Modeling the Logical Behavior of Discrete–state Systems

Gianfranco Ciardo

Department of Computer Science
College of William and Mary
Williamsburg, VA 23187, USA
ciardo@cs.wm.edu

Copyright ©2002 Gianfranco Ciardo
All rights reserved
What is a BDD (Boolean Decision Diagram)?

What is the most cited computer science document in citeseer? (http://citeseer.nj.nec.com/)
Most cited source documents in the ResearchIndex database as of November 2001

This list only includes documents in the ResearchIndex database. Citations where one or more authors of the citing and cited articles match are not included. The data is automatically generated and may contain errors. The list is generated in batch mode and citation counts may differ from those currently in the ResearchIndex database, because the database is continuously updated.

1. Graph-Based Algorithms for Boolean Function Manipulation - Bryant (1986) (Correct)
In this paper we present a new data structure for representing Boolean functions and an associated set of...

An encryption method is presented with the novel property that publicly revealing an encryption key does not...
At any instant of time, a system is in a given state.

The set $S$ of possible states is called state-space.

Example of continuous state-spaces:
- Weight of an organism: $S = [0, +\infty)$
- Level of liquid in a tank: $S = [0, \text{maxlevel}]$

Example of discrete state-spaces:
- Number of passengers in an airplane: $S = \{0, 1, ..., \text{maxseats}\}$
- Number and type of available airplanes: $S = \{(n_1, ..., n_t) : t \in \mathbb{N}, n_1, ..., n_t \in \mathbb{N}\}$
- Value of the program counter and of each variable in a running computer program

we consider only discrete-state systems
The structure of the state

*global* state (of the entire system) vs. *local* state (of a subsystem)

The (global) state is a collection of the local state of each subsystem.

For example, consider a program with:

- Program counter variable $p$
- Boolean variables $b_1, \ldots, b_B$
- Integer variables $i_1, \ldots, i_I$
- Real variables $r_1, \ldots, r_R$

Each variable corresponds to a local state.

Their union corresponds to the (global) state of the program:

$$
\mathcal{S} \subseteq \{0, \ldots, \text{maxpc}\} \times \{0, 1\}^B \times \{\text{minint}, \ldots, \text{maxint}\}^I \times \{\text{minreal}, \ldots, \text{maxreal}\}^R
$$

think of a state as a (boolean or integer) vector
A discrete state model is fully specified by:

- potential state space \( \hat{S} \) (the “type” of the state)
- initial state \( s^{\text{init}} \in \hat{S} \) sometimes we have a set of initial states, \( S^{\text{init}} \)
- next-state function \( \mathcal{N} : S \rightarrow 2^S \) naturally extended to sets: \( \mathcal{N}(\mathcal{X}) = \bigcup_{i \in \mathcal{X}} \mathcal{N}(i) \)

The state space \( S \) of the model, assumed finite, is the smallest set satisfying:

- the recursive definition \( s^{\text{init}} \in S \) and \( i \in S \land j \in \mathcal{N}(i) \Rightarrow j \in S \)
- or the fixed-point equation \( \mathcal{X} = \{s^{\text{init}}\} \cup \mathcal{X} \cup \mathcal{N}(\mathcal{X}) \)

State \( i \) is a trap or absorbing if \( \mathcal{N}(i) = \emptyset \)

We often define a set \( \mathcal{E} \) of model events and decompose \( \mathcal{N}(i) = \bigcup_{e \in \mathcal{E}} \mathcal{N}_e(i) \)

Event \( e \) is disabled in state \( i \) if \( \mathcal{N}_e(i) = \emptyset \)
Event \( e \) is enabled in state \( i \) if \( \mathcal{N}_e(i) \neq \emptyset \), we write \( i \stackrel{e}{\rightarrow} \) or \( e \in \mathcal{E}(i) \)

If \( j \in \mathcal{N}_e(i) \), we write \( i \stackrel{e}{\rightarrow} j \)
$\mathcal{N}_e(i)$ is the set of states that can nondeterministically be reached from $i$ when $e$ occurs (or fires)

If $\mathcal{N}_e(i) = \emptyset$, $e$ is disabled in $i$, otherwise it is enabled

An example of state space with one absorbing state and one recurrent class
A Petri net is a tuple \((\mathcal{P}, \mathcal{E}, \mathbf{D}^-, \mathbf{D}^+, s)\) where:

- \(\mathcal{P}\) \quad \text{set of places, drawn as circles}
- \(\mathcal{E}\) \quad \text{set of events, or transitions, drawn as rectangles}
- \(\mathbf{D}^- : \mathcal{P} \times \mathcal{E} \to \mathbb{N}\) \quad \text{input arc cardinalities}
- \(\mathbf{D}^+ : \mathcal{P} \times \mathcal{E} \to \mathbb{N}\) \quad \text{output arc cardinalities}
- \(s_{\text{init}} \in \mathbb{N}^{\lvert \mathcal{P} \rvert}\) \quad \text{initial state, or marking}

with \(\mathcal{P} \cap \mathcal{E} = \emptyset\)

Condition for event \(e\) to be enabled in state \(i \in \mathbb{N}^{\lvert \mathcal{P} \rvert}\): \(e \in \mathcal{E}(i) \iff \forall p \in \mathcal{P}, \mathbf{D}^-_{p,e} \leq i_p\)

An event \(e\) enabled in state \(i\) can fire: \(i \xrightarrow{e} j \iff \forall p \in \mathcal{P}, j_p = i_p - \mathbf{D}^-_{p,e} + \mathbf{D}^+_{p,e}\)

Thus, \(j \in \mathcal{N}(i) \iff \exists e \in \mathcal{E}, i \xrightarrow{e} j\)

The state space, or reachability set, \(\mathcal{S}\) is defined as usual
Graphical representation of a Petri net

- **p1**: one token
- **p2**: five tokens
- **p3**: output arc
- **t1**: transition
- **input arc**: input arc with cardinality three
- **place**
**Enabling rule**

$e \in \mathcal{E}(i)$ iff each input arc contains at least as many tokens as the cardinality of the input arc:

$$\forall p \in \mathcal{P}, \ D_{p,e}^- \leq i_p \quad \text{or, in vector form} \quad D_{\cdot,e}^- \leq i$$

**Firing rule**

If $i \xrightarrow{e} j$, we obtain $j$ by removing tokens from input places and adding tokens to output places:

$$\forall p \in \mathcal{P}, \ j_p = i_p - D_{p,e}^- + D_{p,e}^+ \quad \text{or, in vector form} \quad j = i - D_{\cdot,e}^- + D_{\cdot,e}^+ = i + D_{\cdot,e}$$

where $D = D^+ - D^-$ is the *incidence matrix*

For example, if $t_1$ fires:
If the initial state is $s^{\text{init}} = (N, 0, 0, 0, 0)$:

$S$ contains $\frac{(N + 1)(N + 2)(2N + 3)}{6}$ reachable states

For any initial state $s^{\text{init}} = (N)$:

$S$ contains $\infty$ reachable states
State-by-state generation of $\mathcal{S}$

Explore Explicit($s^{init}$, $\mathcal{N}$) is

1. $\mathcal{S} \leftarrow \{s^{init}\}$;  
   \quad $\bullet$ $\mathcal{S}$ contains the states known so far
2. $\mathcal{U} \leftarrow \{s^{init}\}$;  
   \quad $\bullet$ $\mathcal{U}$ contains the unexplored states known so far
3. while $\mathcal{U} \neq \emptyset$ do
4. \quad choose a state $i$ in $\mathcal{U}$ and remove it from $\mathcal{U}$;
5. \quad for each $j \in \mathcal{N}(i)$ do
6. \quad \quad if $j \not\in \mathcal{S}$ then
7. \quad \quad \quad $\mathcal{S} \leftarrow \mathcal{S} \cup \{j\}$;  
   \quad \quad \quad $\bullet$ search to determine whether $j$ is a new state
8. \quad \quad $\mathcal{U} \leftarrow \mathcal{U} \cup \{j\}$;  
   \quad \quad $\bullet$ remember to explore $j$ later
9. \quad end if;
10. end for;
11. end while;
12. return $\mathcal{S}$;

the expensive operation is searching for a state (line 6)
How can we store $S$ and $U$ efficiently?

If we store $S$ and $U$ together, we can distinguish them using a linked list for $U$.

$\Rightarrow$ Additional $2 \cdot |U| \cdot B_{\text{pointer}}$ bits.
Or a pointer the next unexplored state, in each tree node

⇒ Additional $|S| \cdot B_{pointer}$ bits
Or store the states in a dynamic array structure

⇒ Additional $|\mathcal{S}| \cdot B_{index}$ bits
If we store $S$ and $U$ separately:
A multilevel data structure to store memory requirements: little over 3, 6, or 12 bytes per state.
Compressing $S$ after generation

Once $S$ has been built, we can compress it using arrays:

Level 1

Level 2

Level $K$

the distance $\psi(i)$ is the lexicographic index of $i$ in $S$

memory requirements: little over 1, 2, or 4 bytes per state
Example for results: a flexible manufacturing system
Results for statespace generation

Bytes for tree storage (FMS)

Number of states
- Single
- Multi

Time (FMS)

Single Splay
Single AVL
Multi Splay
Multi AVL
Results for compression and state search

Bytes for compressed storage (FMS)

Seconds to search 100,000 states (FMS)
CAN WE DO BETTER THAN THIS?
Explicit generation of $S$ adds **one state** at a time

With decision diagrams, we add **sets of states** at each step

memory increases monotonically

memory expands and shrinks
ExploreSymbolic($s^{init}, N$) is

1. $S \leftarrow \{ s^{init} \};$
2. repeat
3. $O \leftarrow S;$
4. $S \leftarrow O \cup N(O);$  
   • old set of known states  
   • current set of known states
5. until $O = S;$
6. return $S;$

with decision diagrams, these set operations can be efficient
Definition of *(RO)BDD*, a *canonical* representation of boolean functions:

- There is a single root node $r$
- Each non-terminal node is labeled with a boolean variable $x_k \in \{x_K, \ldots, x_1\}$
- Terminal nodes are labeled 0 or 1
- A non-terminal node has two outgoing arcs, labeled 0 and 1
- An arc from a node labeled $x_k$ points to a node labeled $x_l$, $k > l$
- Two nodes labeled $x_k$ cannot have the same pattern of children (*no duplicates*)
- The two children of a node are different (*no redundant nodes*)

\[
\begin{align*}
X_4X_3X_2X_1 &+ (X_4 + X_3)(X_2 + X_1) \\
X_3X_2X_1 &+ X_3(X_2 + X_1) \\
X_2X_1 &+ X_2 + X_1
\end{align*}
\]
A partition of a discrete-state model is \textit{consistent} if:

- the next-state function is partitioned into

  \[
  \mathcal{N} = \bigcup_{e \in \mathcal{E}} \mathcal{N}_e
  \]

- the global state \( i \) is partitioned into \( K \) local states

  \[
  i = (i_K, \ldots, i_1)
  \]

- so that

  \[
  S \subseteq \hat{S} = S_K \times \cdots \times S_1
  \]

- and, more importantly,

  \[
  \mathcal{N}_e(i) = \mathcal{N}_{e, K}(i_K) \times \cdots \times \mathcal{N}_{e, 1}(i_1)
  \]

\textit{a very mild requirement in practice:}

\textit{for Petri nets, any partition of the places into \( K \) subsets will do!}
Nodes are organized into $K + 1$ levels

- Level $K$ contains only one root node
- Levels $K - 1$ through 1 contain one or more nodes
- Level 0 contains the only two terminal nodes, 0 and 1 (false and true).

For $k > 0$, a node at level $k$ has $|S_k|$ arcs pointing to nodes at level $k - 1$

No duplicate nodes

$S_4 = \{0, 1, 2, 3\}$

$S_3 = \{0, 1, 2\}$

$S_2 = \{0, 1\}$

$S_1 = \{0, 1, 2\}$

$S = \{ 0 1 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 \}$
The **Union** operator for MDDs

If the MDDs $a$ and $b$ encode the sets $\mathcal{A}$ and $\mathcal{B}$, $\text{Union}(a, b)$ returns the MDD encoding $\mathcal{A} \cup \mathcal{B}$.

The function $\text{Intersection}(a, b)$ differs from $\text{Union}(a, b)$ only in the terminal case:

**Union:**

if $k = 0$ then return $a \lor b$;

**Intersection:**

if $k = 0$ then return $a \land b$;

worst case complexity: $\#\text{nodes}(a) \times \#\text{nodes}(b)$

### mdd `Union(lvl k, mdd a, mdd b)` is

1. if $k = 0$ then return $a \lor b$; • $a$ and $b$ are 0 or 1
2. if $a = b$ then return $a$;
3. if Cache contains entry $\langle (k, a, b) = u \rangle$ then return $u$;
4. for $i = 0$ to $n_k - 1$ do
5. \hspace{1em} $q_i \leftarrow \text{Union}(k-1, a[i], b[i])$;
6. end for
7. $u \leftarrow \text{Unique Table Insert}(k, q_0, \ldots, q_{n_k-1})$;
8. enter $\langle (k, a, b) = u \rangle$ in Cache;
9. return $u$;

**Unique Table:**
determines whether a node we just created is a duplicate

**Operation Cache:**
achieves efficiency. If we did not look it up we would potentially travel every path instead of visit every node in the MDD
Details of event firing

\[ S : \begin{cases} [0, 0, *, 0, 0, *] \\ [2, 0, *, 0, 0, *] \\ [3, 1, 0, 0, 0, *] \end{cases} \]

\[ S : \begin{cases} [0, 0, *, 0, 0, *] \\ [0, 0, 0, 1, 1, *] \end{cases} \]
Using structural information to encode $\mathcal{N}$ \hspace{1cm} (K = 5)

<table>
<thead>
<tr>
<th></th>
<th>$S_5 = ?$</th>
<th>$S_4 = ?$</th>
<th>$S_3 = ?$</th>
<th>$S_2 = ?$</th>
<th>$S_1 = ?$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathcal{N}_{a,5}$:</td>
<td>?</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\mathcal{N}_{a,4}$:</td>
<td>?</td>
<td>$\mathcal{N}_{b,4}$:</td>
<td>$\mathcal{N}_{c,4}$:</td>
<td>$\mathcal{N}_{e,5}$:</td>
<td></td>
</tr>
<tr>
<td>$\mathcal{N}_{a,3}$:</td>
<td>$\mathcal{N}_{b,3}$:</td>
<td>$\mathcal{N}_{c,3}$:</td>
<td>$\mathcal{N}_{e,3}$:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\mathcal{N}_{a,2}$:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\mathcal{N}_{d,2}$:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Diagram:

- $p \rightarrow a$
- $q \rightarrow a$
- $s \rightarrow a$
- $b \rightarrow c$
- $c \rightarrow d$
- $r \rightarrow e$
- $t \rightarrow e$

Top:
- $Top(a): 5$
- $Top(b): 4$
- $Top(c): 4$
- $Top(d): 2$
- $Top(e): 5$

Bot:
- $Bot(a): 2$
- $Bot(b): 3$
- $Bot(c): 3$
- $Bot(d): 1$
- $Bot(e): 1$
The resulting Kronecker encoding of $\mathcal{N}$ \hspace{1cm} (K = 5)

$S_5 = \{0, 1\}$ \hspace{1cm} $S_4 = \{0, 1\}$ \hspace{1cm} $S_3 = \{0, 1\}$ \hspace{1cm} $S_2 = \{0, 1\}$ \hspace{1cm} $S_1 = \{0, 1\}$

| $\mathcal{N}_{a,5}$: $\begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}$ | $\mathcal{N}_{a,4}$: $\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$ | $\mathcal{N}_{b,4}$: $\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$ | $\mathcal{N}_{c,4}$: $\begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}$ | $\mathcal{N}_{e,5}$: $\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$ |
| $\mathcal{N}_{a,3}$: $\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$ | $\mathcal{N}_{b,3}$: $\begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}$ | $\mathcal{N}_{c,3}$: $\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$ | $\mathcal{N}_{e,3}$: $\begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}$ |
| $\mathcal{N}_{a,2}$: $\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$ | $\mathcal{N}_{d,2}$: $\begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}$ | $\mathcal{N}_{d,1}$: $\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$ | $\mathcal{N}_{e,1}$: $\begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}$ |

$Top(a): 5 \hspace{1cm} Top(b): 4 \hspace{1cm} Top(c): 4 \hspace{1cm} Top(d): 2 \hspace{1cm} Top(e): 5$
$Bot(a): 2 \hspace{1cm} Bot(b): 3 \hspace{1cm} Bot(c): 3 \hspace{1cm} Bot(d): 1 \hspace{1cm} Bot(e): 1$
Using structural information to encode $\mathcal{N}$ ($K = 4$)

$Top(b) = Bot(b) = Top(c) = Bot(c) = 3$: merge $b$ and $c$ into a single local event $l$

$S_4 = ?$ \hspace{1cm} $S_3 = ?$ \hspace{1cm} $S_2 = ?$ \hspace{1cm} $S_1 = ?$

\[
\begin{array}{|c|c|c|}
\hline
\mathcal{N}_{a,4} : ? & & \mathcal{N}_{e,4} : ? \\
\hline
\mathcal{N}_{a,3} : ? & \mathcal{N}_{l,3} : ? & \mathcal{N}_{e,3} : ? \\
\hline
\mathcal{N}_{a,2} : ? & & \mathcal{N}_{d,2} : ? \\
\hline
& \mathcal{N}_{d,1} : ? & \mathcal{N}_{e,1} : ? \\
\hline
\end{array}
\]

$Top(a) : 4$ \hspace{1cm} $Top(l) : 3$ \hspace{1cm} $Top(d) : 2$ \hspace{1cm} $Top(e) : 4$

$Bot(a) : 2$ \hspace{1cm} $Bot(l) : 3$ \hspace{1cm} $Bot(d) : 1$ \hspace{1cm} $Bot(e) : 1$
The resulting Kronecker encoding of $\mathcal{N}$ $(K = 4)$

$S_4 = \{0,1\}$  
$S_3 = \{(0q,0r),(1q,0r),(0q,1r)\} = \{0,1,2\}$  
$S_2 = \{0,1\}$  
$S_1 = \{0,1\}$

<table>
<thead>
<tr>
<th>$N_{a,4}$ : $\begin{bmatrix} 0 &amp; 0 \ 1 &amp; 0 \end{bmatrix}$</th>
<th>$N_{e,4}$ : $\begin{bmatrix} 0 &amp; 1 \ 0 &amp; 0 \end{bmatrix}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N_{a,3}$ : $\begin{bmatrix} 0 &amp; 1 &amp; 0 \ 0 &amp; 0 &amp; 0 \ 0 &amp; 0 &amp; 0 \end{bmatrix}$</td>
<td>$N_{l,3}$ : $\begin{bmatrix} 0 &amp; 0 &amp; 0 \ 0 &amp; 0 &amp; 1 \ 0 &amp; 1 &amp; 0 \end{bmatrix}$</td>
</tr>
<tr>
<td>$N_{a,2}$ : $\begin{bmatrix} 0 &amp; 1 \ 0 &amp; 0 \end{bmatrix}$</td>
<td>$N_{d,2}$ : $\begin{bmatrix} 0 &amp; 1 \ 0 &amp; 0 \end{bmatrix}$</td>
</tr>
<tr>
<td>$N_{d,1}$ : $\begin{bmatrix} 0 &amp; 1 \ 0 &amp; 0 \end{bmatrix}$</td>
<td>$N_{e,1}$ : $\begin{bmatrix} 0 &amp; 0 \ 1 &amp; 0 \end{bmatrix}$</td>
</tr>
</tbody>
</table>

**Top** : 4  
**Bot** : 2  
**Top** : 3  
**Bot** : 3  
**Top** : 2  
**Bot** : 1  
**Top** : 4  
**Bot** : 1
Definition of Kronecker product

Given $K$ matrices $A_k \in \mathbb{R}^{n_k \times n_k}$, their Kronecker product is

$$A = \bigotimes_{k=1}^{K} A_k \in \mathbb{R}^{n_1:K \times n_{1:K}}$$

where we define $n_{l:k} = n_l \cdot n_{l+1} \cdots n_k$ and

- $A[i, j] = A_1[i_1, j_1] \cdot A_2[i_2, j_2] \cdots A_K[i_K, j_K]$
- using the mixed-base numbering scheme (indices start at 0)

$$i = \left(\ldots((i_1 \cdot n_2 + i_2) \cdot n_3 \cdots) \cdot n_K + i_K = \sum_{k=1}^{K} i_k \cdot n_{k+1:K}\right)$$

nonzeros: $\eta \left( \bigotimes_{k=1}^{K} A_k \right) = \prod_{k=1}^{K} \eta(A_k)$
Kronecker product by example

Given \( A = \begin{bmatrix} a_{0,0} & a_{0,1} \\ a_{1,0} & a_{1,1} \end{bmatrix} \), \( B = \begin{bmatrix} b_{0,0} & b_{0,1} & b_{0,2} \\ b_{1,0} & b_{1,1} & b_{1,2} \\ b_{2,0} & b_{2,1} & b_{2,2} \end{bmatrix} \),

\[
A \otimes B = \begin{bmatrix}
\frac{a_{0,0}B}{a_{1,0}B} & \frac{a_{0,1}B}{a_{1,1}B}
\end{bmatrix} = \\
\begin{bmatrix}
 a_{0,0}b_{0,0} & a_{0,0}b_{0,1} & a_{0,0}b_{0,2} & a_{0,1}b_{0,0} & a_{0,1}b_{0,1} & a_{0,1}b_{0,2} \\
 a_{0,0}b_{1,0} & a_{0,0}b_{1,1} & a_{0,0}b_{1,2} & a_{0,1}b_{1,0} & a_{0,1}b_{1,1} & a_{0,1}b_{1,2} \\
 a_{0,0}b_{2,0} & a_{0,0}b_{2,1} & a_{0,0}b_{2,2} & a_{0,1}b_{2,0} & a_{0,1}b_{2,1} & a_{0,1}b_{2,2} \\
 a_{1,0}b_{0,0} & a_{1,0}b_{0,1} & a_{1,0}b_{0,2} & a_{1,1}b_{0,0} & a_{1,1}b_{0,1} & a_{1,1}b_{0,2} \\
 a_{1,0}b_{1,0} & a_{1,0}b_{1,1} & a_{1,0}b_{1,2} & a_{1,1}b_{1,0} & a_{1,1}b_{1,1} & a_{1,1}b_{1,2} \\
 a_{1,0}b_{2,0} & a_{1,0}b_{2,1} & a_{1,0}b_{2,2} & a_{1,1}b_{2,0} & a_{1,1}b_{2,1} & a_{1,1}b_{2,2}
\end{bmatrix}
\]

Kronecker product expresses \textit{contemporaneity} or \textit{synchronization}

If \( A \) and \( B \) are the transition probability matrices of two independent discrete-time Markov chains, \( A \otimes B \) is the transition probability matrix of their composition.
Kronecker description of the next-state function

\( \mathcal{N}_{e,k} : \mathcal{S}_k \rightarrow 2^{\mathcal{S}_k} \) can be identified with a boolean matrix \( \mathbf{T}_{e,k} \in \{0, 1\}^{\lvert \mathcal{S}_k \rvert \times \lvert \mathcal{S}_k \rvert} \)

(a missing \( \mathcal{N}_{e,k} \) corresponds to the identity matrix \( \mathbf{I} \) of size \( \lvert \mathcal{S}_k \rvert \times \lvert \mathcal{S}_k \rvert \))

analogously, \( \mathcal{N} : \mathcal{S} \rightarrow 2^{\mathcal{S}} \) can be identified with a boolean matrix \( \hat{\mathbf{T}} \in \{0, 1\}^{\lvert \hat{\mathcal{S}} \rvert \times \lvert \hat{\mathcal{S}} \rvert} \)

Then,

\[
\hat{\mathbf{T}} = \sum_{e \in \mathcal{E}} \left( \bigotimes_{K \geq k \geq 1} \mathbf{T}_{e,k} \right)
\]

encode a huge \( \mathbf{T} \) with a few “small” matrices

“Complexity of memory-efficient Kronecker operations with applications to the solution of Markov models”

Buchholz, Ciardo, Donatelli, Kemper (INFORMS J. Comp., 2000)
If $i \in S$, $i \xrightarrow{e} j$, $\text{Top}(e) = k \land \text{Bot}(e) = l$: 

$$j = (i_K, \ldots, i_{k+1}, j_k, \ldots, j_l, i_{l-1}, \ldots, i_1)$$

If also $i' \in S$ and $(i_k, \ldots, i_1) = (i'_k, \ldots, i'_1)$: $i' \xrightarrow{e} j'$ \land $j' = (i'_K, \ldots, i'_{k+1}, j_k, \ldots, j_l, i_{l-1}, \ldots, i_1)$

Local event $i_k \xrightarrow{e} j_k$

Synchronizing event $(i_k, \ldots, i_l) \xrightarrow{e} (j_k, \ldots, j_k)$

*locality* and *in-place-updates* save huge amounts of computation
Saturation: an efficient iteration strategy

Traditional application of a partitioned $\mathcal{N}$:

$$
\begin{align*}
\mathcal{X}^{(e_1)} & \leftarrow \mathcal{N}_{e_1}(\mathcal{S}) \\
\cdots & \cdots \\
\mathcal{X}^{(e_{|\varepsilon|})} & \leftarrow \mathcal{N}_{e_{|\varepsilon|}}(\mathcal{S}) \\
\mathcal{S} & \leftarrow \mathcal{S} \cup \mathcal{X}^{(e_1)} \cup \ldots \cup \mathcal{X}^{(e_{|\varepsilon|})}
\end{align*}
$$

We can improve by pipelining:

$$
\begin{align*}
\mathcal{S} & \leftarrow \mathcal{S} \cup \mathcal{N}_{e_1}(\mathcal{S}) \\
\cdots & \cdots \\
\mathcal{S} & \leftarrow \mathcal{S} \cup \mathcal{N}_{e_{|\varepsilon|}}(\mathcal{S})
\end{align*}
$$

And even more by exhaustive pipelining:

$$
\begin{align*}
\mathcal{S} & \leftarrow \mathcal{S} \cup \mathcal{N}^*_{e_1}(\mathcal{S}) \\
\cdots & \cdots \\
\mathcal{S} & \leftarrow \mathcal{S} \cup \mathcal{N}^*_{e_{|\varepsilon|}}(\mathcal{S})
\end{align*}
$$

But the best strategy is to saturate MDD nodes recursively bottom-up:

- a node at level $k$ is saturated if it is a fixed point w.r.t. all events $e$ s.t. $\text{Top}(e) \leq k$
- traditional idea of a global fixed-point iteration for the overall MDD disappears

enormous savings in both time and (peak) memory
Problem: local state spaces $S_k$ are not known a priori

Solution: build $S_k$ “on the fly” (explicitly) alongside the overall state space $S$ (symbolically)

1. start from the only known state, the initial state $(s_K, \ldots, s_1)$, and commit its components

2. while MDD encoding $S$ has not reached its fixed point w.r.t. $\mathcal{N}$ do

3. (explicitly) explore all $j_k$ reachable from each newly committed $i_k$ in isolation in one step
   $\Rightarrow$ create corresponding row $i_k$ of $\mathcal{N}_{e,k}$ for each $e \in \mathcal{E}$ dependent on level $k$

4. (symbolically) explore global states reachable from the currently-known $S$
   $\Rightarrow$ use current $\mathcal{N}_{e,k}$ matrices
   $\Rightarrow$ may cause uncommitted local states to be committed

5. end while

no need to know a priori the range of each state variable
$N$ subnets connected in a circular fashion
Example: the dining philosophers (SMART code)

```plaintext
spn phils(int N) := {
    for (int i in {0..N-1}) {
        place Idle[i], WaitL[i], WaitR[i], HasL[i], HasR[i], Fork[i];
        partition (i+1:Idle[i]:WaitL[i]:WaitR[i]:HasL[i]:HasR[i]:Fork[i]);
        trans GoEat[i], GetL[i], GetR[i], Release[i];
        firing (GoEat[i]:expo(1), GetL[i]:expo(1), GetR[i]:expo(1), Release[i]:expo(1));
        init (Idle[i]:1, Fork[i]:1);
    }
    for (int i in {0..N-1}) {
        arcs (Idle[i]:GoEat[i], GoEat[i]:WaitL[i], GoEat[i]:WaitR[i],
             WaitL[i]:GetL[i], Fork[i]:GetL[i], GetL[i]:HasL[i],
             WaitR[i]:GetR[i], Fork[mod(i+1, N)]:GetR[i], GetR[i]:HasR[i],
             HasL[i]:Release[i], HasR[i]:Release[i], Release[i]:Idle[i],
             Release[i]:Fork[i], Release[i]:Fork[mod(i+1, N)]);
    }
    bigint num := card(reachable);
    stateset g := EF(initialstate);
    bigint numg := card(g);
    stateset b := difference(reachable, g);
    void out := printset(b);
};

int N := read_int("number of philosophers");
print("N=", N, "\n");
print("Reachable states: ", phils(N).num, "\n");
print("Good states: ", phils(N).numg, "\n");
print("The bad states are\n"); phils(N).out;
```
Example: the dining philosophers (results)

Reading input.
N=50
Reachable states: 22,291,846,172,619,859,445,381,409,012,498
Good states: 22,291,846,172,619,859,445,381,409,012,496
The bad states are

State 0 : { WaitR[0]:1 HasL[0]:1 WaitR[1]:1 HasL[1]:1 WaitR[2]:1 HasL[2]:1 WaitR[3]:1 HasL[3]:1 WaitR[4]:1 HasL[4]:1 WaitR[5]:1 HasL[5]:1 ... WaitR[45]:1 HasL[45]:1 WaitR[46]:1 HasL[46]:1 WaitR[47]:1 HasL[47]:1 WaitR[48]:1 HasL[48]:1 WaitR[49]:1 HasL[49]:1 }
State 1 : { WaitL[0]:1 HasR[0]:1 WaitL[1]:1 HasR[1]:1 WaitL[2]:1 HasR[2]:1 WaitL[3]:1 HasR[3]:1 WaitL[4]:1 HasR[4]:1 WaitL[5]:1 HasR[5]:1 ... WaitL[45]:1 HasR[45]:1 WaitL[46]:1 HasR[46]:1 WaitL[47]:1 HasR[47]:1 WaitL[48]:1 HasR[48]:1 WaitL[49]:1 HasR[49]:1 }

Done.
<table>
<thead>
<tr>
<th>$N$</th>
<th>Reachable states</th>
<th>Final memory (kB)</th>
<th>Peak memory (kB)</th>
<th>Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SMART</td>
<td>NuSMV</td>
<td>SMART</td>
<td>NuSMV</td>
</tr>
<tr>
<td><strong>Dining Philosophers ($N$ levels)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>$2.23 \times 10^{31}$</td>
<td>18</td>
<td>22</td>
<td>0.15</td>
</tr>
<tr>
<td>200</td>
<td>$2.47 \times 10^{125}$</td>
<td>74</td>
<td>93</td>
<td>0.68</td>
</tr>
<tr>
<td>10,000</td>
<td>$4.26 \times 10^{6269}$</td>
<td>3,749</td>
<td>4,686</td>
<td>877.82</td>
</tr>
<tr>
<td><strong>Slotted Ring Network ($N$ levels)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>$8.29 \times 10^{9}$</td>
<td>4</td>
<td>28</td>
<td>0.13</td>
</tr>
<tr>
<td>15</td>
<td>$1.46 \times 10^{15}$</td>
<td>10</td>
<td>80</td>
<td>0.39</td>
</tr>
<tr>
<td>200</td>
<td>$8.38 \times 10^{211}$</td>
<td>1,729</td>
<td>120,316</td>
<td>902.11</td>
</tr>
<tr>
<td><strong>Round Robin Mutual Exclusion ($N+1$ levels)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>$4.72 \times 10^{7}$</td>
<td>18</td>
<td>20</td>
<td>0.07</td>
</tr>
<tr>
<td>100</td>
<td>$2.85 \times 10^{32}$</td>
<td>356</td>
<td>372</td>
<td>3.81</td>
</tr>
<tr>
<td>300</td>
<td>$1.37 \times 10^{93}$</td>
<td>3,063</td>
<td>3,109</td>
<td>140.98</td>
</tr>
<tr>
<td><strong>Flexible Manufacturing System (19 levels)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>$4.28 \times 10^{6}$</td>
<td>16</td>
<td>26</td>
<td>0.05</td>
</tr>
<tr>
<td>20</td>
<td>$3.84 \times 10^{9}$</td>
<td>55</td>
<td>101</td>
<td>0.20</td>
</tr>
<tr>
<td>250</td>
<td>$3.47 \times 10^{26}$</td>
<td>25,507</td>
<td>69,087</td>
<td>231.17</td>
</tr>
</tbody>
</table>
Symbolic model checking

We can talk about events and states occurring over *relative time*, or *temporal logic*

- Can event $e$ ever fire before event $f$?
- Is it possible to reach a state where both buffers are empty?
- Once both buffers are empty, can they ever both become full at the same time?
- Or even just at different times?
- Can we reach a stable set of states where race conditions cannot occur?
- Can we reach a set of states where, if race conditions occur, they never cause a deadlock?

We use *computation tree logic (CTL)* to express these queries:

- Any atomic proposition (true or false in a state) is a CTL formula
- If $p$ and $q$ are CTL formulas, so are $\neg p$, $p \land q$, $p \lor q$
- If $p$ and $q$ are CTL formulas, so are $\text{EX}p$, $\text{EF}p$, $\text{EG}p$, $\text{E}[p\text{U}q]$, $\text{AX}p$, $\text{AF}p$, $\text{AG}p$, $\text{A}[p\text{U}q]$

*given a model, a CTL formula $\varphi$ identifies a set of states (those states that satisfy $\varphi$)*
CTL semantics

Note that EX, EG, and EU is a complete set of CTL operators, since

\[ EF_p = E[p \lor Up] \]
\[ AX_p = \neg EX \neg p \]
\[ AF_p = \neg EG \neg p \]
\[ AG_p = \neg E[p \lor Up] \]
\[ A[pUq] = \neg E[\neg q \lor Up \land \neg q] \land \neg EG \neg q \]
Applications

Protocol verification

Security

Software correctness

VLSI design and verification

GUI and HCI testing