. 6 for completeness. Procedure Union, also shown in Fig. 6, computes the union of two sets encoded as MDDs; details on this operation can be found in [30].

The saturation algorithm is invoked as follows. Sets $\hat{\mathcal{S}}_k$ and $\mathcal{U}_k$ should be initialized to $\mathcal{S}^0_k$, while $\mathcal{S}_k$ should be initially empty. The next–state functions can then be initialized by calling Update$(1, K)$. Next, the MDD representing the set of initial states $\mathcal{S}^0$ is constructed. If $\mathcal{S}^0 = \mathcal{S}^0_K \times \cdots \times \mathcal{S}^0_1$, then this MDD will contain one node at each level, and all downward pointers at level k will be zero except for pointers corresponding to $\mathcal{S}^0_k$, which will point to the level–$(k-1)$ node. The reachability set is then constructed by saturating all nodes in the MDD for $\mathcal{S}^0$ from the bottom up.

D. Correctness of the algorithm

A proof of correctness for the saturation algorithm is given in [10] when the local state spaces and next–state functions are fixed. Correctness must therefore be re-established when the algorithm is extended to allow for increasing local state spaces and changing next–state functions. As discussed in [11], since the local state space $\mathcal{S}_k$ can increase in size, level–k MDD nodes may vary in size. This is not a problem in theory as long as a correct semantic is used for the missing arcs: if node p does not contain a downward pointer corresponding to local state i (which occurs for nodes created before i was confirmed), it is assumed that $p[i] = 0$.

```
Enabled(in : level k, MDD p, MxD 𝒩)
 1: if 𝒩 = 1 then return p ≠ 0;
 2: if 𝒩 = 0 ∨ p = 0 then return false;
 3: if ∃a : (k, p, 𝒩, a) ∈ enabledTable then return a;
 4: a ← false;
 5: if Top(𝒩) < k then                    • 𝒩 skips this level
 6:    for each i : p[i] ≠ 0 do
 7:       if Enabled(k − 1, p[i], 𝒩) then a ← true;
 8:    end for
 9: else
10:    for each i : p[i] ≠ 0, j : 𝒩[i, j] ≠ 0 do
11:       if Enabled(k − 1, p[i], 𝒩[i, j]) then
12:          a ← true;
13:          if j ∉ 𝒮_k then 𝒰_k ← 𝒰_k ∪ {j};
14:       end if
15:    end for
16: end if
17: enabledTable ← enabledTable ∪ (k, p, 𝒩, a);
18: return a;
```

```
Saturate(in : level k, MDD p)
 1: ℒ ← {i : p[i] ≠ 0 ∧ ∃𝒩_k[i, j] ≠ 0};
 2: while ℒ ≠ ∅ do
 3:    Select and remove i from ℒ;
 4:    for each j : 𝒩_k[i, j] ≠ 0 do
 5:       if Enabled(k − 1, p[i], 𝒩_k[i, j]) then
 6:          if j ∈ 𝒮_k then
 7:             f ← RecFire(k − 1, p[i], 𝒩_k[i, j]);
 8:             u ← Union(k − 1, f, p[j]);
 9:             if p[i] ≠ u then
10:                p[i] ← u;
11:                ℒ ← ℒ ∪ {j};
12:             end if
13:          else
14:             𝒰_k ← 𝒰_k ∪ {j};   • Confirm local state j
15:             Update(1, k);
16:             ℒ ← ℒ ∪ {i};
17:          end if
18:       end if
19:    end for
20: end while
```

```
RecFire(in : level k, MDD p, MxD 𝒩)
 1: if 𝒩 = 1 ∨ k = 0 then return p;
 2: if ∃s : (p, 𝒩, s) ∈ fireTable[k] then return s;
 3: s ← new level-k MDD node;
 4: if Top(𝒩) < k then                    • 𝒩 skips this level
 5:    for each i : p[i] ≠ 0 do
 6:       s[i] ← RecFire(k − 1, p[i], 𝒩);
 7:    end for
 8: else
 9:    for each i : p[i] ≠ 0 do
10:       for each j ∈ 𝒮_k : 𝒩[i, j] ≠ 0 do
11:          f ← RecFire(k − 1, p[i], 𝒩[i, j]);
12:          u ← Union(k − 1, f, s[j]);
13:          if s[j] ≠ u then s[j] ← u;
14:       end for
15:    end for
16: end if
17: if ∃i : s[i] ≠ 0 then
18:    Saturate(k, s);
19: end if
20: Reduce(k, s);
21: fireTable[k] ← fireTable[k] ∪ (p, 𝒩, s);
22: return s;
```

```
Union(in : level k, MDD p, MDD q)
 1: if p = 0 ∨ p = q then return q;
 2: if q = 0 then return p;
 3: if ∃s : (p, q, s) ∈ uTable[k] then return s;
 4: s ← new level-k MDD node;
 5: for each i ∈ 𝒮_k do
 6:    s[i] ← Union(k − 1, p[i], q[i]);
 7: end for
 8: Reduce(k, s);
 9: uTable ← uTable ∪ (p, q, s);
10: return s;
```

```
Update(in : level kbot, level ktop)
 1: for k ← kbot to ktop do
 2:    if 𝒰_k ≠ ∅ then UpdateLevel(k);
 3: end for
 4: BuildAll(𝒩_K, ..., 𝒩_1);
```

Fig. 6. On-the-fly saturation algorithm for matrix diagrams

In [11], the correctness argument is based on the fact that the functions $\mathcal{N}_k$ change but only in a monotone way: the only additions are due to transitions out of newly–discovered local states. As such, once a node p is saturated, it remains saturated with the discovery of a new local state; if the new local state had an effect on p, then it would have been discovered while saturating p. Thus, additional entries in a next–state function $\mathcal{N}_k$, due to the addition of a local state, do not affect correctness.

Finally, we must consider that, since our decomposition of the overall next–state function into $\mathcal{N}_K, \ldots, \mathcal{N}_1$ is *dynamic*, rather than *static* as in [11], the addition of a local state may cause the *removal* of transitions from a function $\mathcal{N}_l$. However, this does not happen in an arbitrary way; rather, it occurs when the addition of a local state i_k causes an event to now depend on component k, causing a "shift" of transitions from $\mathcal{N}_l$ to $\mathcal{N}_k$, for $l < k$. That is, the addition of local state i_k can only add transitions to $\mathcal{N}_{\leq k}$. Thus, the removal of transitions from $\mathcal{N}_l$ does not cause saturated level–l nodes to become incorrect, since those nodes are part of an encoded set at level k. However, there is an issue with the compute tables for RecFire: if level–l node s was created due to a recursive call to RecFire and

then Saturate, then node s is stored in $fireTable[l]$, but this result becomes invalid once transitions are removed from $\mathcal{N}_l$. Strictly speaking, because transitions may be removed from a function $\mathcal{N}_l$, it becomes necessary to store $(p, \mathcal{N}, s, \mathcal{N}_{<k})$ in $fireTable[k]$, and the entries become invalid when transitions are removed from $\mathcal{N}_{<k}$. Alternatively, whenever $\mathcal{N}_l$ changes, and the new $\mathcal{N}'_l$ is not a superset of the old $\mathcal{N}_l$, we must remove all entries from table $fireTable[k]$ for all $l \leq k$.

Note that clearing of the compute tables is necessary only if it is possible to shift transitions from $\mathcal{N}_l$ to $\mathcal{N}_k$. If instead a static assignment of events to next–state functions $\mathcal{N}_k$ is used, then no shift is possible, the new $\mathcal{N}'_l$ will always be a superset of $\mathcal{N}_l$, and the RecFire tables never need to be cleared.

V. Implementation issues

A. MDD nodes

Normally, when the local state spaces are known *a priori*, each level–k MDD node contains exactly $|\mathcal{S}_k|$ downward pointers. If $\mathcal{S}_k$ is allowed to grow after level–k nodes have been created, then the implementation must either expand all level–k nodes whenever $\mathcal{S}_k$ expands, or allow nodes to have different numbers of downward pointers at level–k. Since the former approach adds significant computational (and memory) overhead, we opt instead for the latter approach. Nodes can be stored in "truncated full" format, such that the last downward pointer is not to terminal node zero. Any trailing zero pointers of a node are truncated by procedure Reduce. A benefit of this storage scheme is that the representation of a level–k node is independent of the current size of $\mathcal{S}_k$. Other schemes with this property (such as sparse storage, where only the non–zero downward pointers are stored) can be used just as easily.

It is common to use *reduced* MDDs, which means that redundant nodes are eliminated. Elimination of redundant nodes at level–k causes problems, though, when the set $\mathcal{S}_k$ expands: semantically, expansion of $\mathcal{S}_k$ causes the addition of trailing pointers to terminal node zero to every level–k node, including the eliminated redundant nodes. In other words, except for redundant nodes where all pointers are to terminal node zero, the eliminated redundant nodes at level–k must be rebuilt when $\mathcal{S}_k$ expands, since the addition of a new local state causes these nodes to not be redundant any longer. As in [11], we use quasi–reduced MDDs instead of reduced MDDs to solve this problem: quasi–reduced MDDs do not eliminate redundant nodes (except possibly for nodes whose pointers are all to terminal node zero), thus redundant nodes are also stored in truncated full, and as described earlier, correctly handle expansion of $\mathcal{S}_k$.

The biggest implementation challenge with respect to variable–sized nodes is that of garbage collection. When a node becomes disconnected from the forest, it should be recycled, allowing the memory required to store the node to be reclaimed. Any remaining compute table entries with a recycled node must be purged. When level–k nodes are a fixed size, recycling the memory of a level–k node is trivial, since the next level–k node to be created can use the old node's memory. This is not necessarily true with variable–sized nodes. One approach, described in [10], is to perform garbage collection only occasionally, when the amount of memory to be reclaimed exceeds a threshold; this allows for several nodes to be purged from the compute tables simultaneously. Holes in memory (due to the freed nodes) are eliminated by compaction.

B. Matrix diagram nodes

As with MDD nodes, the matrix diagram nodes are required to be more dynamic than as presented in [28]. The level–k nodes must support a matrix of downward pointers containing $|\hat{\mathcal{S}}_k|$ columns and $|\mathcal{S}_k|$ rows, where the sets $\hat{\mathcal{S}}_k$ and $\mathcal{S}_k$ are allowed to increase over time. In practice, this can be achieved using a sparse matrix representation based on linked–lists for each node, where missing entries are assumed to be to terminal node zero. The MxD nodes for a given level should be allowed to vary in size.

As an alternative, matrix diagrams can be implemented as MDDs, where each level–k matrix diagram node corresponds to two levels (usually denoted k and k') of MDD nodes. This may be preferable, as the common problems of memory management, garbage collection, and node reductions must be implemented only once. With this approach, the level–k MDD nodes correspond to the matrix diagram rows, and have dimension $|\mathcal{S}_k|$, while the level–k' MDD nodes correspond to the matrix diagram columns, and have dimension $|\hat{\mathcal{S}}_k|$. If the row level is directly above the column level, and level–k matrix diagram node p is represented by level–k MDD node p', then $p[i_k, j_k]$ is represented by $(p'[i_k])[j_k]$. Note that, in this storage scheme, equal matrix diagram node rows are shared: they become duplicate level–k' nodes. It is important, though, that the elimination of identity nodes is handled correctly by the MDD implementation. That is, while MDDs may be used to implement matrix diagrams, the meaning of skipped levels must still correspond to identity nodes in the matrix diagram. Thus, if a level–k matrix diagram node is represented by two adjacent MDD levels k and k', then a pointer from a level–$(k+1)'$ MDD node to

a level–$(k-1)$ MDD node corresponds to a skipped level–k matrix diagram node, i.e., an eliminated identity node.

Problems can arise due to identity nodes when sets $\mathcal{S}_k$ increase. For instance, if a particular next–state function $\mathcal{N}$ skips level k, this implies that for the current $\mathcal{S}_k$, function $\mathcal{N}$ does not modify component k. However, if this is no longer true when $\mathcal{S}_k$ increases in size, then a level–k node must appear in $\mathcal{N}$. This is exactly analogous to the discussion of redundant MDD nodes. To avoid this issue, we do not remove identity nodes from group functions $\mathcal{N}_{\mathcal{G}}^e$; this guarantees that the group functions will remain correct when local states are added. Since the overall next–state function $\mathcal{N}$ is constructed as the union of the event–wise functions $\mathcal{N}_e$, each of which is reconstructed as the cross–product of the group functions (in procedure BuildAll), we avoid the need to discover and replace identity nodes.

A more serious issue occurs with respect to compute table entries. If a next–state function $\mathcal{N}_e$ does not affect component k, then its MxD will not contain any level–k nodes, since those nodes would be identity nodes. Thus, when the set $\mathcal{S}_k$ increases, the new next–state function $\mathcal{N}_e'$ will have the same physical representation as $\mathcal{N}_e$; we therefore can have different functions $\mathcal{N}_e$ and $\mathcal{N}_e'$ that are represented by the same MxD nodes. Now, if another next–state function $\mathcal{N}_f$ does affect component k, then a union operation $\mathcal{N}_e \cup \mathcal{N}_f$ will produce a new MxD that contains level–k nodes. But the union operation $\mathcal{N}_e' \cup \mathcal{N}_f$ produces a different MxD, since it represents a different function. Thus, we must distinguish between $\mathcal{N}_e \cup \mathcal{N}_f$ and $\mathcal{N}_e' \cup \mathcal{N}_f$ in the union compute table. Since the function represented by a given MxD depends also on the sets $\mathcal{S}_K, \ldots, \mathcal{S}_1$, one way to avoid the above issue is to store these sets with each compute table entry. When a set $\mathcal{S}_k$ increases to $\mathcal{S}_k'$, none of the entries in the compute table with $\mathcal{S}_k$ will ever be matched. As such, an equivalent approach is to store only the MxD top–level nodes in each compute table entry (as is done ordinarily), and to clear all the matrix diagram compute tables at and above level k whenever the set of local states $\mathcal{S}_k$ increases in size. While this approach is more strict than necessary (it is necessary to remove only the entries whose MxDs contain identity nodes at level k), the sets of local states tend to be rather small in practice, so the number of times this must be done will also be rather small.

C. Changing next–state functions

During saturation at level k, an update can modify $\mathcal{N}_k$; care must be taken to ensure that the most up to date versions of the next–state functions are used. In particular, a transition that triggered an update might move from $\mathcal{N}_l$ to $\mathcal{N}_k$ after the update. Indeed, after an update, it is possible that $\mathcal{N}_k = \emptyset$. Thus, procedure Saturate must frequently check for updates to $\mathcal{N}_k$.

Note that a call to Saturate at level k can recursively call procedure RecFire at level $(k-1)$, which can call RecFire and Saturate at level $(k-2)$. It is possible for an update at level $(k-2)$ to change $\mathcal{N}_k$, which means that when execution returns to RecFire at level $(k-1)$, its passed MxD might not exist; we handle this by delaying the destruction of old MxD nodes until safe to do so. This is valid since the old MxD node used by RecFire at level $(k-1)$ still describes valid transitions within the context of the parameters passed to RecFire. Destruction of old MxD nodes is done in procedure Saturate: when called at level k, Saturate checks for updates to $\mathcal{N}_k$; if it has changed, the old $\mathcal{N}_k$ is destroyed and replaced with the newest version.

VI. Experimental results

Prototypes of our algorithms have been implemented in SMART [8]. We test our approach on several models from the literature (described below), running on 933 Mhz Pentium III workstations with 512Mb of RAM, under the Linux operating system. For all models, we use the finest possible decomposition into components: each state variable (usually a Petri net place) becomes its own component. Results are shown in Table I, including the number of tangible reachable states $|\mathcal{T}|$, the (final) number of MDD nodes required to represent $\mathcal{T}$ (which is the same for all algorithms since the model components are unchanged), and performance measures for saturation using our pregeneration approach described in Section III (columns "PRE"), using our on-the-fly approach described in Section IV (columns "OTF"), and using the on-the-fly, Kronecker product version of saturation described in [11] (columns "[11]"). The implementations of all three methods use the same internal representation for MDD nodes, and use the "lazy" garbage collection policy described in [10], where memory holes are freed only after the reachability set has been generated. We use a linked–list matrix representation for each matrix diagram node, a data structure similar to the matrix representations used in [11]. Thus, we make every effort for a fair comparison, so that observed performance differences are due primarily to the algorithms themselves, rather than implementation details. Performance measures we consider are the peak number of MDD nodes required, the peak total number of compute table entries, and the total CPU time required to generate $\mathcal{T}$ (including the removal of vanishing states).

Model	N	$\|\mathcal{T}\|$	Final nodes	Peak MDD nodes PRE	Peak MDD nodes OTF	Peak MDD nodes [11]	CPU time (sec) PRE	CPU time (sec) OTF	CPU time (sec) [11]	Peak compute table PRE	Peak compute table OTF	Peak compute table [11]
Kanban	20	8.05×10^{11}	696	14,619	14,094	24,067	2	4	4	56,830	86,122	82,448
	40	9.94×10^{14}	2,176	89,219	86,974	158,507	12	45	36	368,820	568,595	561,298
	80	1.56×10^{18}	7,536	610,419	601,134	1,142,587	145	758	775	2,622,400	4,101,717	4,116,198
FTMP	6	9.91×10^{15}	592	1,102	1,102	2,235	1	1	5	3,113	5,417	133,952
	25	5.32×10^{65}	8,876	19,532	19,532	29,215	4	19	985	55,933	96,845	9,786,104
	50	1.42×10^{131}	34,626	78,432	78,432	—	19	139	—	225,533	390,395	—
	100	1.00×10^{262}	136,751	314,357	314,357	—	111	1,130	—	905,983	1,568,120	—
Swaps	12	4.79×10^{8}	4,095	4,096	4,096	4,096	8	14	160	294,790	429,958	135,168
	14	8.72×10^{10}	16,383	16,384	16,384	16,384	60	98	1,828	1,605,462	2,350,934	745,472
	16	2.09×10^{13}	65,535	65,536	65,536	65,536	489	825	29,825	8,388,382	12,320,542	3,932,160
Courier	10	4.25×10^{9}	1,431	543,861	71,735	n/a	538	14	n/a	1,743,038	304,612	n/a
	20	2.26×10^{12}	4,191	2,150,924	227,230	n/a	7,119	82	n/a	7,008,138	857,572	n/a
	40	2.18×10^{15}	13,911	—	801,920	n/a	—	668	n/a	—	2,646,692	n/a
FMS	10	2.54×10^{7}	713	39,960	18,107	n/a	6	4	n/a	188,080	87,848	n/a
	20	8.83×10^{9}	2,213	202,740	77,712	n/a	101	31	n/a	1,090,505	362,788	n/a
	40	4.97×10^{12}	7,613	1,191,600	355,922	n/a	3,091	323	n/a	7,381,310	1,657,168	n/a
Stack	8	6.56×10^{3}	8	50	37	4,347	1	1	230	120	136	334,265
	9	1.97×10^{4}	9	61	38	13,091	3	3	2,292	162	144	1,267,139
	15	1.43×10^{7}	15	154	74	—	3,303	3,304	—	417	264	—

TABLE I
EXPERIMENTAL RESULTS

Measures relating to MxD nodes are omitted for space; in summary, unless otherwise mentioned, we found that the matrix diagram memory requirements are similar to the next–state function memory requirements of [11].

A. Models

We first consider a model of a Kanban manufacturing system [30] which satisfies the Kronecker product requirement of [11]. Note that our approaches are quite competitive even with models satisfying the necessary requirements of [11]. In this case, our pregeneration algorithm has a different number of peak nodes than our on-the-fly algorithm; this is due to different visitation order of local states within procedure Saturate (an update requires us to re-visit a local state).

Next we consider a fault–tolerant multiprocessor system (FTMP) [18], [35]. To satisfy the requirements for [11], events are split into several similar sub-events, each satisfying the Kronecker product requirement; this leads to increased computation time and compute table sizes, since RecFire must be called separately for each event. Indeed, the table memory requirements for [11] makes it inapplicable for the $N = 50$ and $N = 100$ cases.

The "Swaps" model represents an array of N distinct integers, with events to exchange two neighboring integers; this is the model discussed in Section II-D. To satisfy the Kronecker product requirement of [11], each event must be split into N^2 sub-events, one for each possible state of the neighboring integers. This causes a dramatic increase in CPU time for the approach of [11]. Note that there are exactly $N!$ reachable states for this model.

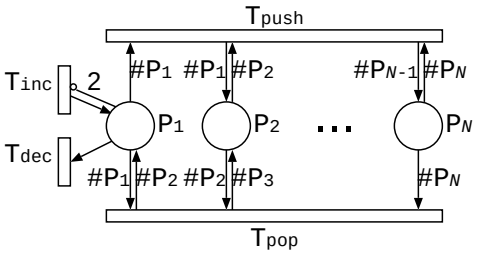


Fig. 7. The stack model

The Courier model describes Courier protocol software [37] (using $N = M$), and the FMS model describes a flexible manufacturing system [14]. Both of these models contain immediate events, and thus cannot be analyzed by [11]. In addition, the FMS model contains events that do not satisfy the Kronecker product requirement of [11] (i.e., we do not consider a simplified version of the model, as done in [28]).

Finally, to illustrate the limitations of our approach, we consider a stack model, shown in Fig. 7, where the stack holds only the integers $\{0, 1, 2\}$ and has depth N. Since each element of the stack can have value 0, 1, or 2, there are exactly 3^N reachable states. Events T_{inc} and T_{dec} affect only component 1. Events T_{push} and

T_{pop}, however, are examples of worst–case behavior for our approach and also the approach of [11], as they affect every component, and do so in such a way that the next–state function cannot be broken into a cross–product. Thus, using our approach, construction of the next–state functions for events T_{push} and T_{pop} requires explicit enumeration of all reachable states. The results in Table I confirm this expectation, and overall execution time is comparable to explicit reachability set generation algorithms. The memory requirements for the next–state function stored as MxDs (not shown in the table) are quite large for this model, around 300 Mb for $N = 15$ (the largest for the other models was around 2 Mb). To satisfy the Kronecker product requirement of [11], transitions T_{push} and T_{pop} must be split into separate events for each reachable state. A Kronecker product representation for each of the $2+2{\cdot}3^N$ events has a much higher storage cost than the MxD representation (about 19 Mb versus 414 Kb for $N = 9$). Since [11] considers each event separately during saturation, many temporary MDD nodes are created, added to compute tables, and destroyed, leading to a much longer execution time.

B. Discussion

For models "Kanban", "FTMP", and "Swaps", Table I shows that our pregeneration approach performs better than our on-the-fly approach. In these cases, pregeneration constructs the same local state spaces as on-the-fly, and the difference in computation time is due to the overhead of reconstructing the next–state MxDs during updates. The increased compute table size for on-the-fly is due to the additional table $enabledTable$. We also see that our approaches perform as good or better than the approach of [11]; this is especially true for models whose events require splitting to satisfy the Kronecker product requirement. Indeed, the strength of our approach is that events are essentially grouped together into K "macro events", where the MxD for each $\mathcal{N}_k$ is able to exploit similarities between the grouped events.

However, for models "Courier" and "FMS", we instead see that the on-the-fly approach significantly outperforms the pregeneration approach. The primary reason for this is that the pregeneration approach constructs local state spaces that are slightly too large (since event priority is not considered during their construction), leading to unnecessarily larger and more complex MxDs. Complexity is exacerbated due to the corrections for priority.

Finally, we note that, to use the pregeneration approach it was necessary to modify models "Kanban", "Courier", and "FMS" (by hand, via addition of inhibitor arcs, implicit places, etc.) so that the pregeneration approach would obtain finite local state spaces.

VII. Related work

Early implicit approaches for reachability set construction (e.g., [7]), used BDDs to represent sets of states and BDDs to represent a single next–state function. The generation algorithm was a simple, iterative, breadth–first search: iteration n discovers the set of states reachable in n steps from the set of initial states; these are added to the overall reachability set using a set union operation. Generation is complete when the reachability set converges, which is trivial to check thanks to the canonicity of BDDs.

Several variations of this approach have been proposed in the literature; we mention only a few here. For many models, the memory bottleneck was due to storage of the next–state function; an approach to address this is to partition the next–state function to save space [6]. The saturation approach can be seen as a special case of disjunctive partitioning: the overall next–state function is partitioned into next–state functions based on the top–most affected component, $\mathcal{N} = \mathcal{N}_K \cup \cdots \cup \mathcal{N}_1$. However, this partition is motivated not by space savings, but instead by computational savings.

The use of Kronecker algebra for asynchronous systems was first proposed in [33] for performance evaluation, where a Kronecker product is used for each event for the representation of large Markov chains, rather than next–state functions (see [5] for an in–depth discussion). Kronecker algebra was utilized for explicit reachability set generation in [23], and were adapted for use with implicit reachability set generation with MDDs in [30]. A similar approach, which constructs BDD next–state functions by event, and exploits the asynchronous nature of events, is discussed in [25].

Another contribution of [30] was the use of an iteration order different from straightforward breadth–first search, which exploited the Kronecker representation: events which affect only one model component k were repeatedly applied to level–k MDD nodes, until a fixed–point was obtained. This approach builds on the idea of *chaining* [34], in which newly–discovered states are explored as soon as possible, rather than at the end of the current iteration. A similar algorithm was used in [18] for models that do not use a Kronecker representation. Finally, [30] noted that events that do not affect a given component will produce an identity matrix in the Kronecker representation, which can be exploited during reachability set generation.

The idea of optimizing for local events in [30] has been used elsewhere (e.g., in [4] for explicit reacha-

bility set generation), and was further refined in the implicit setting in [9]; additional refinements produced the saturation algorithm [10]. The use of Kronecker representations for saturation leads to limitations which have been gradually eliminated: [11] built the next–state function and reachability set simultaneously, [28] adapted saturation for use with immediate events ([16] also dealt with immediate events, but not with saturation). Finally, this article, which is an extension of [29], eliminates the requirement of saturation that events have a Kronecker product form.

VIII. Conclusion

We presented modifications to the saturation algorithm that allow its application to models whose events do not satisfy the Kronecker product requirement. We considered both pregeneration and on-the-fly construction of local state spaces and next–state functions. As expected, when pregeneration is able to construct exactly the same local state spaces and next–state functions, it performs better than on-the-fly; however, on-the-fly can perform significantly better than pregeneration in cases where the local state spaces cannot be accurately constructed in isolation.

While the original Kronecker product version of saturation can be applied to models whose events do not satisfy the Kronecker product requirement, doing so requires re-defining the model by splitting events into sub-events that are expressible as Kronecker products. Our experiments indicate that the use of matrix diagrams instead can substantially reduce computation time in these cases. Furthermore, our algorithms are competitive with original saturation, even for models that satisfy the Kronecker product requirement.

Our new approach is based on expressing the next–state function for each model event as a cross product of functions that operate on only a subset of components. The worst case occurs when a model event function affects every component and cannot be broken into a cross product; while our approach still applies in this case, the explicit construction of the next–state function makes our approach about as efficient as explicit generation techniques. The Kronecker product version of saturation performs much worse in this case.

References

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

[2] R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, Sept. 1992.

[3] P. Buchholz. Hierarchical structuring of superposed GSPNs. In *Proc. PNPM'97*, pages 81–90, Saint Malo, France, June 1997.

[4] P. Buchholz and P. Kemper. Efficient computation and representation of large reachability sets for composed automata. *Discrete Event Dynamic Systems: Theory and Applications*, 12:265–286, 2002.

[5] P. Buchholz and P. Kemper. Kronecker based matrix representations for large markov models. In *Validation of Stochastic Systems*, LNCS 2925, pages 256–295. Springer-Verlag, 2004.

[6] J. Burch, E. Clarke, and D. Long. Symbolic model checking with partitioned transition relations. In *Int. Conf. on VLSI*, pages 49–58, Edinburgh, Scotland, Aug. 1991. IFIP Transactions, North-Holland.

[7] 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.

[8] G. Ciardo, R. Jones, A. Miner, and R. Siminiceanu. Logical and stochastic modeling with SMART. In *Proc. TOOLS 2003*, LNCS 2794, pages 78–97, Urbana, IL, USA, Sept. 2003. Springer-Verlag.

[9] G. Ciardo, G. Luettgen, and R. Siminiceanu. Efficient symbolic state-space construction for asynchronous systems. In *Proc. ICATPN 2000*, LNCS 1825, pages 103–122, Aarhus, Denmark, June 2000. Springer-Verlag.

[10] G. Ciardo, G. Luettgen, and R. Siminiceanu. Saturation: an efficient iteration strategy for symbolic state space generation. In *Proc. TACAS 2001*, LNCS 2031, pages 328–342, Genova, Italy, Apr. 2001. Springer-Verlag.

[11] G. Ciardo, R. Marmorstein, and R. Siminiceanu. Saturation unbound. In *Proc. TACAS 2003*, LNCS 2619, pages 379–393, Warsaw, Poland, Apr. 2003. Springer-Verlag.

[12] G. Ciardo and A. S. Miner. A data structure for the efficient Kronecker solution of GSPNs. In *Proc. PNPM'99*, pages 22–31, Zaragoza, Spain, Sept. 1999. IEEE Comp. Soc. Press.

[13] G. Ciardo and R. Siminiceanu. Using edge–valued decision diagrams for symbolic generation of shortest paths. In *Proc. FMCAD 2002*, LNCS 2517, pages 256–273, Portland, OR, USA, Nov. 2002. Springer-Verlag.

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

[15] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[16] I. Davies, W. J. Knottenbelt, and P. S. Kritzinger. Symbolic methods for the state space exploration of GSPN models. In *Proc. TOOLS 2002*, LNCS 2324, pages 188–199, London, UK, Apr. 2002. Springer-Verlag.

[17] M. Davio. Kronecker products and shuffle algebra. *IEEE Trans. Comp.*, C-30(2):116–125, Feb. 1981.

[18] S. Derisavi, P. Kemper, and W. H. Sanders. Symbolic state–space exploration and numerical analysis of state-sharing composed models. *Linear Algebra and Its Applications*, 386:137–166, July 2004.

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

[20] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall International, 1985.

[21] 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.

[22] S. Katz and D. Peled. An efficient verification method for parallel and distributed programs. In *Workshop on Linear Time, Branching Time and Partial Order Logics and Models for Concurrency*, LNCS 354, pages 489–507. Springer, 1988.

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

[24] P. Kemper. Reachability analysis based on structured representations. In *Proc. ICATPN 1996*, LNCS 1091, pages 269–288, Osaka, Japan, June 1996. Springer-Verlag.

[25] K. Lampka and M. Siegle. MTBDD-based activity-local state graph generation. In *PMCCS-6*, pages 15–18, Sept. 2003.

[26] A. Miner and D. Parker. Symbolic representations and analysis of large state spaces. In *Validation of Stochastic Systems*, LNCS 2925, pages 296–338. Springer-Verlag, 2004.

[27] A. S. Miner. Efficient solution of GSPNs using Canonical Matrix Diagrams. In *Proc. PNPM'01*, pages 101–110, Aachen, Germany, Sept. 2001. IEEE Comp. Soc. Press.

[28] A. S. Miner. Implicit GSPN reachability set generation using decision diagrams. *Perf. Eval.*, 56(1-4):145–165, Mar. 2004.

[29] A. S. Miner. Saturation for a general class of models. In *Proc. QEST'04*, pages 282–291, Enschede, The Netherlands, Sept. 2004.

[30] A. S. Miner and G. Ciardo. Efficient reachability set generation and storage using decision diagrams. In *Proc. ICATPN 1999*, LNCS 1639, pages 6–25, Williamsburg, VA, USA, June 1999. Springer-Verlag.

[31] T. Murata. Petri nets: Properties, analysis and applications. *Proc. IEEE*, 77(4):541–580, Apr. 1989.

[32] E. Pastor, O. Roig, J. Cortadella, and R. Badia. Petri net analysis using boolean manipulation. In *Proc. ICATPN 1994*, LNCS 815, pages 416–435, Zaragoza, Spain, June 1994. Springer-Verlag.

[33] B. Plateau. On the stochastic structure of parallelism and synchronisation models for distributed algorithms. In *Proc. SIGMETRICS 1985*, pages 147–153, Austin, TX, USA, May 1985.

[34] O. Roig, J. Cortadella, and E. Pastor. Verification of asynchronous circuits by BDD–based model checking of Petri nets. In *Proc. ICATPN 1995*, LNCS 935, pages 374–391, Turin, Italy, June 1995. Springer-Verlag.

[35] W. H. Sanders and L. M. Malhis. Dependability evaluation using composed SAN–based reward models. *J. Par. and Distr. Comp.*, 15(3):238–254, July 1992.

[36] A. Valmari. A stubborn attack on state explosion. In *Proc. CAV 1990*, LNCS 531, pages 156–165. Springer, 1990.

[37] C. M. Woodside and Y. Li. Performance Petri net analysis of communications protocol software by delay-equivalent aggregation. In *Proc. PNPM'91*, pages 64–73, Melbourne, Australia, Dec. 1991. IEEE Comp. Soc. Press.

Andrew S. Miner received the B.S. degree in Computer Science and Physics from Randolph–Macon College, Ashland, VA in 1993, and received the M.S. and Ph.D. degrees in Computer Science from the College of William and Mary, Williamsburg, VA in 1995 and 2000, respectively. He is currently an Assistant Professor in the Department of Computer Science at Iowa State University, Ames, IA. His current research interests include model checking, stochastic modeling, and performance evaluation of asynchronous systems. He is a co-designer and developer of the software tool SMART (Stochastic Model checking Analyzer for Reliability and Timing).