NASA/CR-2001-210663
ICASE Report No. 2001-5

Saturation: An Efficient Iteration Strategy for Symbolic State-space Generation

Gianfranco Ciardo
The College of William & Mary, Williamsburg, Virginia

Gerald Lüttgen
The University of Sheffield, Sheffield, United Kingdom

Radu Siminiceanu
The College of William & Mary, Williamsburg, Virginia

ICASE
NASA Langley Research Center
Hampton, Virginia
Operated by Universities Space Research Association

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

Prepared for Langley Research Center
under Contract NAS1-97046

February 2001

SATURATION: AN EFFICIENT ITERATION STRATEGY FOR SYMBOLIC STATE–SPACE GENERATION[*]

GIANFRANCO CIARDO[†], GERALD LÜTTGEN[‡], AND RADU SIMINICEANU[†]

Abstract. This paper presents a novel algorithm for generating state spaces of asynchronous systems using Multi–valued Decision Diagrams. In contrast to related work, the next–state function of a system is not encoded as a single Boolean function, but as cross–products of integer functions. This permits the application of various iteration strategies to build a system's state space. In particular, this paper introduces a new elegant strategy, called *saturation*, and implements it in the tool SMART. On top of usually performing several orders of magnitude faster than existing BDD–based state–space generators, the algorithm's required peak memory is often close to the final memory needed for storing the overall state spaces.

Key words. iteration strategy, multi–valued decision diagrams, saturation, state–space generation

Subject classification. Computer Science

1. Introduction.

State–space generation is one of the most fundamental challenges for many formal verification tools, such as model checkers [16]. The high complexity of today's digital systems requires constructing and storing huge state spaces in the relatively small memory of a workstation. One research direction widely pursued in the literature suggests the use of *decision diagrams*, usually Binary Decision Diagrams [8] (BDDs), as a data structure for implicitly representing large sets of states in a compact fashion. This proved to be very successful for the verification of synchronous digital circuits, as it increased the manageable sizes of state spaces from about 10^6 states, with traditional explicit state–space generation techniques [18, 19], to about 10^{20} states [10]. Unfortunately, symbolic techniques are known not to work well for *asynchronous systems*, such as communication protocols, which suffer from state–space explosion.

The latter problem was addressed in previous work by the authors in the context of state–space generation using *Multi–valued Decision Diagrams* [25] (MDDs), which exploited the fact that, in event–based asynchronous systems, each event updates just a few components of a system's state vector [11]. Hence, firing an event requires only the application of *local next–state functions* and the local manipulation of MDDs. This is in contrast to classic BDD–based techniques which construct state spaces by iteratively applying a single, global next–state function which is itself encoded as a BDD [28]. Additionally, in most concurrency frameworks including Petri nets [31] and process algebras [5], next–state functions satisfy a *product form* allowing each component of the state vector to be updated somewhat independently of the others. Experimental results implementing these ideas of locality showed significant improvements in speed and memory consumption when compared to other state–space generators [30].

[*]This work was supported by the National Aeronautics and Space Administration under NASA Contract No. NAS1–97046 while the authors were in residence at ICASE, NASA Langley Research Center, Hampton, VA 23681–2199, USA. G. Ciardo and R. Siminiceanu were also partially supported by NASA Grant No. NAG–1–2168.

[†]Department of Computer Science, P.O. Box 8795, College of William and Mary, Williamsburg, VA 23187–8795, USA, email: {ciardo, radu}@cs.wm.edu.

[‡]Department of Computer Science, The University of Sheffield, 211 Portobello Street, Sheffield S1 4DP, U.K., email: g.luettgen@dcs.shef.ac.uk.

In this paper we take our previous approach a significant step further by observing that the reachable state space of a system can be built by firing the system's events in any order, as long as every event is considered often enough [21]. We exploit this freedom by proposing a novel strategy which exhaustively fires all events affecting a given MDD node, thereby bringing it to its final *saturated* shape. Moreover, nodes are considered in a depth–first fashion, i.e., when a node is processed, all its descendants are already saturated. The resulting state–space generation algorithm is not only concise, but also allows for an elegant proof of correctness. Compared to our previous work [11], saturation eliminates much administration overhead, reduces the average number of firing events, and enables a simpler and more efficient cache management.

We implemented the new algorithm in the tool SMART [12], and experimental studies indicate that it performs on average about one order of magnitude faster than our old algorithm and several orders of magnitude faster than other existing state–space generators [11]. Even more important and in contrast to related work, the peak memory requirements of our algorithm are often close to its final memory requirements. In the case of the well–known dining philosophers' problem, we are able to construct the associated state space of about 10^{627} states, for 1000 philosophers, in under 1 second on a 800 MHz Pentium III PC using only 390KB of memory. Our results imply that future state–based verification tools will be able to handle much larger asynchronous systems than is currently possible and will also provide faster feedback to engineers.

The remainder of this paper is organized as follows. The next section introduces our formal framework and notation, including MDDs. Section 3 then presents our idea of node saturation as well as our novel state–space generation algorithm. Some implementation details are discussed in Section 4, and our algorithm is evaluated in Section 5 by applying it to a suite of asynchronous system models. Finally, related work is surveyed in Section 6, while Section 7 contains our conclusions and directions for future work.

2. MDDs for Encoding Structured State Spaces.

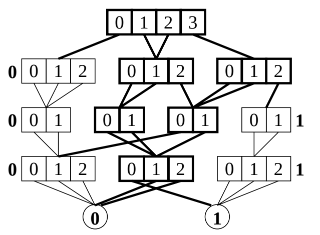
A discrete–state system model expressed in a high–level formalism must specify three objects: (i) $\widehat{\mathcal{S}}$, the set of *potential states* describing the "type" of states; (ii) $\mathbf{s} \in \widehat{\mathcal{S}}$, the *initial state* of the system; and (iii) $\mathcal{N} : \widehat{\mathcal{S}} \longrightarrow 2^{\widehat{\mathcal{S}}}$, the *next–state function*, describing which states can be reached from a given state in a single system step. In many cases, such as Petri nets and process algebras, a model expresses this function as a union $\mathcal{N} = \bigcup_{e \in \mathcal{E}} \mathcal{N}_e$, where $\mathcal{E}$ is a finite set of *events* and $\mathcal{N}_e$ is the next–state function associated with event e. We say that $\mathcal{N}_e(s)$ is the set of states the system can enter when event e occurs, or *fires*, in state s. Moreover, event e is called *disabled* in s if $\mathcal{N}_e(s) = \emptyset$; otherwise, it is *enabled*.

The *reachable state space* $\mathcal{S} \subseteq \widehat{\mathcal{S}}$ of the model under consideration is the smallest set containing the initial system state $\mathbf{s}$ and being closed with respect to $\mathcal{N}$, i.e., $\mathcal{S} = \{\mathbf{s}\} \cup \mathcal{N}(\mathbf{s}) \cup \mathcal{N}(\mathcal{N}(\mathbf{s})) \cup \cdots = \mathcal{N}^*(\mathbf{s})$, where "$*$" denotes reflexive and transitive closure. When $\mathcal{N}$ is composed of several functions $\mathcal{N}_e$, for $e \in \mathcal{E}$, we can iterate these functions in any order, as long as we consider each $\mathcal{N}_e$ often enough. This results in "chaotic" fixed point iterations, which are known to yield the desired fixed point, i.e., the reachable state space $\mathcal{S}$ [21]. In other words, $i \in \mathcal{S}$ if and only if it can be reached from $\mathbf{s}$ through zero or more event firings. In this paper we assume that $\mathcal{S}$ is finite; however, for most practical asynchronous systems, the size of $\mathcal{S}$ is enormous due to the *state–space explosion* problem.

2.1. Multi–valued Decision Diagrams.

One way to cope with this problem is to use efficient data structures to encode $\mathcal{S}$. This is usually possible when the system has some *structure*. We consider the common case in asynchronous system design, where a system model is composed of K *submodels*, for some $K \in \mathbb{N}$, so that a global system state is a K–tuple $(i^K, \ldots, i^1)$, where i^k is the local state for submodel k. (We

$\mathcal{S}^4 = \{0,1,2,3\}$
$\mathcal{S}^3 = \{0,1,2\}$
$\mathcal{S}^2 = \{0,1\}$
$\mathcal{S}^1 = \{0,1,2\}$



$\mathcal{S} = \{1000, 1010, 1100, 1110, 1210, 2000, 2010, 2100, 2110, 2210, 3010, 3110, 3200, 3201, 3202, 3210, 3211, 3212\}$

FIG. 2.1. *An example MDD and the state space $\mathcal{S}$ encoded by it.*

use *superscripts* for submodel indexes —not for exponentiation— and *subscripts* for event indexes.) Thus, $\widehat{\mathcal{S}} = \mathcal{S}^K \times \cdots \times \mathcal{S}^1$, with each *local* state space $\mathcal{S}^k$ having some finite size n^k. In Petri nets, for example, the set of places can be partitioned into K subsets, and the marking can be written as the composition of the K corresponding submarkings. When identifying $\mathcal{S}^k$ with the initial integer interval $\{0, \ldots, n^k - 1\}$, for each $K \geq k \geq 1$, one can encode $\mathcal{S} \subseteq \widehat{\mathcal{S}}$ via a (*quasi–reduced ordered*) MDD, i.e., a directed acyclic 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, $\langle 0.\mathbf{0} \rangle$ and $\langle 0.\mathbf{1} \rangle$. (We use boldface for indexes 0 or 1 because they have a special meaning, as we will explain later.)
- 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 $\langle k.p \rangle[i] = q$. Unlike in the original BDD setting [8, 9], we allow for *redundant* nodes, having all arcs pointing to the same node. This will be convenient for our purposes, as eliminating such nodes would lead to arcs spanning multiple levels.
- A non–terminal node cannot *duplicate* (i.e., have the same pattern of arcs as) another node at the same level.

Given a node $\langle k.p \rangle$, we can recursively define the node reached from it through any integer sequence $\gamma =_{\text{df}} (i^k, i^{k-1}, \cdots, i^l) \in S^k \times S^{k-1} \times \cdots \times S^l$ of length $k - l + 1$, for $K \geq k \geq l \geq 1$, as

$$node(\langle k.p \rangle, \gamma) = \begin{cases} \langle k.p \rangle & \text{if } \gamma = (), \text{ the empty sequence} \\ node(\langle k-1.q \rangle, \delta) & \text{if } \gamma = (i^k, \delta) \text{ and } \langle k.p \rangle[i^k] = q. \end{cases}$$

The substates encoded by p or reaching p are then, respectively,

$$\begin{aligned} \mathcal{B}(\langle k.p \rangle) &= \{\beta \in \mathcal{S}^k \times \cdots \times \mathcal{S}^1 : node(\langle k.p \rangle, \beta) = \langle 0.\mathbf{1} \rangle\} && \text{“}below\text{” } \langle k.p \rangle\,; \\ \mathcal{A}(\langle k.p \rangle) &= \{\alpha \in \mathcal{S}^K \times \cdots \times \mathcal{S}^{k+1} : node(\langle K.r \rangle, \alpha) = \langle k.p \rangle\} && \text{“}above\text{” } \langle k.p \rangle\,. \end{aligned}$$

Thus, $\mathcal{B}(\langle k.p \rangle)$ contains the substates that, prefixed by a substate in $\mathcal{A}(\langle k.p \rangle)$, form a (global) state encoded by the MDD. We reserve the indexes 0 and 1 at each level k to encode the sets $\emptyset$ and $\mathcal{S}^k \times \cdots \times \mathcal{S}^1$, respectively. In particular, $\mathcal{B}(\langle 0.\mathbf{0} \rangle) = \emptyset$ and $\mathcal{B}(\langle 0.\mathbf{1} \rangle) = \{()\}$. Figure 2.1 shows a four–level example MDD and the set $\mathcal{S}$ encoded by it; only the highlighted nodes are actually stored.

Many algorithms for generating the state space $\mathcal{S}$ using BDDs have been proposed [28], and adapting them to MDDs is straightforward. However, a key difference in our new approach is that we do *not* encode the next–state function as an MDD over $2K$ variables, recording the K state components before and after a system step. Instead, we explicitly and efficiently update MDD nodes directly, adding the new states reached through one step of the global next–state function when firing a given event. For asynchronous system models, this function is often expressible as the cross–product of local next–state functions.

2.2. Product–form Behavior. An asynchronous system model exhibits such behavior if, for each event e, its next–state function $\mathcal{N}_e$ can be written as a cross–product of K local functions, i.e., $\mathcal{N}_e = \mathcal{N}_e^K \times \cdots \times \mathcal{N}_e^1$ where $\mathcal{N}_e^k : \mathcal{S}^k \longrightarrow 2^{\mathcal{S}^k}$, for all $K \geq k \geq 1$. This requirement is quite natural for two reasons. First, many modeling formalisms satisfy it, e.g., any Petri net model conforms to this behavior for any partition of its places. Second, if a given model does not respect the product–form behavior, we can always coarsen K or refine $\mathcal{E}$ so that it does. As an example, consider a model partitioned into four submodels, where $\mathcal{N}_e = \mathcal{N}_e^4 \times \mathcal{N}_e^{3,2} \times \mathcal{N}_e^1$, but $\mathcal{N}^{3,2} : \mathcal{S}^3 \times \mathcal{S}^2 \longrightarrow 2^{\mathcal{S}^3 \times \mathcal{S}^2}$ cannot be expressed as a product $\mathcal{N}_e^3 \times \mathcal{N}_e^2$. We can achieve the product–form requirement by simply partitioning the model into three, not four, submodels. Alternatively, we may substitute event e with "subevents" satisfying the product form. This is possible since, in the worst case, we can define a subevent $e_{i,j}$, for each $i = (i^3, i^2)$ and $j = (j^3, j^2) \in \mathcal{N}_e^{3,2}(i)$, with $\mathcal{N}_{e_{i,j}}(i^3) = \{j^3\}$ and $\mathcal{N}_{e_{i,j}}(i^2) = \{j^2\}$. Of course, carrying this argument too far leads to explicit representations, where $K = 1$ or where every state–to–state transition corresponds to a different event. However, this did not happen in the numerous asynchronous systems we considered in our studies.

Finally, we introduce some notational conventions. We say that event e *depends* on level k, if the local state at level k does affect the enabling of e or if it is changed by the firing of e. Let $\mathit{First}(e)$ and $\mathit{Last}(e)$ be the first and last levels on which event e depends. Events e such that $\mathit{First}(e) = \mathit{Last}(e) = k$ are said to be *local events* and can be merged into a single *macro–event* λ^k without violating the product–form requirement, since one can write $\mathcal{N}_{\lambda^k} = \mathcal{N}_{\lambda^k}^K \times \cdots \times \mathcal{N}_{\lambda^k}^1$ where $\mathcal{N}_{\lambda^k}^k = \bigcup_{\{e:\mathit{First}(e)=\mathit{Last}(e)=k\}} \mathcal{N}_e^k$, while $\mathcal{N}_{\lambda^k}^l(i^l) = \{i^l\}$ for $l \neq k$ and $i^l \in \mathcal{S}^l$. The set $\{e \in \mathcal{E} : \mathit{First}(e) = k\}$ of events "starting" at level k is denoted by $\mathcal{E}^k$. We also extend $\mathcal{N}_e$ to substates instead of full states: $\mathcal{N}_e((i^k, \ldots, i^l)) = \mathcal{N}_e^k(i^k) \times \cdots \times \mathcal{N}_e^l(i^l)$, for $K \geq k \geq l \geq 1$; to sets of states: $\mathcal{N}_e(\mathcal{X}) = \bigcup_{i \in \mathcal{X}} \mathcal{N}_e(i)$, for $\mathcal{X} \subseteq \mathcal{S}^k \times \cdots \times \mathcal{S}^l$; and to sets of events: $\mathcal{N}_{\mathcal{F}}(\mathcal{X}) = \bigcup_{e \in \mathcal{F}} \mathcal{N}_e(\mathcal{X})$, for $\mathcal{F} \subseteq \mathcal{E}$. In particular, we write $\mathcal{N}_{\leq k}$ for $\mathcal{N}_{\{e:\mathit{First}(e) \leq k\}}$.

3. A Novel Algorithm Employing Node Saturation. Recall that we describe the behavior of an event–based asynchronous system using a product–form next–state function for each event. The system's state space may then be built by iterating these functions in any order, as long as each is considered often enough [21], i.e., until no additional reachable states are found. We refer to a specific order of iteration as *iteration strategy*. Clearly, the choice of strategy influences the efficiency of state–space generation. In our previous work [11] we employed a naive strategy that cycled through MDDs level–by–level and fired, at each level k, all events e with $\mathit{First}(e) = k$.

As main contribution of this paper, we present a novel iteration strategy, called *saturation*, which not only simplifies our previous algorithm, but also significantly improves its time *and* space efficiency. The key idea is to fire events node–wise and exhaustively, instead of level–wise and just once per iteration. Formally, we say that an MDD node $\langle k.p \rangle$ is *saturated* if it encodes a set of states that is a fixed point with respect to the firing of any event at its level or at a lower level, i.e., if $\mathcal{B}(\langle k.p \rangle) = \mathcal{N}_{\leq k}^*(\mathcal{B}(\langle k.p \rangle))$ holds; it can easily be shown by contradiction that any node below node $\langle k.p \rangle$ must be saturated, too. It should be noted that the routine for firing some event, in order to reveal and add globally reachable states to the MDD–representation of the state space under construction, is similar to [11]. In particular, MDDs are only locally manipulated with respect to the levels on which the fired event depends, and, due to the product–form behavior, these manipulations can be carried out very efficiently. We do not further comment on these issues here, but concentrate solely on the new idea of node saturation and its implications.

Just as in traditional symbolic state–space generation algorithms, we use a *unique table*, to detect duplicate nodes, and *operation caches*, in particular a *union cache* and a *firing cache*, to speed–up computation.

However, our approach is distinguished by the fact that only saturated nodes are checked in the unique table or referenced in the caches. Given the MDD encoding of the initial state $\mathbf{s}$, we saturate its nodes bottom–up. This improves both memory and execution–time efficiency for generating state spaces because of the following reasons. First, our saturation order ensures that the firing of an event affecting only the current and possibly lower levels adds as many new states as possible. Then, since each node in the final encoding of $\mathcal{S}$ is saturated, any node we insert in the unique table has at least a chance of being still part of the final MDD, while any unsaturated node inserted by a traditional symbolic approach is *guaranteed* to be eventually deleted and replaced with another node encoding a larger subset of states. Finally, once we saturate a node at level k, we never need to fire any event $e \in \mathcal{E}^k$ in it again, while, in classic symbolic approaches, $\mathcal{N}$ is applied to the entire MDD at *every iteration*.

In the pseudo–code of our new algorithm implementing node saturation, which is shown in Figure 3.1, we use the data types *evnt* (model event), *lcl* (local state), *lvl* (level), and *idx* (node index within a level); in practice these are simply integers in appropriate ranges. We also assume the following dynamically–sized global hash tables: (a) $UT[k]$, for $K \geq k \geq 1$, the *unique table* for nodes at level k, to retrieve p given the key $\langle k.p\rangle[0], \ldots, \langle k.p\rangle[n^k - 1]$; (b) $UC[k]$, for $K > k \geq 1$, the *union cache* for nodes at level k, to retrieve s given nodes p and q, where $\mathcal{B}(\langle k.s\rangle) = \mathcal{B}(\langle k.p\rangle) \cup \mathcal{B}(\langle k.q\rangle)$; and (c) $FC[k]$, for $K > k \geq 1$, the *firing cache* for nodes at level k, to retrieve s given node p and event e, where $First(e) > k$ and $\mathcal{B}(\langle k.s\rangle) = \mathcal{N}^*_{\leq k}(\mathcal{N}_e(\mathcal{B}(\langle k.p\rangle)))$. 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 call $Generate(\mathbf{s})$ creates the MDD encoding the initial state, saturating each MDD node as soon as it creates it, in a bottom–up fashion. Hence, when it calls $Saturate(k, r)$, all children of $\langle k.r\rangle$ are already saturated. Thus, our focus for the algorithm's correctness is on the correctness of *Saturate* and the routine *RecFire* invoked by it.

Theorem 3.1 (Correctness). *Consider a node $\langle k.p\rangle$ with $K \geq k \geq 1$ and saturated children. Moreover, (a) let $\langle l.q\rangle$ be one of its children, satisfying $q \neq \mathbf{0}$ and $l = k-1$; (b) let $\mathcal{U}$ stand for $\mathcal{B}(\langle l.q\rangle)$ before the call $RecFire(e, l, q)$, for some event e with $l < First(e)$, and let $\mathcal{V}$ represent $\mathcal{B}(\langle l.f\rangle)$, where f is the value returned by this call; and (c) let $\mathcal{X}$ and $\mathcal{Y}$ denote $\mathcal{B}(\langle k.p\rangle)$ before and after calling $Saturate(k, p)$, respectively. Then, (i) $\mathcal{V} = \mathcal{N}^*_{\leq l}(\mathcal{N}_e(\mathcal{U}))$ and (ii) $\mathcal{Y} = \mathcal{N}^*_{\leq k}(\mathcal{X})$.*

By choosing, for node $\langle k.p\rangle$, the root $\langle K.r\rangle$ of the MDD representing the initial system state $\mathbf{s}$, we obtain $\mathcal{Y} = \mathcal{N}^*_{\leq K}(\mathcal{B}(\langle K.r\rangle)) = \mathcal{N}^*_{\leq K}(\{\mathbf{s}\}) = \mathcal{S}$, as desired.

Proof. To prove both statements we employ a simultaneous induction on k. For the induction base, $k = 1$, we have: (i) The only possible call $RecFire(e, 0, \mathbf{1})$ immediately returns $\mathbf{1}$ because of the test on l (cf. line 1). Then, $\mathcal{U} = \mathcal{V} = \{()\}$ and $\{()\} = \mathcal{N}^*_{\leq 0}(\mathcal{N}_e(\{()\}))$. (ii) The call $Saturate(1, p)$ repeatedly explores λ^1, the only event in $\mathcal{E}^1$, in every local state i for which $\mathcal{N}^1_{\lambda^1}(i) \neq \emptyset$ and for which $\langle 1.p\rangle[i]$ is either $\mathbf{1}$ at the beginning of the "while $\mathcal{L} \neq \emptyset$" loop, or has been modified (cf. line 12) from $\mathbf{0}$ to $\mathbf{1}$, which is the value of f, hence u, since the call $RecFire(e, 0, \mathbf{1})$ returns $\mathbf{1}$. The iteration stops when further attempts to fire λ^1 do not add any new state to $\mathcal{B}(\langle 1.p\rangle)$. At this point, $\mathcal{Y} = \mathcal{N}^*_{\lambda^1}(\mathcal{X}) = \mathcal{N}^*_{\leq 1}(\mathcal{X})$.

For the induction step we assume that the calls to $Saturate(k-1, \cdot)$ as well as to $RecFire(e, l-1, \cdot)$ work correctly. Recall that $l = k - 1$.

(i) Unlike *Saturate* (cf. line 14), *RecFire* does not add further local states to $\mathcal{L}$, since it modifies "in–place" the new node $\langle l.s\rangle$, and not the node $\langle l.q\rangle$ describing the states from where the firing is explored. The call $RecFire(e, l, q)$ can be resolved in three ways. If $l < Last(e)$, then the returned

```
Generate(in s:array[1..K] of lcl):idx
  Build an MDD rooted at ⟨K.r⟩ encoding $\mathcal{N}^*_{\mathcal{E}}(\mathbf{s})$ and return r, in UT[K].

  declare r,p:idx;
  declare k:lvl;
   1. $p \Leftarrow \mathbf{1}$;
   2. for $k = 1$ to $K$ do
   3.   $r \Leftarrow NewNode(k)$; $\langle k.r\rangle[\mathbf{s}[k]] \Leftarrow p$;
   4.   $Saturate(k, r)$; $Check(k, r)$;
   5.   $p \Leftarrow r$;  return $r$;

Saturate(in k:lvl, p:idx)
  Update ⟨k.p⟩, not in UT[k], in–place, to encode $\mathcal{N}^*_{\le k}(\mathcal{B}(\langle k.p\rangle))$.

  declare e:evnt;
  declare $\mathcal{L}$:set of lcl;
  declare f,u:idx;
  declare i,j:lcl;
  declare pCng:bool;
   1. repeat
   2.   $pCng \Leftarrow false$;
   3.   foreach $e \in \mathcal{E}^k$ do
   4.     $\mathcal{L} \Leftarrow Locals(e, k, p)$;
   5.     while $\mathcal{L} \ne \emptyset$ do
   6.       $i \Leftarrow Pick(\mathcal{L})$;
   7.       $f \Leftarrow RecFire(e, k-1, \langle k.p\rangle[i])$;
   8.       if $f \ne \mathbf{0}$ then
   9.         foreach $j \in \mathcal{N}^k_e(i)$ do
  10.           $u \Leftarrow Union(k-1, f, \langle k.p\rangle[j])$;
  11.           if $u \ne \langle k.p\rangle[j]$ then
  12.             $\langle k.p\rangle[j] \Leftarrow u$; $pCng \Leftarrow true$;
  13.             if $\mathcal{N}^k_e(j) \ne \emptyset$ then
  14.               $\mathcal{L} \Leftarrow \mathcal{L} \cup \{j\}$;
  15. until $pCng = false$;

Union(in k:lvl, p:idx, q:idx):idx
  Build an MDD rooted at ⟨k.s⟩, in UT[k], encoding $\mathcal{B}(\langle k.p\rangle) \cup \mathcal{B}(\langle k.q\rangle)$. Return s.

  declare i:lcl;
  declare s,u:idx;
   1. if $p = \mathbf{1}$ or $q = \mathbf{1}$ then return $\mathbf{1}$;
   2. if $p = \mathbf{0}$ or $p = q$ then return $q$;
   3. if $q = \mathbf{0}$ then return $p$;
   4. if $Find(UC[k], \{p, q\}, s)$ then return $s$;
   5. $s \Leftarrow NewNode(k)$;
   6. for $i = 0$ to $n^k - 1$ do
   7.   $u \Leftarrow Union(k-1, \langle k.p\rangle[i], \langle k.q\rangle[i])$;
   8.   $\langle k.s\rangle[i] \Leftarrow u$;
   9. $Check(k, s)$; $Insert(UC[k], \{p, q\}, s)$;
  10. return $s$;

RecFire(in e:evnt, l:lvl, q:idx):idx
  Build an MDD rooted at ⟨l.s⟩, in UT[l], encoding $\mathcal{N}^*_{\le l}(\mathcal{N}_e(\mathcal{B}(\langle l.q\rangle)))$. Return s.

  declare $\mathcal{L}$:set of lcl;
  declare f,u,s:idx;
  declare i,j:lcl;
  declare sCng:bool;
   1. if $l < Last(e)$ then return $q$;
   2. if $Find(FC[l], \{q, e\}, s)$ then return $s$;
   3. $s \Leftarrow NewNode(l)$; $sCng \Leftarrow false$;
   4. $\mathcal{L} \Leftarrow Locals(e, l, q)$;
   5. while $\mathcal{L} \ne \emptyset$ do
   6.   $i \Leftarrow Pick(\mathcal{L})$;
   7.   $f \Leftarrow RecFire(e, l-1, \langle l.q\rangle[i])$;
   8.   if $f \ne \mathbf{0}$ then
   9.     foreach $j \in \mathcal{N}^l_e(i)$ do
  10.       $u \Leftarrow Union(l-1, f, \langle l.s\rangle[j])$;
  11.       if $u \ne \langle l.s\rangle[j]$ then
  12.         $\langle l.s\rangle[j] \Leftarrow u$; $sCng \Leftarrow true$;
  13. if $sCng$ then $Saturate(l, s)$;
  14. $Check(l, s)$; $Insert(FC[l], \{q, e\}, s)$;
  15. return $s$;

Find(in tab, key, out v):bool
  If (key, x) is in hash table tab, set v to x and return true. Else, return false.

Insert(inout tab, in key, v)
  Insert (key, v) in hash table tab, if it does not contain an entry (key, ·).

Locals(in e:evnt, k:lvl, p:idx):set of lcl
  Ret. $\{i \in \mathcal{S}^k : \langle k.p\rangle[i] \ne \mathbf{0}, \mathcal{N}^k_e(i) \ne \emptyset\}$, the local states in p locally enabling e. Return $\emptyset$ or $\{i \in \mathcal{S}^k : \mathcal{N}^k_e(i) \ne \emptyset\}$, respectively, if p is $\mathbf{0}$ or $\mathbf{1}$.

Pick(inout $\mathcal{L}$:set of lcl):lcl
  Remove and return an element from $\mathcal{L}$.

NewNode(in k:lvl):idx
  Create ⟨k.p⟩ with arcs set to $\mathbf{0}$, return p.

Check(in k:lvl, inout p:idx)
  If ⟨k.p⟩, not in UT[k], duplicates ⟨k.q⟩, in UT[k], delete ⟨k.p⟩ and set p to q. Else, insert ⟨k.p⟩ in UT[k]. If $\langle k.p\rangle[0] = \cdots = \langle k.p\rangle[n^k - 1] = \mathbf{0}$ or $\mathbf{1}$, delete ⟨k.p⟩ and set p to $\mathbf{0}$ or $\mathbf{1}$, since $\mathcal{B}(\langle k.p\rangle)$ is $\emptyset$ or $\mathcal{S}^k \times \cdots \times \mathcal{S}^1$, respectively.
```

Fig. 3.1. *Pseudo–code for the node–saturation algorithm.*

value is $f = q$ and $\mathcal{N}_e^l(\mathcal{U}) = \mathcal{U}$ for any set $\mathcal{U}$; since q is saturated, $\mathcal{B}(\langle l.q \rangle) = \mathcal{N}^*_{\leq l}(\mathcal{B}(\langle l.q \rangle)) = \mathcal{N}^*_{\leq l}(\mathcal{N}_e(\mathcal{B}(\langle l.q \rangle)))$. If $l \geq \mathit{Last}(e)$ but *RecFire* has been called previously with the same parameters, then the call $\mathit{Find}(FC[l], \{q, e\}, s)$ is successful. Since node q is saturated and in the unique table, it has not been modified further; note that in–place updates are performed only on nodes not yet in the unique table. Thus, the value s in the cache is still valid and can be safely used. Finally, we need to consider the case where the call $\mathit{RecFire}(e, l, q)$ performs "real work." First, a new node $\langle l.s \rangle$ is created, having all its arcs initialized to $\mathbf{0}$. We explore the firing of e in each state i satisfying $\langle l.q \rangle[i] \neq \mathbf{0}$ and $\mathcal{N}_l^e(i) \neq \emptyset$. By induction hypothesis, the recursive call $\mathit{RecFire}(e, l-1, \langle l.q \rangle[i])$ returns $\mathcal{N}^*_{\leq l-1}(\mathcal{N}_e(\mathcal{B}(\langle l-1.\langle l.q \rangle[i] \rangle)))$. Hence, when the "while $\mathcal{L} \neq \emptyset$" loop terminates, $\mathcal{B}(\langle l.s \rangle) = \bigcup_{i \in \mathcal{S}^l} \mathcal{N}_e^l(i) \times \mathcal{N}^*_{\leq l-1}(\mathcal{N}_e(\mathcal{B}(\langle l-1.\langle l.q \rangle[i] \rangle))) = \mathcal{N}^*_{\leq l-1}(\mathcal{N}_e(\mathcal{B}(\langle l.q \rangle)))$ holds. Thus, all children of node $\langle l.s \rangle$ are saturated. According to the induction hypothesis, the call $\mathit{Saturate}(l, s)$ correctly saturates $\langle l.s \rangle$. Consequently, we have $\mathcal{B}(\langle l.s \rangle) = \mathcal{N}^*_{\leq l}(\mathcal{N}^*_{\leq l-1}(\mathcal{N}_e(\mathcal{B}(\langle l.q \rangle)))) = \mathcal{N}^*_{\leq l}(\mathcal{N}_e(\mathcal{B}(\langle l.q \rangle)))$ after the call.

(ii) As in the base case, $\mathit{Saturate}(k, p)$ repeatedly explores the firing of each event e that is locally enabled in $i \in \mathcal{S}^k$, by calling $\mathit{RecFire}(e, k-1, \langle k.p \rangle[i])$ which, as shown above and since $l = k-1$, returns $\mathcal{N}^*_{\leq k-1}(\mathcal{N}_e(\mathcal{B}(\langle k-1.\langle k.p \rangle[i] \rangle)))$. Further, $\mathit{Saturate}(k, p)$ terminates when firing the events in $\mathcal{E}^k = \{e_1, e_2, \ldots, e_m\}$ does not add any new state to $\mathcal{B}(\langle k.p \rangle)$. At this point, the set $\mathcal{Y}$ encoded by $\langle k.p \rangle$ is the fixed–point of the iteration

$$\mathcal{Y}^{(m+1)} \Leftarrow \mathcal{Y}^{(m)} \cup \mathcal{N}^*_{\leq k-1}(\mathcal{N}_{e_1}(\mathcal{N}^*_{\leq k-1}(\mathcal{N}_{e_2}(\cdots \mathcal{N}^*_{\leq k-1}(\mathcal{N}_{e_m}(\mathcal{Y}^{(m)}))\cdots)))),$$

initialized with $\mathcal{Y}^{(0)} \Leftarrow \mathcal{X}$ [21]. Hence, $\mathcal{Y} = \mathcal{N}^*_{\leq k}(\mathcal{X})$, as desired.

This completes the correctness proof of the algorithm. □

Figure 3.2 illustrates our saturation–based state–space generation algorithm on a small example, where $K = 3$, $|\mathcal{S}_3| = 2$, $|\mathcal{S}_2| = 3$, and $|\mathcal{S}_1| = 3$. The initial state is $(0, 0, 0)$, and there are three local events l_1, l_2, and l_3, plus two further events, e_{21} (depending on levels 2 and 1) and e_{321} (depending on all levels). Their effects, i.e., their next–state functions, are summarized in the table at the top of Figure 3.2; the symbol "$*$" indicates that a level does not affect an event. The MDD encoding $\{(0, 0, 0)\}$ is displayed in Snapshot (a). Nodes $\langle 3.2 \rangle$ and $\langle 2.2 \rangle$ are actually created in Steps (b) and (g), respectively, but we show them from the beginning for clarity. The level *lvl* of a node $\langle \mathit{lvl}.\mathit{idx} \rangle$ is given at the very left of the MDD figures, whereas the index *idx* is shown to the right of each node. We use dashed lines for newly created objects, double boxes for saturated nodes, and shaded local states for substates enabling the event to be fired. We do not show nodes with index $\mathbf{0}$ nor any arcs to them.

- *Snapshots (a–b):* The call $\mathit{Saturate}(1, 2)$ updates node $\langle 1.2 \rangle$ to represent the effect of firing l_1^*; the result is equal to the reserved node $\langle 1.\mathbf{1} \rangle$.
- *Snapshots (b–f):* The call $\mathit{Saturate}(2, 2)$ fires event l_2, adding arc $\langle 2.2 \rangle[1]$ to $\langle 1.\mathbf{1} \rangle$ (cf. Snapshot (c)). It also fires event e_{21} which finds the "enabling pattern" $(*, 0, 1)$, with arbitrary first component, and starts building the result of the firing, through the sequence of calls $\mathit{RecFire}(e_{21}, 1, \langle 2.2 \rangle[0])$ and $\mathit{RecFire}(e_{21}, 0, \langle 1.\mathbf{1} \rangle[1])$. Once node $\langle 1.3 \rangle$ is created and its arc $\langle 1.3 \rangle[0]$ is set to $\mathbf{1}$ (cf. Snapshot (d)), it is saturated by repeatedly firing event l_1. Node $\langle 1.3 \rangle$ then becomes identical to node $\langle 1.\mathbf{1} \rangle$ (cf. Snapshot (e)). Hence, it is not added to the unique table but deleted. Returning from *RecFire* on level 1 with result $\langle 1.\mathbf{1} \rangle$, arc $\langle 2.2 \rangle[1]$ is updated to point to the outcome of the firing (cf. Snapshot (f)). This does not add any new state to the MDD, since the state set $\mathcal{S}^3 \times \{1\} \times \{0\}$ was already encoded in $\mathcal{B}(\langle 2.2 \rangle)$.

level	event: l_1	event: l_2	event: l_3	event: e_{21}	event: e_{321}
3	$*$	$*$	$1 \to 0$	$*$	$0 \to 1$
2	$*$	$0 \to 1, 2 \to 1$	$*$	$0 \to 1$	$0 \to 2$
1	$0 \to 1, 1 \to 2, 2 \to 0$	$*$	$*$	$1 \to 0$	$0 \to 1$

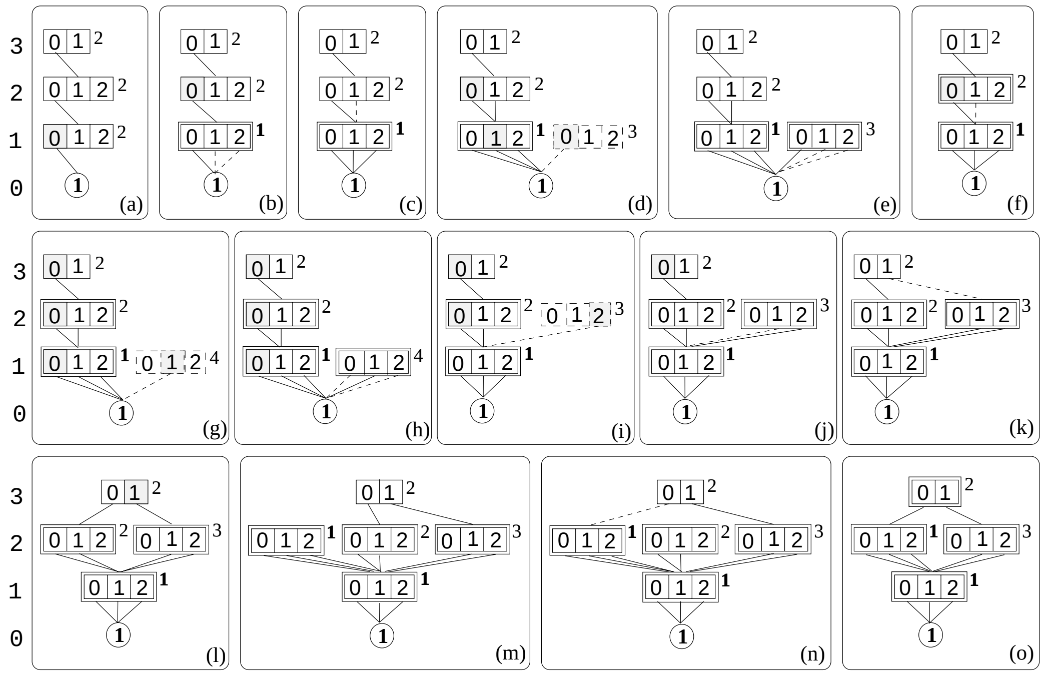


FIG. 3.2. *Example of the execution of the Saturate and RecFire routines.*

- *Snapshots (f–o):* Once $\langle 2.2 \rangle$ is saturated, we call *Saturate*$(3, 2)$. Local event l_3 is not enabled, but event e_{321} is, by the pattern $(0, 0, 0)$. The calls to *RecFire* build a chain of nodes encoding the result of the firing (cf. Snapshots (g–i)). Each of them is in turn saturated (cf. Snapshots (h–j)), causing first the newly created node $\langle 1.4 \rangle$ to be deleted, since it becomes equal to node $\langle 1.\mathbf{1} \rangle$, and second the saturated node $\langle 2.3 \rangle$ to be added to the MDD. The firing of e_{321} (cf. Snapshot (k)) not only adds state $(1, 2, 1)$, but the entire subspace $\{1\} \times \{1, 2\} \times \mathcal{S}^1$, now known to be exhaustively explored, as node $\langle 2.3 \rangle$ is marked saturated. Event l_3, which was found disabled in node $\langle 3.2 \rangle$ at the first attempt, is now enabled, and its firing calls *Union*$(2, \langle 3.2 \rangle[1], \langle 3.2 \rangle[0])$. The result is a new node which is found by *Check* to be the reserved node $\langle 2.\mathbf{1} \rangle$ (cf. Snapshot (m)). This node encoding $\mathcal{S}_2 \times \mathcal{S}_1$ is added as the descendant of node $\langle 3.2 \rangle$ in position 0, and the former descendant $\langle 2.2 \rangle$ in that position is removed (cf. Snapshot (n)), causing it to become disconnected and deleted. Further attempts to fire events l_3 or e_{321} add no more states to the MDD, whence node $\langle 3.2 \rangle$ is declared saturated (cf. Snapshot (o)). Thus, our algorithm terminates and returns the overall state space $(\{0\} \times \mathcal{S}^2 \times \mathcal{S}^1) \cup (\{1\} \times \{1, 2\} \times \mathcal{S}^1)$.

To summarize, since MDD nodes are saturated as soon as they are created, each node will either be present in the final diagram or will eventually become disconnected, but never be modified. This reduces the amount of work needed to explore subspaces. Once all events in $\mathcal{E}^k$ are exhaustively fired in some node $\langle k.p \rangle$, any further state discovered that uses $\langle k.p \rangle$ for its encoding benefits in advance from the "knowledge" encapsulated in $\langle k.p \rangle$ and its descendants.

4. Garbage Collection, Optimizations, and Generalizations. Before evaluating our saturation algorithm by means of experimental studies, we briefly discuss some implementation details regarding garbage–collection policies, mention two optimizations noticeably affecting the algorithm's performance, and remark on extending the algorithm to deal with multiple initial system states.

4.1. Garbage Collection. MDD nodes can become disconnected, i.e., unreachable from the root, and should be "recycled." Disconnection is detected by associating an *incoming–arc counter* to each node $\langle k.p \rangle$ such that $\langle k.p \rangle$ is disconnected if and only if its counter is zero. Recycling disconnected nodes is a major issue in traditional symbolic state–space generation algorithms, where usually many nodes become disconnected. In our algorithm, this phenomenon is much less frequent, and the best runtime is achieved by removing these nodes only at the end; we refer to this policy as LAZY policy.

We also implemented a STRICT policy where, if a node $\langle k.p \rangle$ becomes disconnected, its "delete–flag" is set and its arcs $\langle k.p \rangle[i]$ are re–directed to $\langle k-1.\mathbf{0} \rangle$, with possible recursive effects on the nodes downstream. When a hit in the union cache $UC[k]$ or the firing cache $FC[k]$ returns s, we consider this entry stale if the delete–flag of node $\langle k.s \rangle$ is set. By keeping a per–level count of the nodes with delete–flag set, we can decide in routine $NewNode(k)$ whether (a) to allocate new memory for a node at level k or (b) to recycle the indexes and the physical memory of all nodes at level k with delete–flag set, after having removed all the entries in $UC[k]$ and $FC[k]$ referring to them. The threshold that triggers recycling can be set in terms of numbers of nodes or bytes of memory. The policy using a threshold of one node, denoted as STRICT(1), is optimal in terms of memory consumption, but has a higher overhead due to frequent clean–ups.

4.2. Optimizations. In our implementation we employ several optimizations. For example, the two outermost loops in *Saturate* ensure that firing any event $e \in \mathcal{E}^k$ adds no new states. However, if we always consider these events in the same order, we can stop iterating as soon as $|\mathcal{E}^k|$ consecutive events have been explored without revealing any new state. This saves $|\mathcal{E}^k|/2$ firing attempts on average, which translates to speed–ups of up to 25% in our experimental studies. Also, in *Union*, the call $Insert(UC[k], \{p, q\}, s)$ records that $\mathcal{B}(\langle k.s \rangle) = \mathcal{B}(\langle k.p \rangle) \cup \mathcal{B}(\langle k.q \rangle)$. Since this implies $\mathcal{B}(\langle k.s \rangle) = \mathcal{B}(\langle k.p \rangle) \cup \mathcal{B}(\langle k.s \rangle)$ and $\mathcal{B}(\langle k.s \rangle) = \mathcal{B}(\langle k.s \rangle) \cup \mathcal{B}(\langle k.q \rangle)$, we can, optionally, also issue the calls $Insert(UC[k], \{p, s\}, s)$, if $s \neq p$, and $Insert(UC[k], \{q, s\}, s)$, if $s \neq q$. This *speculative union heuristic* improves performance up to 20%.

4.3. Generalizations. So far we only discussed state–space generation starting from an MDD encoding a single initial system state. We also implemented an extended version of our algorithm that can compute $\mathcal{N}^*(\mathcal{B}(\langle K.r \rangle))$ for any arbitrary MDD rooted at node $\langle K.r \rangle$. This is of importance for adapting our ideas to *model checking* [16]. The necessary technical details underlying this issue are quite straightforward and, thus, are omitted here.

5. Experimental Results. In this section we compare the performance of our new algorithm, using both the STRICT and LAZY policies, with previous MDD–based ones, namely the traditional RECURSIVE MDD approach in [30] and the level–by–level FORWARDING–arcs approach in [11]. All three approaches are implemented in SMART [12], a tool for the logical and stochastic–timing analysis of discrete–state systems. For asynchronous systems, these approaches greatly outperform the more traditional BDD–based approaches [28], where next–state functions are encoded using decision diagrams. To evaluate our saturation algorithm, we have chosen a suite of examples with a wide range of characteristics. In all cases, the state space sizes depend on a parameter $N \in \mathbb{N}$.

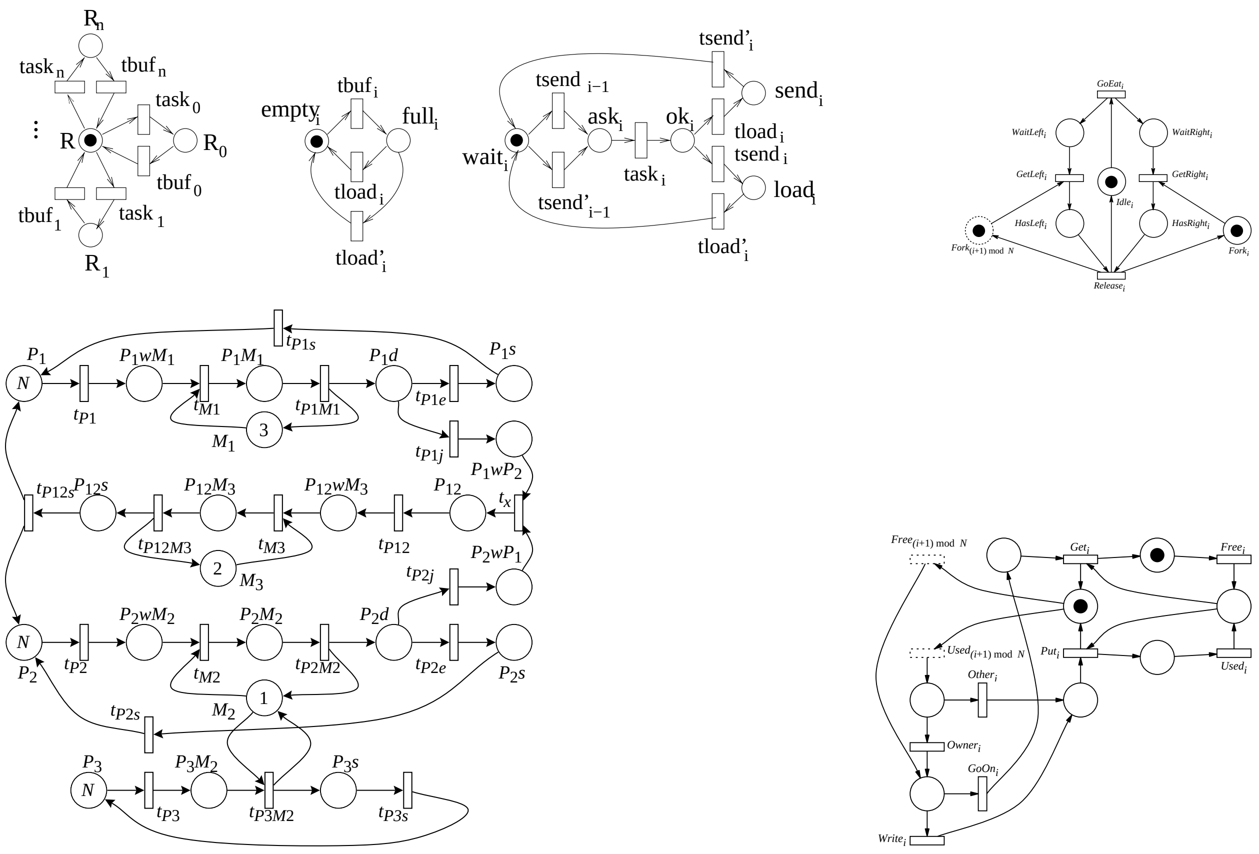


FIG. 5.1. *Petri nets used in our experiments: round–robin mutex protocol (upper left), dining philosophers (upper right), FMS (lower left), and slotted-ring (lower right).*

- The classic N *queens problem* requires to find a way to position N queens on a $N \times N$ chess board such that they do not attack each other. Since there will be exactly one queen per row in the final solution, we use a *safe* (i.e., at most one token per place) Petri net model with $N \times N$ transitions and N rows, one per MDD level, of $N+1$ places. For $1 \le i, j \le N$, place p_{ij} is initially empty, and place p_{i0} contains the token (queen) still to be placed on row i of the chess board. Transition t_{ij} moves the queen from place p_{i0} to place p_{ij}, in competition with all other transitions t_{il}, for $l \neq j$. To encode the mutual exclusion of queens on the same column or diagonal, we employ *inhibitor arcs*. A correct placement of the N queens corresponds to a marking where all places p_{i0} are empty. Note that our state space contains *all* reachable markings, including those where queens n to N still need to be placed, for any n. In this model, locality is poor, since t_{ij} depends on levels 1 through i.
- The *dining philosophers* and *slotted ring models* [11, 33] are obtained by connecting N identical safe subnets "in a circle." The MDD has $N/2$ MDD levels (two subnets per level) for the former model and N levels (one subnet per level) for the latter. Events are either local or synchronize adjacent subnets, thus they span only two levels, except for those synchronizing subnet N with subnet 1.
- The *round–robin mutex protocol* model [23] also has N identical safe subnets placed in a circular fashion, which represent N processes, each mapped to one MDD level. Another subnet models a resource shared by the N processes, giving raise to one more level, at the bottom of the MDD. There are no local events and, in addition to events synchronizing adjacent subnets, the model contains events synchronizing levels n and 1, for $2 \le n \le N+1$.

- The *flexible manufacturing system* (FMS) model [30] has a fixed shape, but is parameterized by the initial number N of tokens in some places. We partition this model into 19 subnets, giving rise to a 19–level MDD with a moderate degree of locality, as events span from two to six levels.

The Petri nets for these systems, except for the queens problem, are depicted in Figure 5.1.

Figure 5.2 compares three variants of our new algorithm, using the Lazy policy or the Strict policy with thresholds of 1 or 100 nodes per level, respectively, against the Recursive algorithm in [30] and the Forwarding algorithm in [11]. We ran SMART on a 800 MHz Intel Pentium III PC under Linux. On the left of Figure 5.2, we give the size of the state space for each model and the value of N. The graphs in the middle and right columns show the peak and final numbers of MDD nodes and the CPU time in seconds required for the state–space generations, respectively.

For the models introduced above, our new approach is up to two orders of magnitude faster than [30] (a speed–up factor of 384 is obtained for the 1000 dining philosophers' model), and up to one order of magnitude faster than [11] (a speed–up factor of 38 is achieved for the slotted ring model with 50 slots). These results are observed for the Lazy variant of the algorithm, which yields the best runtimes; the Strict policy also outperforms [30] and [11]. Furthermore, the gap keeps increasing as we scale up the models. Just as important, the saturation algorithm tends to use many fewer MDD nodes, whence less memory. This is most apparent in the FMS model, where the difference between the peak and the final number of nodes is just a constant, 10, for any Strict policy. Also notable is the reduced memory consumption for the slotted ring model, where the Strict(1) policy uses 23 times fewer nodes compared to [30], for $N = 50$. In terms of absolute memory requirements, the number of nodes is essentially proportional to bytes of memory. For reference, the largest memory consumption in our experiments was recorded with 9.7MB for the FMS model with 100 tokens; auxiliary data structures required up to 2.5MB for encoding the next–state functions and 200KB for storing the local state spaces, while the caches used less than 1MB. Other SMART structures account for another 4MB.

In a nutshell, with respect to generation time, the best algorithm is Lazy, followed by Strict(100), Strict(1), Forwarding, and Recursive. According to memory consumption, the best algorithm is Strict(1), followed by Strict(100), Lazy, Forwarding, and Recursive. Thus, our new algorithm is consistently faster and uses less memory than previously proposed approaches. The worst model for all algorithms is the queens problem, which has a very large number of nodes in the final representation of $\mathcal{S}$ and little locality. Even here, however, our algorithm uses slightly fewer nodes and is substantially faster. Finally, we observe that, when the Lazy and Strict policies differ widely in terms of memory consumption and CPU time, the choice of threshold for the Strict policy lets us trade–off time vs. space efficiency.

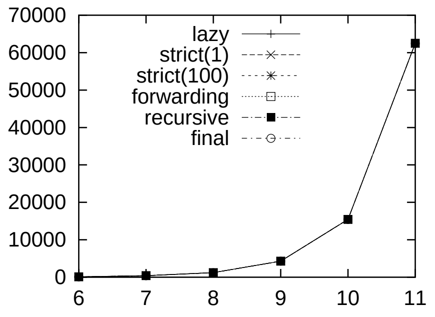
Hence, exploiting the locality inherent in asynchronous systems and employing a clever strategy for iterating their local next–state functions, is the key to efficiency for symbolic state–space generators.

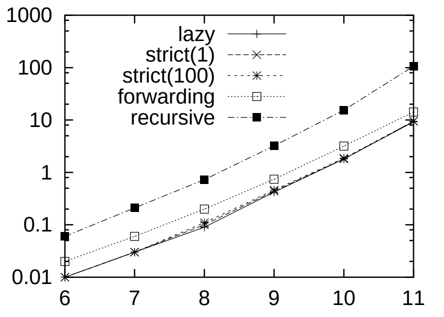
6. Related Work. We already pointed out the significant differences of our approach to symbolic state–space generation when compared to traditional approaches reported in the literature [28], which are usually deployed for model checking [14]. Hence, for comparing our algorithm to this work fairly, it needs to be extended to a full model checker first, which is currently being investigated. The following sections briefly survey some orthogonal and alternative approaches to improving the scalability of state–space generation and model–checking techniques. These approaches can be classified according to whether state spaces are represented either *explicitly* or *symbolically.*

Model & size | **Peak & final MDD nodes** | **Generation time (sec.)**

Queens

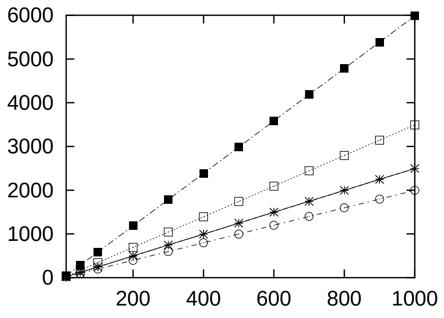
N	$\mathcal{S}$
6	$1.53{\cdot}10^{2}$
7	$5.52{\cdot}10^{2}$
8	$2.06{\cdot}10^{3}$
9	$8.39{\cdot}10^{3}$
10	$3.55{\cdot}10^{4}$
11	$1.67{\cdot}10^{5}$

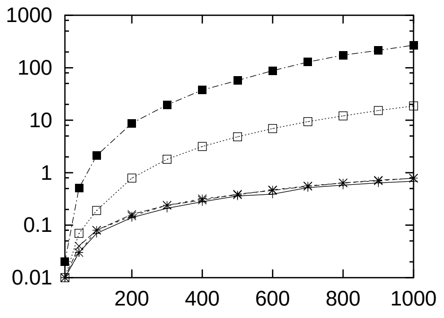




Dining phil.

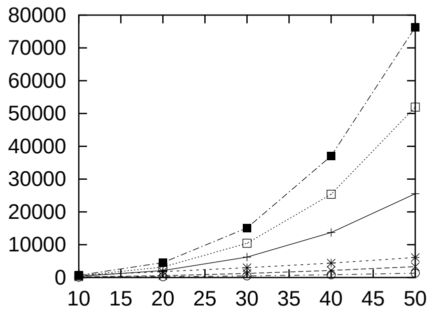
N	$\mathcal{S}$
100	$4.97{\cdot}10^{62}$
200	$2.47{\cdot}10^{125}$
400	$6.10{\cdot}10^{250}$
600	$1.51{\cdot}10^{376}$
800	$3.72{\cdot}10^{501}$
1000	$9.18{\cdot}10^{626}$

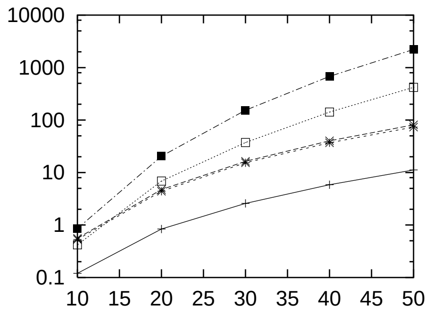




Slotted ring

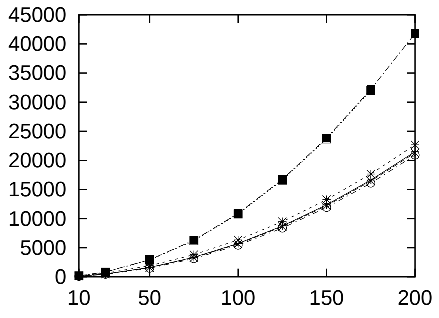
N	$\mathcal{S}$
10	$8.29{\cdot}10^{9}$
20	$2.73{\cdot}10^{20}$
30	$1.04{\cdot}10^{31}$
40	$4.16{\cdot}10^{41}$
50	$1.72{\cdot}10^{52}$

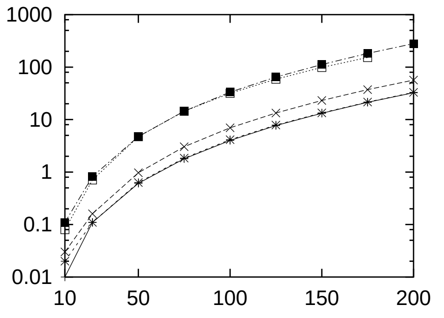




Round robin

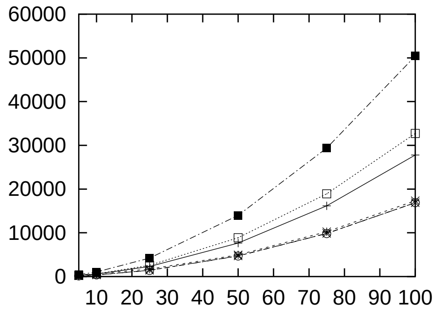
N	$\mathcal{S}$
10	$2.30{\cdot}10^{4}$
25	$1.89{\cdot}10^{9}$
50	$1.27{\cdot}10^{17}$
100	$2.85{\cdot}10^{32}$
150	$4.82{\cdot}10^{47}$
200	$7.23{\cdot}10^{62}$





FMS

N	$\mathcal{S}$
5	$2.90{\cdot}10^{6}$
10	$2.50{\cdot}10^{9}$
25	$8.54{\cdot}10^{13}$
50	$4.24{\cdot}10^{17}$
75	$6.98{\cdot}10^{19}$
100	$2.70{\cdot}10^{21}$



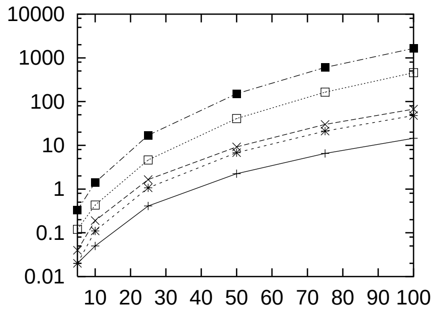


FIG. 5.2. *State space sizes, memory consumption, and generation times (*logscale*). Note: The curves in the upper left diagram are almost identical and, thus, appear to coincide.*

6.1. Explicit State–space Generation. Explicit techniques represent state spaces by trees, hash tables, or graphs, where each state corresponds to an entity of the underlying data structure. Thus, the memory needed to store the state space of a system is linear in the number of the system's states. To achieve space efficiency, numerous techniques have been introduced, including multi–level data structures [13] and merging common bitvectors [24]. To avoid state–space explosion for asynchronous system models, researchers often employ compositional construction techniques based on context constraints [23, 26], partial-order techniques [22], or symmetry reduction [15].

6.2. Symbolic State–space Generation. Regarding *synchronous hardware systems*, symbolic techniques using BDDs, which can represent state spaces in sublinear space, have been thoroughly investigated [17]. Several implementations of BDDs are available. We refer the reader to [36] for a good survey on BDD packages and their performance. To improve the time efficiency of BDD–based algorithms, breadth–first BDD–manipulation algorithms [4] have been explored and compared against the traditional depth–first ones. However, the results show no significant speed–ups, although breadth–first algorithms lead to more regular access patterns of hash tables and caches. Regarding space efficiency, a fair amount of work has concentrated on choosing appropriate variable orderings and on dynamically re–ordering variables [20].

For *asynchronous software systems*, symbolic techniques have been investigated less, and mostly only in the setting of Petri nets. For safe Petri nets, BDD-based algorithms for the generation of the reachability set have been developed in [33, 35] via encoding each place of a net as a Boolean variable. These algorithms are capable of generating state spaces of large nets within hours. Recently, more efficient encodings of nets have been introduced, which take place invariants [32] into account, although the underlying logic is still based on Boolean variables. In contrast, our work uses a more general version of decision diagrams, namely MDDs [25, 30], where more complex information is carried in each node of a diagram. In particular, MDDs allow for a natural encoding of asynchronous system models, such as distributed embedded systems.

For the sake of completeness, we briefly mention some other BDD–based techniques exploiting the component–based structure of many digital systems. They include partial model checking [3], compositional model checking [27], partial–order reduction [2], and conjunctive decompositions [29]. Finally, also note that approaches to symbolic verification have been developed, which do not rely on decision diagrams but instead on arithmetic or algebra [1, 6, 34].

7. Conclusions and Future Work. We presented a novel approach for constructing the state spaces of asynchronous system models using MDDs. By avoiding to encode a given global next–state function as an MDD, but splitting it into several local next–state functions instead, we gained the freedom to choose the sequence of event firings, which controls the fixed–point iteration resulting in the desired global state space. Our central contribution is the development of a specific elegant iteration strategy based on saturating MDD nodes. Its utility is proved by experimental studies which show that our algorithm often performs several orders of magnitude faster than most existing algorithms. Equally important, the peak sizes of MDDs are usually kept close to their final sizes.

Regarding future work, we plan to employ our idea of saturation for implementing an MDD–based CTL model checker within SMART [12], to compare the model checker to state–of–the–art BDD–based model checkers, and to test our tool on examples that are extracted from real software. Moreover, we intend to investigate whether our new algorithm is suitable for parallelization.

REFERENCES

[1] P. A. Abdulla, P. Bjesse, and N. Eén, *Symbolic reachability analysis based on SAT-solvers*, in 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2000), S. Graf and M. Schwartzbach, eds., Vol. 1785 of Lecture Notes in Computer Science, Berlin, Germany, March/April 2000, Springer-Verlag, pp. 411–425.

[2] R. Alur, R. Brayton, T. Henzinger, S. Qadeer, and S. Rajamani, *Partial-order reduction in symbolic state-space exploration*, in 9th International Conference on Computer-Aided Verification (CAV '97), O. Grumberg, ed., Vol. 1254 of Lecture Notes in Computer Science, Haifi, Israel, June 1997, Springer-Verlag, pp. 340–351.

[3] H. Andersen, J. Staunstrup, and N. Maretti, *Partial model checking with ROBDDs*, in Brinksma [7], pp. 35–49.

[4] P. Ashar and M. Cheong, *Efficient breadth–first manipulation of binary decision diagrams*, in IEEE International Conference on Computer Aided Design (ICCAD '94), San Jose, CA, USA, November 1994, Computer Society Press, pp. 622–627.

[5] J. Bergstra, A. Ponse, and S. Smolka, *Handbook of Process Algebra*, Elsevier Science, 2000.

[6] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, *Symbolic model checking without BDDs*, in 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '99), R. Cleaveland, ed., Vol. 1579 of Lecture Notes in Computer Science, Amsterdam, The Netherlands, April 1999, Springer-Verlag, pp. 193–207.

[7] E. Brinksma, ed., *3rd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '97)*, Vol. 1217 of Lecture Notes in Computer Science, Enschede, The Netherlands, April 1997, Springer-Verlag.

[8] R. Bryant, *Graph-based algorithms for Boolean function manipulation*, IEEE Transactions on Computers, 35 (1986), pp. 677–691.

[9] ——, *Symbolic Boolean manipulation with ordered binary-decision diagrams*, ACM Computing Surveys, 24 (1992), pp. 393–418.

[10] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang, *Symbolic model checking: 10^{20} states and beyond*, Information and Computation, 98 (1992), pp. 142–170.

[11] G. Ciardo, G. Lüttgen, and R. Siminiceanu, *Efficient symbolic state–space construction for asynchronous systems*, in 21st International Conference on Application and Theory of Petri Nets (ICATPN 2000), M. Nielsen and D. Simpson, eds., Vol. 1639 of Lecture Notes in Computer Science, Aarhus, Denmark, June 2000, Springer-Verlag, pp. 103–122.

[12] G. Ciardo and A. Miner, *SMART: Simulation and Markovian Analyzer for Reliability and Timing*, in IEEE International Computer Performance and Dependability Symposium (IPDS '96), Urbana–Champaign, IL, USA, September 1996, Computer Society Press, p. 60.

[13] ——, *Storage alternatives for large structured state spaces*, in 9th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation (Tools '97), R. Marie, B. Plateau, M. Calzarossa, and G. Rubino, eds., Vol. 1245 of Lecture Notes in Computer Science, St. Malo, France, June 1997, Springer-Verlag, pp. 44–57.

[14] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri, *NuSMV: A new symbolic model verifier*, in 11th International Conference on Computer-Aided Verification (CAV '99), N. Halbwachs and D. Peled, eds., Vol. 1633 of LNCS, Trento, Italy, July 1999, Springer-Verlag, pp. 495–499.

[15] E. Clarke, T. Filkorn, and S. Jha, *Exploiting symmetry in model checking*, in 5th International Workshop on Computer Aided Verification (CAV '93), C. Courcoubetis, ed., Vol. 697 of Lecture Notes in Computer Science, Elounda, Greece, June/July 1993, Springer-Verlag, pp. 450–462.

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

[17] E. Clarke and J. Wing, *Formal methods: State of the art and future directions*, ACM Computing Surveys, 28 (1996), pp. 626–643.

[18] R. Cleaveland, E. Madelaine, and S. Sims, *Generating front-ends for verification tools*, in 1st International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '95), E. Brinksma, W. Cleaveland, K. Larsen, T. Margaria, and B. Steffen, eds., Vol. 1019 of Lecture Notes in Computer Science, Aarhus, Denmark, May 1995, Springer-Verlag, pp. 153–173.

[19] R. Cleaveland, J. Parrow, and B. Steffen, *The Concurrency Workbench: A semantics-based tool for the verification of finite-state systems*, ACM Transactions on Programming Languages and Systems, 15 (1993), pp. 36–72.

[20] M. Fujita, H. Fujisawa, and Y. Matsunaga, *Variable ordering algorithms for ordered binary decision diagrams and their evaluation*, IEEE Transactions on Computer–Aided Design of Integrated Circuits and Systems, 12 (1993), pp. 6–12.

[21] A. Geser, J. Knoop, G. Lüttgen, B. Steffen, and O. Rüthing, *Chaotic fixed point iterations*, Tech. Report MIP-9403, University of Passau, Germany, October 1994.

[22] P. Godefroid, *Partial-order Methods for the Verification of Concurrent Systems*, Vol. 1032 of Lecture Notes in Computer Science, Springer-Verlag, 1996.

[23] S. Graf, B. Steffen, and G. Lüttgen, *Compositional minimisation of finite state systems using interface specifications*, Formal Aspects of Computing, 8 (1996), pp. 607–616.

[24] G. Holzmann, *The model checker Spin*, IEEE Transactions on Software Engineering, 23 (1997), pp. 279–295.

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

[26] J.-P. Krimm and L. Mounier, *Compositional state space generation from Lotos programs*, in Brinksma [7], pp. 239–258.

[27] K. Larsen, P. Pettersson, and W. Yi, *Compositional and symbolic model-checking of real-time systems*, in 16th IEEE Real-Time Systems Symposium (RTSS '95), Pisa, Italy, September 1995, Computer Society Press, pp. 76–89.

[28] K. McMillan, *Symbolic Model Checking: An Approach to the State-explosion Problem*, Ph.D. thesis, Carnegie-Mellon University, Pittsburgh, PA, USA, 1992.

[29] ———, *A conjunctively decomposed Boolean representation for symbolic model checking*, in 8th International Conference on Computer-Aided Verification (CAV '96), R. Alur and T. Henzinger, eds., Vol. 1102 of Lecture Notes in Computer Science, New Brunswick, NJ, USA, July 1996, Springer-Verlag, pp. 13–24.

[30] A. Miner and G. Ciardo, *Efficient reachability set generation and storage using decision diagrams*, in 20th International Conference on Application and Teory of Petri Nets (ICATPN '99), J. Kleijn and S. Donatelli, eds., Vol. 1639 of Lecture Notes in Computer Science, Williamsburg, VA, USA, June 1999, Springer-Verlag, pp. 6–25.

[31] T. Murata, *Petri nets: Properties, analysis and applications*, Proceedings of the IEEE, 77 (1989), pp. 541–579.

[32] E. Pastor and J. Cortadella, *Efficient encoding schemes for symbolic analysis of Petri nets*, in IEEE Conference on Design, Automation and Test in Europe (DATE '98), Paris, France, March 1998, Computer Society Press, pp. 790–795.

[33] E. Pastor, O. Roig, J. Cortadella, and R. Badia, *Petri net analysis using Boolean manipulation*, in 15th International Conference on the Application and Theory of Petri Nets (ICATPN '94), R. Valette, ed., Vol. 815 of Lecture Notes in Computer Science, Zaragoza, Spain, June 1994, Springer-Verlag, pp. 416–435.

[34] M. Sheeran and G. Stålmarck, *A tutorial on Stålmarck's proof procedure for propositional logic*, Formal Methods in System Design, 16 (2000), pp. 23–58.

[35] K. Varpaaniemi, J. Halme, K. Hiekkanen, and T. Pyssysalo, *PROD reference manual*, Tech. Report B13, Helsinki University of Technology, Finland, August 1995.

[36] B. Yang, R. Bryant, D. O'Hallaron, A. Biere, O. Coudert, G. Janssen, R. Ranjan, and F. Somenzi, *A performance study of BDD–based model checking*, in 2nd International Conference on Formal Methods in Computer–Aided Design (FMCAD '98), G. Gopalakrishnan and P. Windley, eds., Vol. 1522 of Lecture Notes in Computer Science, Palo Alto, CA, USA, November 1998, Springer-Verlag, pp. 255–289.