Saturation Unbound*

Gianfranco Ciardo, Robert Marmorstein, and Radu Siminiceanu

College of William and Mary, Williamsburg, Virginia 23187
{ciardo,rmmarm,radu}@cs.wm.edu

Abstract. In previous work, we proposed a "saturation" algorithm for symbolic state-space generation characterized by the use of multi-valued decision diagrams, boolean Kronecker operators, event locality, and a special iteration strategy. This approach outperforms traditional BDD-based techniques by several orders of magnitude in both space and time but, like them, assumes a priori knowledge of each submodel's state space. We introduce a new algorithm that merges explicit local state-space discovery with symbolic global state-space generation. This relieves the modeler from worrying about the behavior of submodels in isolation.

1 Introduction

Since their introduction, implicit methods for symbolic model checking, such as decision diagrams, in particular BDDs [5,6,8], have been enormously successful. However, the systems targeted have been mainly synchronous VLSI designs and protocols, where the possible values of each state variable can be easily determined a priori. For arbitrary systems modeled in a high-level formalism such as Petri nets or pseudocode, determining the range of the state variables is more difficult. Traditionally, the burden of this task has been placed on the user. In NuSMV [14], for example, the domain of multi-valued variables must be explicitly specified as a set or integer range. In our own previous work [11,12,23], the input (a Petri net) must be partitioned so that the state space of each "local subnet" can be generated in isolation. This practice requires careful addition of inhibitor arcs or other constructs to ensure correct local behavior without affecting the global behavior, a difficult and error-prone endeavor.

In this paper, we address this problem with an algorithm that produces a multi-valued decision diagram (MDD) [21] representation of the final state-space and a separately stored representation of the "minimal" local state spaces. The algorithm interleaves explicit local exploration of each submodel with symbolic exploration of the global state space. The new algorithm is based on our *saturation* algorithm [12], which uses an MDD to store the global states and boolean Kronecker matrices to encode the transition relation. By using a *disjunctively-partitioned transition relation* [20], exploiting *event locality* [23], performing *in-place updates* [11] of MDD nodes, and using an innovative iteration strategy,

* Work supported in part by the National Aeronautics and Space Administration under grants NAG-1-2168 and NAG-1-02095 and by the National Science Foundation under grants CCR-0219745 and ACI-0203971.

H. Garavel and J. Hatcliff (Eds.): TACAS 2003, LNCS 2619, pp. 379–393, 2003.

saturation showed massive time and space improvements over traditional symbolic state-space generation approaches. Our new algorithm exploits these same ideas, but can be applied to a more general class of models. The inherent asynchronicity of software models makes them ideal candidates for our new approach.

Section 2 gives the necessary background on symbolic state-space generation and the rationale behind our contribution. Section 3 presents our new algorithm. We discuss theoretical and practical issues of design and implementation in Section 4, and provide experimental results in Section 5. Section 6 discusses related work and Section 7 concludes by suggesting further research directions.

2 State Space Generation

A discrete-state model is a triple $(\widehat{\mathcal{S}}, \mathbf{s}, \mathcal{N})$, where $\widehat{\mathcal{S}}$ is the set of *potential states* of the model, $\mathbf{s}$ is the *initial state* (a set of initial states could be easily handled), and $\mathcal{N} : \widehat{\mathcal{S}} \to 2^{\widehat{\mathcal{S}}}$ is the *next-state function* specifying which states can be reached from a given state in a single step. Since we target asynchronous systems, we partition $\mathcal{N}$ into a union of next-state functions [20]: $\mathcal{N}(s) = \bigcup_{e \in \mathcal{E}} \mathcal{N}_e(s)$, where $\mathcal{E}$ is a finite set of *events*, $\mathcal{N}_e$ is the next-state function associated with event e, i.e., $\mathcal{N}_e(s)$ is the set of states the system can enter when e occurs, or *fires*, in state s. An event e is said to be *disabled* in s if $\mathcal{N}_e(s) = \emptyset$; otherwise, it is *enabled*.

The *reachable state space* $\mathcal{S} \subseteq \widehat{\mathcal{S}}$ is the smallest set containing $\mathbf{s}$ and closed with respect to $\mathcal{N}$: $\mathcal{S} = \{\mathbf{s}\} \cup \mathcal{N}(\mathbf{s}) \cup \mathcal{N}(\mathcal{N}(\mathbf{s})) \cup \cdots = \mathcal{N}^*(\mathbf{s})$, where "$*$" denotes the reflexive and transitive closure, and we let $\mathcal{N}(\mathcal{X}) = \bigcup_{s \in \mathcal{X}} \mathcal{N}(s)$. Thus, $\mathcal{S}$ is the smallest fixed point of the equation $\mathcal{S} = \mathcal{N}(\mathcal{S})$ in which $\mathcal{S}$ contains $\{\mathbf{s}\}$. Since $\mathcal{N}$ is composed of several functions $\mathcal{N}_e$, we can build $\mathcal{S}$ by applying each function in any order, as long as we consider each event often enough [17].

The systems we target can be partitioned into interacting submodels. For a model composed of K *submodels*, a *global* system state is a K-tuple $(i_K, \ldots, i_1)$, where i_k is the *local* state of submodel k, for $K \geq k \geq 1$. Thus, the potential state space $\widehat{\mathcal{S}}$ is given by the cross-product $\mathcal{S}_K \times \cdots \times \mathcal{S}_1$ of K *local* state spaces. For example, the places of a Petri net can be partitioned into K subsets, with the marking written as a vector of the K corresponding submarkings. Partitioning enables us to use techniques targeted at exploiting system structure, including *symbolic* state space storage techniques based on decision diagrams.

Multi-valued decision diagrams for the state space. If each local state space $\mathcal{S}_k$ is known, we can use mappings $\psi_k : \mathcal{S}_k \to \{0, 1, \ldots, n_k - 1\}$, where $n_k = |\mathcal{S}_k|$, and encode $\mathcal{S} \subseteq \widehat{\mathcal{S}}$ via a (*quasi-reduced ordered*) MDD, i.e., a directed acyclic edge-labeled multi-graph where:

- Nodes are organized into $K + 1$ *levels*. We write $\langle k{:}p \rangle$ to denote a generic node, where k is the level and p is a unique index for that level.
- Level K contains only a single *non-terminal* node $\langle K{:}r \rangle$, the *root*, whereas levels $K-1$ through 1 contain one or more non-terminal nodes.
- Level 0 consists of two *terminal* nodes, $\langle 0{:}0 \rangle$ and $\langle 0{:}1 \rangle$.

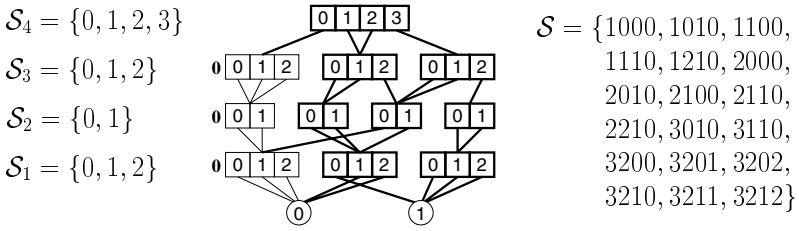

Fig. 1. An example MDD and the state space $\mathcal{S}$ encoded by it.

– A non-terminal node $\langle k{:}p\rangle$ has n_k arcs pointing to nodes at level $k{-}1$. If the i^{th} arc, for $i \in \mathcal{S}_k$, is to node $\langle k{-}1{:}q\rangle$, we write $\langle k{:}p\rangle[i] = q$. *Duplicate* nodes are not allowed but, unlike *reduced ordered* MDDs, *redundant* nodes where all arcs point to the same node are allowed (both versions are canonical).

Let $\mathcal{B}(\langle k{:}p\rangle)$ be the set of (sub-)states encoded by the MDD rooted at $\langle k{:}p\rangle$. As in [12], we reserve the node index 0 at each level k, so that $\mathcal{B}(\langle k{:}0\rangle) = \emptyset$. In previous work, we also reserved the index 1, so that $\mathcal{B}(\langle k{:}1\rangle) = \mathcal{S}_k \times \cdots \mathcal{S}_1$, but as discussed in Section 4, this optimization can no longer be used. Fig. 1 shows a four-level MDD and the set $\mathcal{S}$ it encodes. The lightly bordered nodes represent the empty set and are not explicitly stored in the MDD.

Kronecker encoding for the next-state function. Effective symbolic state space generation requires an efficient encoding of the next-state function. Unlike BDD approaches, where $\mathcal{N}$, or each $\mathcal{N}_e$, is encoded in a $2K$-level BDD, we adopt a Kronecker representation inspired by work on Markov chains [2,7,25]. As in [11,12,23], we use a *consistent* model partition, where each $\mathcal{N}_e$ is decomposed into K local next-state functions $\mathcal{N}_{e,k}$, for $K \geq k \geq 1$, which satisfy

$$\forall (i_K, \ldots, i_1) \in \widehat{\mathcal{S}},\ \mathcal{N}_e(i_K, \ldots, i_1) = \mathcal{N}_{e,K}(i_K) \times \cdots \times \mathcal{N}_{e,1}(i_1).$$

By defining matrices $\mathbf{W}_{e,k} \in \{0,1\}^{n_k \times n_k}$, where $\mathbf{W}_{e,k}[i_k, j_k] = 1 \Leftrightarrow j_k \in \mathcal{N}_{e,k}(i_k)$, the next-state function is encoded as the incidence matrix given by the (boolean) sum of Kronecker products $\sum_{e \in \mathcal{E}} \bigotimes_{K \geq k \geq 1} \mathbf{W}_{e,k}$. The $\mathbf{W}_{e,k}$ matrices are extremely sparse (for standard Petri nets, each row contains at most one nonzero entry), and are indexed using the same mapping ψ_k used to index $\mathcal{S}_k$.

In addition to efficiently representing $\mathcal{N}$, the Kronecker encoding allows us to exploit event *locality* [11,23] and employ *saturation* [12]. Locality means that most events depend on few submodels, hence few levels of the MDD (event e is *independent* of level k if $\mathbf{W}_{e,k} = \mathbf{I}$, the identity). If we let $Top(e)$ and $Bot(e)$ denote the highest and lowest levels on which e depends, respectively, the saturation strategy iterates until node $\langle k{:}p\rangle$ has reached a fixed point with respect to all $\mathcal{N}_e$ such that $Top(e) \leq k$, collectively written as $\mathcal{N}_{\leq k}$, without examining the nodes above $\langle k{:}p\rangle$ in the MDD.

Local state spaces. Traditional symbolic state-space generation assumes a priori knowledge of the local state spaces. For BDDs, each $\mathcal{S}_k$ is simply $\{0, 1\}$; in our previous MDD work [11,12,23], each $\mathcal{S}_k$ is finite and built prior to state-space generation. Since $\mathcal{S}_k$ is known, we can store its elements in a search structure,

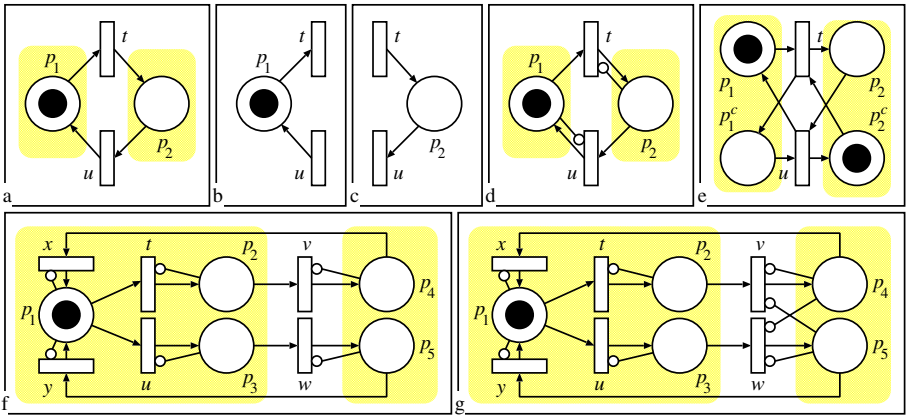

Fig. 2. Local state spaces built in isolation can be strict supersets of the actual ones.

which encodes the mapping ψ_k. Local states in the MDD may then be referenced exclusively through their integer indices in ψ_k.

For a given high-level formalism, each $\mathcal{S}_k$ can be *pregenerated* with an explicit traversal of the local state-to-state transition graph of the submodel obtained by considering all the variables associated with the submodel and all the events affecting those variables. Unfortunately, this pregeneration may create spurious local states. For example, if the two places of the Petri net in Fig. 2(a) are partitioned into two subsets, the corresponding subnets, (b) and (c), have unbounded local state spaces when considered in isolation. In subnet (b), transition u can keep adding tokens to place p_1 since, without the input arc from p_2, u is always locally enabled. Hence, in isolation, p_1 may contain arbitrarily many tokens. The same can be said for subnet (c). However, $\mathcal{S} = \{(1,0),(0,1)\}$, so we would ideally like to define $\mathcal{S}_2 = \mathcal{S}_1 = \{0,1\}$. This can be enforced by adding either inhibitor arcs, (d), or complementary places, (e). Consider now the Petri net of Fig. 2(f), partitioned into two subnets, one containing p_1, p_2, and p_3, the other containing p_4 and p_5. The inhibitor arcs shown avoid unbounded local state spaces in isolation, but they don't ensure that the local state spaces are as small as possible. For example, the local state space built in isolation for the subnet containing p_4 and p_5 is $\{(0,0),(1,0),(0,1),(1,1)\}$, while only the first three states are actually reachable in the overall net, since p_4 and p_5 can never contain a token at the same time. This is corrected in (g) by adding two more inhibitor arcs, from p_5 to v and from p_4 to w. An analogous problem exists for the other subnet as well, and correcting it with inhibitor arcs is even more cumbersome.

Thus, there are two problems with pregeneration: a local state space in isolation might be unbounded (causing pregeneration to fail) or contain spurious states (causing inefficiencies in the symbolic state-space generation). Asking the modeler to cope with this by adding constraints to the original model (e.g., the inhibitor arcs in Fig. 2) is at best burdensome, since it requires a priori knowledge of $\mathcal{S}$, the output of state-space generation, and at worst dangerous, since doing so might "mask" undesirable behaviors that are present in the overall model.

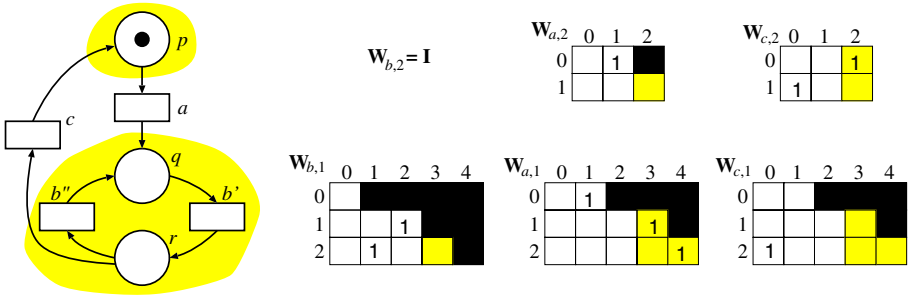

Fig. 3. An example snapshot of transition matrices $\mathbf{W}_{e,k}$.

3 Local State Spaces with Unknown Bounds

We now describe an *on-the-fly* algorithm that merges explicit local state with symbolic global state explorations and builds the *smallest* local state spaces $\mathcal{S}_k$ needed to encode the correct global state space $\mathcal{S} \subseteq \widehat{\mathcal{S}} = \mathcal{S}_K \times \cdots \times \mathcal{S}_1$. Ideally, the additional time spent exploring local state spaces on-the-fly should be comparable to that spent in the pregeneration phase of our old algorithm. This is, in fact, the case for our new algorithm, which incrementally discovers a set $\widehat{\mathcal{S}}_k$ of *locally-reachable* local states, of which only a subset $\mathcal{S}_k$ is also globally reachable. When our algorithm determines that a local state is globally reachable, it labels it as *confirmed*. Since *unconfirmed* local states in $\widehat{\mathcal{S}}_k \setminus \mathcal{S}_k$ are limited to a "rim" around the confirmed ones, and since unconfirmed states do not increase the size of MDD nodes, memory overhead is small in practice.

Expanding local state spaces. Initially, for $K \geq k \geq 1$, $\widehat{\mathcal{S}}_k = \mathcal{S}_k = \{\mathbf{s}_k\}$, the k^{th} component of the initial state $\mathbf{s}$. The iteration strategy of [12] saturates nodes bottom-up through an exhaustive symbolic reachability analysis: it fires globally enabled events on a node as long as new global states are found. In our new version, the MDD encodes only confirmed states, but our Kronecker encoding describes all possible transitions from confirmed local states to both confirmed and unconfirmed local states. Thus, we only explore *global symbolic firings* originating in confirmed states. The next-state function for event e in local state i of node $\langle k{:}p \rangle$ (Fig. 4, lines 9 of *Saturate* and 9 of *RecFire*) may lead to a state $j \in \mathcal{S}_k$ or $j \in \widehat{\mathcal{S}}_k \setminus \mathcal{S}_k$. In the former case, j is already confirmed and row j of $\mathbf{W}_{e,k}$ has been built (thus the local states reachable from j are in $\widehat{\mathcal{S}}_k$). In the latter case, j is unconfirmed: it is locally, but not necessarily globally, reachable, thus it appears as a column index but has no corresponding row in $\mathbf{W}_{e,k}$. Local state j will be confirmed if the global symbolic firing that used the entry $\mathbf{W}_{e,k}[i,j]$ is actually possible, i.e., if e can fire in an MDD path from $Top(e)$ to $Bot(e)$ through node $\langle k{:}p \rangle$. Only when j is confirmed, its rows in $\mathbf{W}_{e,k}$ (for all events e that depend on k) are built, using one forward step of *explicit local reachability analysis*. This step must consult the description of the model itself, and thus works on actual submodel variables, not state indices. This is the only operation that may discover new unconfirmed local states.

Transition matrices. The on-the-fly algorithm uses "rectangular" Kronecker matrices over $\{0,1\}^{\mathcal{S}_k \times \widehat{\mathcal{S}}_k}$ where only confirmed local states "know" their successors, confirmed or not.

Fig. 3 shows an example. The net is partitioned into subnet 2, containing place p, and subnet 1, with places q and r. The potential local state spaces discovered with the saturation algorithm are $\widehat{\mathcal{S}}_2 = \{0 \equiv (p^1), 1 \equiv (p^0), 2 \equiv (p^2)\}$, and $\widehat{\mathcal{S}}_1 = \{0 \equiv (q^0r^0), 1 \equiv (q^1r^0), 2 \equiv (q^0r^1), 3 \equiv (q^2r^0), 4 \equiv (q^1r^1)\}$. The model events are grouped and ordered as dictated by the saturation strategy: transitions that are local to subnet 1, b' and b", are merged into macroevent b, the remaining two being the "synchronizing" events a and c. There are two Kronecker matrices for each event, $\mathbf{W}_{e,k}$, $e \in \{a,b,c\}$, $k \in \{1,2\}$, hence six in total, among which $\mathbf{W}_{b,2}$ is the identity, since macroevent b does not affect subnet 2.

Three regions of the matrices are highlighted: the white portion corresponds to moves between confirmed states and those are kept in the final matrices; the shaded area corresponds to moves from confirmed to unconfirmed states; the black region means that the corresponding column indices were out of range at the time the row was built. The actual evolution of the matrices is:

1. The initial state is inserted, with index 0, in each local state space. The row for local state 0 of $\widehat{\mathcal{S}}_1$ in each of the corresponding three matrices is built explicitly: only event a leads to a new local state (q^1r^0), which is indexed 1. Similarly at level 2, a new local state (p^0) is discovered by locally firing a and another, (p^2), by locally firing c, these are indexed 1 and 2, respectively.
2. MDD saturation starts; in the initial MDD node at level 1, no events are enabled, but at level 2, event a is successfully (globally) fired, leading to global state $(1,1) \equiv (p^0, q^1r^0)$. As a result, local states 1 at both levels are confirmed. Their corresponding rows are built with an explicit exploration in each subnet. At level 1, local state 1 can move to $2 \equiv (q^0r^1)$ by firing the b' component of b, or to $3 \equiv (q^2r^0)$ by locally firing a. At level 2, only event c can fire in local state 1, leading to local state $0 \equiv (p^1)$.
3. Event b is globally fired resulting in the confirmation of local state 2 of $\widehat{\mathcal{S}}_1$ as globally reachable. Its row is built in $\mathbf{W}_{b,1}, \mathbf{W}_{a,1}, \mathbf{W}_{c,1}$. During this phase, one more local state is discovered, $4 \equiv (q^1r^1)$, corresponding to a potential firing of event a in state 2, as well as the already confirmed states 1, by firing the b" component of b, and 0 by firing c.
4. Event c is globally fired from global state $(1,2) \equiv (p^0, q^0r^1)$, leading to global state $(0,0) \equiv (p^1, q^0r^0)$, whose local states are both confirmed, hence no explicit exploration is needed for either of them.
5. Saturation ends, with the final local state spaces $\mathcal{S}_2 = \{0,1\}$, $\mathcal{S}_1 = \{0,1,2\}$ and global state-space $\mathcal{S} = \{(0,0),(1,1),(1,2)\}$. The matrices are trimmed to their final "square" shape: 2×2 at level 2, and 3×3 at level 1, by discarding the unconfirmed local states along with their corresponding columns.

The algorithm. The pseudocode of our new algorithm, in Fig. 4, uses the data types *ev* (model event), *local* (local state), *level*, and *idx* (node index within

GenerateSS():*idx*

Build the MDD encoding $\mathcal{N}^*_{\mathcal{E}}(\mathbf{s})$.

declare p,r:*idx*, k:*level*, s:*state*,i:*local*;
1. $p \leftarrow 1$;
2. for $k = 1$ to K do
3. $s \leftarrow InitialState(k)$;
4. $i \leftarrow InsertState(k, s)$;
5. $Confirm(k, i)$;
6. $r \leftarrow NewNode(k)$;
7. $\langle k{:}r\rangle[i] \leftarrow p$;
8. $Saturate(k, r)$;
9. $CheckIn(k, r)$;
10. $p \leftarrow r$;
11. return r; • $\langle K{:}r\rangle$ *is the MDD root*

Saturate(in k:*level*, p:*idx*)

Update $\langle k{:}p\rangle$, to encode $\mathcal{N}^*_{\leq k}(\mathcal{B}(\langle k{:}p\rangle))$.

declare $\mathcal{L}$:set of *local*, i,j,j':*local*;
declare e:*ev*, f,u:*idx*; $pChng$:*bool*;
1. repeat
2. $pChng \leftarrow false$;
3. foreach $e \in \mathcal{E}$ s.t. $Top(e) = k$ do
4. $\mathcal{L} \leftarrow Locals(e, k, p)$;
5. while $\mathcal{L} \neq \emptyset$ do
6. pick and remove i from $\mathcal{L}$;
7. $f \leftarrow RecFire(e, k{-}1, \langle k{:}p\rangle[i])$;
8. if $f \neq 0$ then
9. foreach j s.t. $\mathbf{W}_{e,k}[i, j]{=}1$ do
10. $u \leftarrow Union(k{-}1, f, \langle k{:}p\rangle[j])$;
11. if $u{\neq}\langle k{:}p\rangle[j]$ then
12. if $j \notin \mathcal{S}_k$ then $Confirm(k, j)$;
13. $\langle k{:}p\rangle[j] \leftarrow u$; $pChng \leftarrow true$;
14. if $\exists j', \mathbf{W}_{e,k}[j, j'] = 1$ then
15. $\mathcal{L} \leftarrow \mathcal{L}{\cup}\{j\}$;
16. until $pChng = false$;

Union(in k:*level*, p:*idx*, q:*idx*):*idx*

Build the MDD for $\mathcal{B}(\langle k{:}p\rangle) \cup \mathcal{B}(\langle k{:}q\rangle)$.

1. if $p = 0$ or $p = q$ then return q;
2. if $q = 0$ then return p;
3. if $Find(UC[k], \{p, q\}, s)$ then return s;
4. $s \leftarrow NewNode(k)$;
5. for $i = 0$ to $|\mathcal{S}_k| - 1$
6. $u \leftarrow Union(k{-}1, \langle k{:}p\rangle[i], \langle k{:}q\rangle[i])$;
7. $\langle k{:}s\rangle[i] \leftarrow u$;
8. $CheckIn(k, s)$;
9. $Insert(UC[k], \{p, q\}, s)$;
10. return s;

RecFire(in e:*ev*, l:*level*, q:*idx*):*idx*

Build the MDD for $\mathcal{N}^*_{\leq l}(\mathcal{N}_e(\mathcal{B}(\langle l{:}q\rangle)))$.

declare $\mathcal{L}$:set of *local*, i,j:*local*;
declare f,u,s:*idx*, $sChng$:*bool*;
1. if $l < Bot(e)$ then return q;
2. if $Find(FC[e, l], q, s)$ then return s;
3. $s \leftarrow NewNode(l)$; $sChng \leftarrow false$;
4. $\mathcal{L} \leftarrow Locals(e, l, q)$;
5. while $\mathcal{L} \neq \emptyset$ do
6. pick and remove i from $\mathcal{L}$;
7. $f \leftarrow RecFire(e, l{-}1, \langle l{:}q\rangle[i])$;
8. if $f \neq 0$ then
9. foreach j s.t. $\mathbf{W}_{e,l}[i, j] = 1$ do
10. $u \leftarrow Union(l{-}1, f, \langle l{:}s\rangle[j])$;
11. if $u{\neq}\langle l{:}s\rangle[j]$ then
12. if $j \notin \mathcal{S}_l$ then $Confirm(l, j)$;
13. $\langle l{:}s\rangle[j] \leftarrow u$; $sChng \leftarrow true$;
14. if $sChng$ then $Saturate(l, s)$;
15. $CheckIn(l, s)$;
16. $Insert(FC[e, l], q, s)$;
17. return s;

Confirm(in k:*level*, i:*local*)

Add i to $\mathcal{S}_k$ and build its rows in all $\mathbf{W}_{e,k}$ where e depends on submodel k.

declare e:*ev*, j:*local*, n:*int*, s,u:*state*;
1. $s \leftarrow GetState(k, i)$;
2. foreach e dependent on submodel k
3. foreach $u \in NextState(e, k, s)$
4. $j \leftarrow InsertState(k, u)$;
5. $\mathbf{W}_{e,k}[i, j] \leftarrow 1$;
6. $\mathcal{S}_k \leftarrow \mathcal{S}_k \cup \{i\}$;

Locals(in e:*ev*, k:*level*, p:*idx*):set of *local*

Local indices of $\langle k{:}p\rangle$ locally enabling e: $\{i{\in}\mathcal{S}_k{:}\langle k{:}p\rangle[i]{\neq}0 \wedge \exists j, \mathbf{W}_{e,k}[i, j] = 1\}$.

CheckIn(in k:*level*, inout p:*idx*)

If $\forall i \in \mathcal{S}_k, \langle k{:}p\rangle[i] = 0$, delete $\langle k{:}p\rangle$ and set p to 0. If $\langle k{:}p\rangle$ duplicates $\langle k{:}q\rangle$, delete $\langle k{:}p\rangle$ and set p to q. Else, insert $\langle k{:}p\rangle$ in the unique table $UT[k]$.

NewNode(in k:*level*):*idx*

Create a new MDD node at level k, with its arcs set to 0.

Fig. 4. Pseudocode for the on-the-fly version of our saturation algorithm [12].

a level); in practice, these are simply integers in appropriate ranges. In addition, the model-specific type *state* represents the explicit description of a local state. We assume the existence of the following dynamically-sized global hash tables: $UT[k]$, for $K \geq k \geq 1$, the *unique table* for nodes at level k, to retrieve p given $\langle k{:}p\rangle[0], \ldots, \langle k{:}p\rangle[n_k-1]$; $UC[k]$, for $K > k \geq 1$, the *union cache* for nodes at level k, to retrieve s given p and q, where $\mathcal{B}(\langle k{:}s\rangle) = \mathcal{B}(\langle k{:}p\rangle) \cup \mathcal{B}(\langle k{:}q\rangle)$; and $FC[e,k]$, for $Top(e) > k \geq Bot(e)$, the *firing cache* for event e and nodes at level k, to retrieve s given node p, where $\mathcal{B}(\langle k{:}s\rangle) = \mathcal{N}^*_{\leq k}(\mathcal{N}_e(\mathcal{B}(\langle k{:}p\rangle)))$, where $\mathcal{N}_e(i_k, \ldots, i_1)$ is to be interpreted as $\mathcal{N}_{e,k}(i_k) \times \cdots \times \mathcal{N}_{e,1}(i_1)$. Hash-table access is provided by *Insert*(*key*,*val*), which associates *val* to *key*, and *Find*(*key*,*res*), which returns *true* if *key* is in the cache and sets *res* to the associated value. Furthermore, we use K dynamically-sized arrays to store nodes, so that $\langle k{:}p\rangle$ can be efficiently retrieved as the p^{th} entry of the k^{th} array. The two-dimensional array of sparse matrices $\mathbf{W}_{e,k}$ is a global variable. We also assume a global variable of type *model* with the following functional interface: $InsertState(level, state) : local$, $GetState(level, local) : state$, $NextState(ev, level, state) : state$, and $InitialState(level) : state$.

4 Comments

We now discuss some details of the on-the-fly algorithm, especially the challenges raised by its design and implementation.

MDD nodes of variable size. With pregeneration, the size of MDD nodes at level k is fixed, easing their creation, deletion, and reuse. With our on-the-fly expansion of $\mathcal{S}_k$, we might instead allocate nodes of increasing size. Fortunately, a saturated node $\langle k{:}p\rangle$ remains saturated when a new state i is added to $\mathcal{S}_k$, as long as the semantic of the missing arc $\langle k{:}p\rangle[i]$ is taken to be an arc to $\langle k{-}1{:}0\rangle$. However, the growth of $\mathcal{S}_k$ implies that we cannot reserve a special meaning for node $\langle k{:}1\rangle$, as done with pregeneration [12], where $\mathcal{B}(\langle k{:}1\rangle) = \mathcal{S}_k \times \cdots \times \mathcal{S}_1$. In [12], this optimization could speed-up the computation whenever node $\langle k{:}1\rangle$ is involved in a union, since we could immediately conclude that $\mathcal{B}(\langle k{:}1\rangle) \cup \mathcal{B}(\langle k{:}p\rangle) = \mathcal{B}(\langle k{:}1\rangle)$. Indeed, such nodes were not even stored explicitly in [12]. To reserve index 1 for the same purpose with the on-the-fly approach is problematic, since, whenever we increase $\mathcal{S}_l$, for $l \leq k$, the meaning of $\mathcal{B}(\langle k{:}1\rangle)$ implicitly changes. We can still reserve index 0 for the empty set, and exploit the relation $\mathcal{B}(\langle k{:}0\rangle) \cup \mathcal{B}(\langle k{:}p\rangle) = \mathcal{B}(\langle k{:}p\rangle)$, since the representation of the empty set is the same as that in [12].

This observation led us to use *quasi-reduced* (instead of *reduced*) MDDs. The latter eliminates redundant nodes and is potentially more efficient, but its arcs can span multiple levels. As discussed in [11], such arcs are more difficult to manage and can yield a slower state-space generation when exploiting locality. With the on-the-fly algorithm, they create an even worse problem: they become "incorrect" when a local state space grows. For example, both the reduced and the quasi-reduced 3-level MDDs in Fig. 5(a) and (c) encode the state space $\mathcal{S} = \{(0,0,2),(0,1,2),(1,0,2),(1,1,2),(3,0,2)\}$, when $\mathcal{S}_3 = \{0,1,2,3\}$, $\mathcal{S}_2 = \{0,1\}$, and $\mathcal{S}_1 = \{0,1,2\}$. If we want to add global state $(3,2,2)$ to $\mathcal{S}$, we need to add

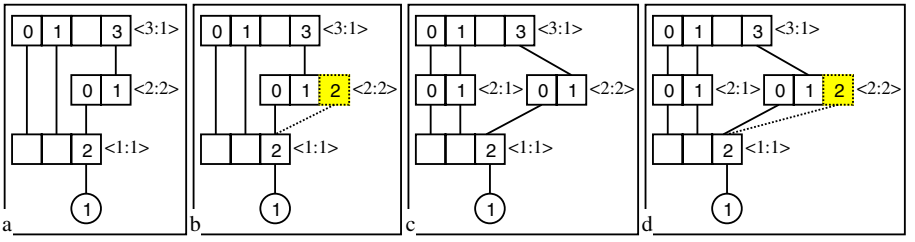

Fig. 5. Before and after, with reduced (a-b) vs. quasi-reduced (c-d) MDDs.

local state 2 to $\mathcal{S}_2$ and set arc $\langle 2{:}2\rangle[2]$ to $\langle 1{:}1\rangle$. However, while the resulting quasi-reduced MDD in (d) is correct, the reduced one in (b) is not, since now it also encodes global states $(0, 2, 2)$ and $(1, 2, 2)$. To fix the problem, we could reintroduce the formerly-redundant node $\langle 2{:}1\rangle$ so that the new reduced and quasi-reduced MDDs coincide. While it would be possible to modify the MDD to obtain a correct reduced ordered MDD in this manner whenever a local state space grows, the cost of doing so is unjustifiably high.

Compacting the arc arrays. To facilitate dynamically-sized nodes, we use indirection. Node $\langle k{:}p\rangle$ is associated with entry p of array $nodes_k$, as we need direct access to it, given p. However, the arcs of all nodes at level k are stored together in a second array $arcs_k$. Entry $nodes_k[p]$ has fixed size and stores, among other information, fields bg, the beginning location where the "chunk" of arcs for $\langle k{:}p\rangle$ begins in $arcs_k$, and sz, the chunk's size, since n_k changes over time. Saturation "updates-in-place" a node $\langle k{:}p\rangle$ when exploring the firing of an event e such that $Top(e) = k$. Fortunately, this implies that only the currently unsaturated node at level k (there is at most one such node at any point in time) might cause $\mathcal{S}_k$ to grow: if we store the arcs of this node as the last chunk in $arcs_k$, we never need to grow any "internal" chunk.

However, saturated nodes may still become obsolete and need to be recycled. Deleting $\langle k{:}p\rangle$ leaves a "hole" of size $nodes_k[p].sz$ in $arcs_k$. Since these holes may be smaller than the chunks needed for new nodes, we cannot easily reuse them as-is. Instead, we mark them as invalid and periodically clean them by "compacting to the left" the entire $arcs_k$ array. The space opened at the right end of $arcs_k$ can then be reused for new nodes. Shifting chunks of arcs requires that the values $nodes_k[p].bg$ pointing to these chunks be updated as well. There are several ways to do this. Our implementation stores, together with the chunk for $\langle k{:}p\rangle$ in $arcs_k$, a back pointer to the node itself, i.e., the value p. This allows us to compact $arcs[k]$ via a linear scan: valid values are shifted to the left over invalid values until all holes have been removed. Simultaneously, using the back pointer p, we access and update the value $nodes_k[p].bg$ after shifting the corresponding chunk. With respect to our pregeneration implementation where arc chunks are stored directly in $nodes_k$ as arrays of fixed size n_k, this requires 12 additional bytes per node, for bg and sz in $nodes_k$ and for the back pointer in $arcs_k$. However, it can also save memory, since chunks can be smaller than for pregeneration and could be stored with sparse techniques.

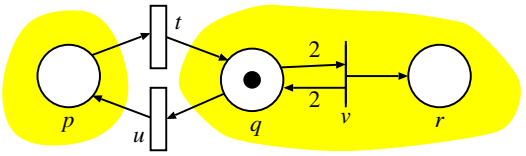

Fig. 6. Potential but not actual overflow of a local state space.

We keep track of the number of invalid entries, and trigger compaction whenever a portion of the entries in $arcs_k$ become invalid (50% in our experiments).

Overflowing of potential local state spaces. Our new algorithm eliminates the need to specify additional constraints for any formalism where each state can reach a finite number of states in a single step. A subtle problem remains, however, if an infinite number of states can be reached in one step. For example, in Generalized Stochastic Petri Nets [1], *immediate* transitions, such as v in Fig. 6, are processed not by themselves, but as events that can take place instantaneously after the firing of *timed* transitions, such as t and u (somewhat analogous to *internal* events in process algebra). In Fig. 6, we partition the net into subnet 2, containing place p, and subnet 1, containing places q and r. The initial local states are (p^0) and (q^1r^0) respectively. When the latter state is confirmed into $\mathcal{S}_1$, an explicit local exploration begins. Transition t can fire in subnet 1 in isolation, leading to marking (q^2r^0). This enables immediate transition v which, processed right away as part of the firing of t, leads to markings (q^2r^1), (q^2r^2), (q^2r^3), ... and so on. Thus, the explicit local exploration fails with an overflow in place r, while a traditional explicit global exploration would not, since it would never reach a global marking with two tokens in q. This situation is quite artificial, however. It can occur only if the formalism allows a state to reach an infinite number of states in one "timed step".

BDDs vs. MDDs. Our choice of MDDs over BDDs for modeling asynchronous systems is prompted by the success of the saturation strategy, which exploits event locality and gives flexibility in the choice of partitioning. MDDs are an excellent match for our Kronecker encoding of the next-state function; indeed, the two, in conjunction with a "good" partition, allow us to *increase* the amount of locality. However, the issues addressed in our work are not exclusive to MDDs.

It is true that determining the potential state space for BDDs "simply" means deciding the number K of boolean variables. However, when the actual model has integer variables, the modeler needs to know how many bits are needed for their encoding, and this is just as difficult as determining the size of our local state spaces a priori. Just as with our MDDs, then, choosing too many bits can lead to inefficiencies, choosing too few can mask errors. An on-the-fly algorithm for BDDs would have to expand the decision diagrams "vertically", creating more levels of binary variables to cope with a growing range for an integer variable, while MDDs can simply extend local state spaces "horizontally". Furthermore, in our approach, unconfirmed local states appear only in the columns of the Kronecker matrices encoding the next-state function, not in the MDD.

Table 1. Generation of the state space: On-The-Fly vs. PREgeneration vs. NuSMV

N	Reachable states	Final memory (KB)			Peak memory (KB)			Time (sec)		
		OTF	PRE	NuSMV	OTF	PRE	NuSMV	OTF	PRE	NuSMV
Dining Philosophers: $K=N$, $\|\mathcal{S}_k\|=34$ for all k										
20	3.46×10^{12}	4	3	4,178	5	4	4,192	0.01	0.01	0.4
50	2.23×10^{31}	11	10	8,847	14	12	8,863	0.03	0.02	13.1
100	4.97×10^{62}	24	20	8,891	28	25	15,256	0.06	0.05	990.8
200	2.47×10^{125}	48	40	21,618	57	50	59,423	0.15	0.11	18,129.3
5,000	6.53×10^{3134}	1,210	1,015	—	1,445	1,269	—	65.55	51.29	—
Slotted Ring Network: $K=N$, $\|\mathcal{S}_k\|=15$ for all k										
5	5.39×10^{4}	1	1	502	5	5	507	0.01	0.01	0.1
10	8.29×10^{9}	5	5	4,332	28	27	8,863	0.06	0.04	6.1
15	1.46×10^{15}	10	9	771	80	77	11,054	0.18	0.13	2,853.1
100	2.60×10^{105}	434	398	—	15,753	14,486	—	41.72	25.78	—
Round Robin Mutual Exclusion: $K=N+1$, $\|\mathcal{S}_k\|=10$ for all k except $\|\mathcal{S}_1\|=N+1$										
10	2.30×10^{4}	5	5	917	6	7	932	0.01	0.01	0.2
20	4.72×10^{7}	18	17	5,980	20	21	5,985	0.04	0.03	1.4
30	7.25×10^{10}	37	36	2,222	41	41	8,716	0.09	0.07	5.6
100	2.85×10^{32}	357	355	13,789	372	372	21,814	2.11	1.55	2,836.5
150	4.82×10^{47}	784	781	—	807	807	—	7.04	5.07	—
FMS: $K=19$, $\|\mathcal{S}_k\|=N+1$ for all k except $\|\mathcal{S}_{17}\|=4, \|\mathcal{S}_{12}\|=3, \|\mathcal{S}_7\|=2$										
5	1.92×10^{4}	5	6	2,113	6	9	2,126	0.01	0.01	1.0
10	2.50×10^{9}	16	19	1,152	26	31	8,928	0.02	0.02	41.6
25	8.54×10^{13}	86	135	17,045	163	239	152,253	0.16	0.11	17,321.9
150	4.84×10^{23}	6,291	15,459	—	16,140	29,998	—	18.50	10.92	—

5 Results

We compare the space and runtime required by our new algorithm with those of its pregeneration predecessor [12] and of NuSMV [14], a symbolic verifier built on top of the CUDD library [27]. We use a 2.4 Ghz Pentium IV with 1GB of memory. Our examples include four models from [12], parametrized by an integer N: dining philosophers, slotted ring, round robin mutual exclusion, and flexible manufacturing system (FMS). The first three models are safe Petri nets: N affects the height of the MDD but not the size of the local state spaces (except for $\mathcal{S}_1$ in the round robin model, which grows linearly in N). The FMS has instead an MDD with fixed height but size of the nodes increasing linearly in N. In the pregeneration and NuSMV models, additional constraints are manually imposed to ensure that the correct local state spaces are built in isolation.

Table 1 lists the peak and final memory and the runtime for these algorithms. For comparison's sake, we assume that a BDD node in NuSMV uses 16 bytes. To be fair, we point out that our memory consumption refers to the MDD only, while we believe that the number of nodes reported by NuSMV includes also those for the next-state function; however, our Kronecker encoding for $\mathcal{N}$ is extremely efficient, requiring at most 300KB in any model, except for the model with 5,000 dining philosophers, where it requires 5.2MB. The memory

for the operation caches is not included in either our or NuSMV results (for our algorithms, caches never exceeded 20MB on these examples). Both the on-the-fly and the pregeneration MDD algorithms are able to handle significantly larger models than NuSMV. We show the largest value of N for each of the four models where generation was possible, in the penultimate row for NuSMV and the last row for SMART. When comparisons with NuSMV can be made, our algorithms show speed-up ratios over 100,000 and memory reduction ratios over 1,000.

The results demonstrate that the overhead of the on-the-fly algorithm versus pregeneration is acceptable. Moreover, the additional 12-byte per node memory overhead required to manage dynamically-sized nodes at a given level k can be offset by the ability to store nodes with $m < n_k$ arcs (because they were created when $\mathcal{S}_k$ contained only m states, or because the last $n_k - m$ arcs point to $\langle k-1{:}0 \rangle$ and are truncated). In fact, for the FMS model, this results in smaller memory requirements than with pregeneration, suggesting that the use of sparse nodes is advantageous in models with large local state spaces. Even if our on-the-fly implementation is not yet as optimized as that of pregeneration, the runtime of the on-the-fly algorithm is still excellent, at most 70% over that of pregeneration. This is a good tradeoff, given the increase in modeling ease and safety afforded by not having to worry about local state space generation in isolation.

6 Related Work

Symbolic analysis of unbounded discrete-event systems has been considered before in a more general setting than ours: in most cases, the goal is the study of systems with infinite but regular state spaces. For example, the Queue BDDs of [18] allow one to model systems with a finite number of boolean variables plus one or more unbounded queues, as long as the contents of the queue can be represented by a DFA. The MONA system [19], implementing *monadic second-order logic*, can be used to verify *parametric* systems without relying on a proof by induction. These types of approach can be generally classified under the umbrella of *regular model checking* [4].

Our goal in this paper is more modest, since we only target models with a finite state space, but it is also very different. The saturation approach we introduced in [12] has been shown to be vastly superior to traditional breadth-first approaches for globally-asynchronous locally-synchronous systems (see also [13] for its application to edge-valued decision diagrams). However, it was limited to models for which bounds on the state variables are known a priori. Here, we extended it to bounded models with unknown bounds, such as those arising when modeling distributed software, one of the most challenging problems in symbolic methods. Our description formalism of choice, Petri nets with inhibitor arcs, self-modifying behavior, and non-deterministic decision, is Turing-equivalent, thus we can only hope that the state space of a given model is finite. Even when it is finite, though, it can still have a highly "irregular" pattern. It is in these patterns that the efficiency of our approach is particularly desirable. To our knowledge, the algorithm we presented is the first to combine symbolic generation of the

global state space with exact explicit generation of the local state spaces, but the issue is related, at least for Petri nets, to the existence of invariants [24]. Indeed, we could use *place invariants* to bound our local state spaces and proceed using pregeneration, but our on-the-fly approach is superior because it has a small overhead, while invariant analysis is very expensive in pathological cases [22]. More importantly, the invariant approach is limited: a net might not be fully covered by invariants yet be bounded because of inhibitor arcs or other constructs; invariant analysis alone might suggest that a place is unbounded because it concludes that a transition t can keep "pumping" tokens into it, while in reality t might never be enabled given the particular initial marking. Indeed, invariant analysis does not take into account the presence of inhibitor arcs, and it can deal only with a limited class of marking-dependent arc cardinalities used in *self-modifying nets* [9], yet both constructs are very useful in practice to define compact and realistic models. Our solution is fast, general, user-friendly, and terminates in at least all cases where previous algorithms terminate.

The idea of a disjunctively-partitioned transition relation is natural for asynchronous systems [15,20], and in particular for Petri nets. However, our inspiration for its (boolean) Kronecker encoding comes from the field of Markov chains, where (real) Kronecker operators are increasingly used to encode the infinitesimal generators of large Markov models described compositionally [2,7,25]. Thanks to this encoding, we can exploit the presence of locality in the transition relation of each individual event, achieving much greater efficiency.

With regard to the saturation approach itself, several works proposed abandoning the breadth-first approach of traditional symbolic state-space generation, with the goal of reducing the peak number of nodes, but they often still have some vestiges of breadth-first search. For example, [3,26] improve efficiency by exploring only a portion of the newly-found states, those encoded by the "densest" nodes in the decision diagram. The closest to our saturation approach is the "modified breadth-first search" mentioned in [20]. We believe that our saturation approach is the first one to fully avoid any flavor of a global breadth-first iteration and, for globally-asynchronous locally-synchronous models, its locality-based ordering of the events appears to be much more effective, at times even optimal: cases where the peak number of nodes is only $O(1)$ larger than the final number of nodes were reported in [12].

7 Conclusion and Future Work

Traditional symbolic state-space generation approaches require a priori knowledge of the range of the state variables. In practice this puts a considerable burden on the modeler who must add artificial constraints and may risk masking actual errors in the original model. In this paper, we presented a new approach that avoids this requirement by integrating explicit local state-space generation with symbolic global state-space generation, and building a Kronecker representation of the next-state function "on-the-fly". Our algorithm provides this new capability at a small overhead cost with respect to our previous "pregeneration"

algorithm, but it remains enormously more efficient than other approaches in which the next-state function is encoded as a BDD.

We stress that the data structure we employ, MDDs whose nodes can be expanded at runtime, may have useful applications beyond state-space generation. On the other hand, while our saturation algorithm is very efficient, it should be clear that its use is not a prerequisite to the on-the-fly exploration of the local state spaces we presented; however, we showed how the two can be seamlessly integrated, and this enhances the usefulness and applicability of saturation.

In the future, we plan to investigate reordering, splitting, and merging of MDD levels. These are more general than the reordering of BDD variables, but we hope to extend some heuristics already known for that problem [16]. Also, our algorithms are currently implemented in SMART [10], a simulation and modeling tool for logic and stochastic analysis. Eventually, we intend to make them available as C++ libraries.

References

1. M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with Generalized Stochastic Petri Nets*. John Wiley & Sons, New York, 1995.
2. V. Amoia, G. De Micheli, and M. Santomauro. Computer-oriented formulation of transition-rate matrices via Kronecker algebra. *IEEE Trans. Rel.*, 30:123–132, June 1981.
3. R. Bloem, K. Ravi, and F. Somenzi. Symbolic guided search for CTL model checking. In *Proc. 37th Conf. on Design Automation*, p. 29–34. ACM Press, 2000.
4. A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular model checking. In *Computer Aided Verification*, pages 403–418, 2000.
5. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comp.*, 35(8):677–691, Aug. 1986.
6. R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comp. Surv.*, 24(3):393–318, 1992.
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, 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. In *Proc. 5th Annual IEEE Symp. on Logic in Computer Science*, pages 428–439, Philadelphia, PA, 4–7 June 1990. IEEE Comp. Soc. Press.
9. G. Ciardo. Petri nets with marking-dependent arc multiplicity: properties and analysis. In R. Valette, editor, *Proc. 15th Int. Conf. on Applications and Theory of Petri Nets*, LNCS 815, pages 179–198, Zaragoza, Spain, June 1994. Springer-Verlag.
10. G. Ciardo, R. L. Jones, A. S. Miner, and R. Siminiceanu. SMART: Stochastic Model Analyzer for Reliability and Timing. In P. Kemper, editor, *Tools of Int. Multiconference on Measurement, Modelling and Evaluation of Computer-Communication Systems*, pages 29–34, Aachen, Germany, Sept. 2001.
11. G. Ciardo, G. Luettgen, and R. Siminiceanu. Efficient symbolic state-space construction for asynchronous systems. In M. Nielsen and D. Simpson, editors, *Proc. 21th Int. Conf. on Applications and Theory of Petri Nets*, LNCS 1825, pages 103–122, Aarhus, Denmark, June 2000. Springer-Verlag.

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 and R. Siminiceanu. Using edge-valued decision diagrams for symbolic generation of shortest paths. In M. D. Aagaard and J. W. O'Leary, editors, *Proc. Fourth International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, LNCS 2517, pages 256–273, Portland, OR, USA, Nov. 2002. Springer-Verlag.
14. A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: A new symbolic model verifier. In *CAV '99*, LNCS 1633, pages 495–499. Springer-Verlag, 1999.
15. O. Coudert and J. C. Madre. Symbolic computation of the valid states of a sequential machine: algorithms and discussion. In *1991 Int. Workshop on Formal Methods in VLSI Design*, pages 1–19, Miami, FL, USA, 1991.
16. M. Fujita, H. Fujisawa, and Y. Matsunaga. Variable ordering algorithms for ordered binary decision diagrams and their evaluation. *IEEE Trans. on Computer–Aided Design of Integrated Circuits and Systems*, 12(1):6–12, 1993.
17. A. Geser, J. Knoop, G. Lüttgen, B. Steffen, and O. Rüthing. Chaotic fixed point iterations. Technical Report MIP-9403, Univ. of Passau, 1994.
18. P. Godefroid and D. E. Long. Symbolic protocol verification with queue BDDs. *Formal Methods in System Design*, 14(3):257–271, May 1999.
19. J. G. Henriksen, J. L. Jensen, M. E. Jørgensen, N. Klarlund, R. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In E. Brinksma, R. Cleaveland, K. G. Larsen, T. Margaria, and B. Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1019, pages 89–110. Springer, 1995.
20. J.R. Burch, E.M. Clarke, and D.E. Long. Symbolic model checking with partitioned transition relations. In A. Halaas and P.B. Denyer, editors, *Int. Conference on Very Large Scale Integration*, pages 49–58, Edinburgh, Scotland, Aug. 1991. IFIP Transactions, North-Holland.
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. J. Martinez and M. Silva. A simple and fast algorithm to obtain all invariants of a generalised Petri net. In *Proc. 2nd European Workshop on Application and Theory of Petri Nets*, pages 411–422, Bad Honnef, Germany, 1981.
23. A. S. Miner and G. Ciardo. Efficient reachability set generation and storage using decision diagrams. In H. Kleijn and S. Donatelli, editors, *Proc. 20th Int. Conf. on Applications and Theory of Petri Nets*, LNCS 1639, pages 6–25, Williamsburg, VA, USA, June 1999. Springer-Verlag.
24. T. Murata and R. Church. Analysis of marked graphs and Petri nets by matrix equations. Research report MDC 1.1.8, Department of information engineering, Univeristy of Illinois, Chicago, IL, Nov. 1975.
25. B. Plateau. On the stochastic structure of parallelism and synchronisation models for distributed algorithms. In *Proc. ACM SIGMETRICS*, pages 147–153, Austin, TX, USA, May 1985.
26. K. Ravi and F. Somenzi. Efficient fixpoint computation for invariant checking. In *Proc. Int. Conference on Computer Design (ICCD)*, pages 467–474, Austin, TX, Oct. 1999. IEEE Comp. Soc. Press.
27. F. Somenzi. CUDD: CU Decision Diagram Package, Release 2.3.1. http://vlsi.colorado.edu/~fabio/CUDD/cuddIntro.html.