Decision Diagrams to Encode and Manipulate Large Structured Data

Slides made by **Marco Beccuti**

Università degli Studi di Torino

Dipartimento di Informatica

May 2020

Outline

Storing large structured sets

How can a set of values of size 10^{100} be stored efficiently?

- If it comes from sampling, from digitizing a picture, etc ⇒ we can use compression techniques.
- But what if it comes from some kind of man-made structured artifact or abstraction?
  - *The set of a graph paths connecting two nodes*
  - *The set of reachable states during the execution of a distributed program*

Observe: These problems are described in a relatively compact way, but their answer is combinatorially large

Storing large structured sets

Why is state-space generation important?

- State-space generation is one of first steps in model quantitative and qualitative analysis;
- State-space generation is enough to answer safety queries
  - *Can we reach a "bad" state?*
  - *Is it true that VAL is greater than N whenever FLAG is On?*

Storing large structured sets

How can set of reachable states during the execution of system be stored/managed efficiently?

- aggregation based methods, where markings are grouped into classes, according to some equivalence relation;
- composition/decomposition based methods, where an efficient representation of the whole system state space is given in terms of system component state space;
- compressed hash table based methods;
- ...

Can we do better if the reachable states are highly "structured" ?

Storing large structured sets

How can the set of reachable states during the execution of system be stored/managed efficiently?

- Using symbolic (implicit) data structure where each memory location may store information about multiple states
- Using symbolic (implicit) algorithm where states are manipulated one set at a time

For instance using symbolic approach

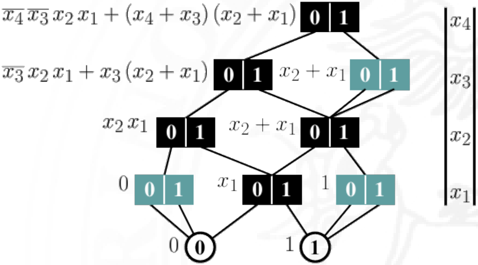
- *Rechability set of Dining philosopher model with 200 philosophers (i.e. its size is 2.46e*^{125}*)* can be stored in 6MB and computed in 2s. using Intel Core I7;
- *Rechability set of Flexible manufacturing system with 100 parts (i.e. its size is 2.70e*^{21}*)* can be stored in 170MB and computed in 1m. using Intel Core I7.

Binary Decision Diagram

DD for Boolean functions

Binary Decision Diagram

A **Binary Decision Diagram** (BDD) [1] is an acyclic directed graph used to represent functions of the form $f : \mathcal{V}_N \times \cdots \times \mathcal{V}_1 \rightarrow \{0, 1\}$, where the set of possible values for variable i is $\mathcal{V}_i = \{0, 1\}$.

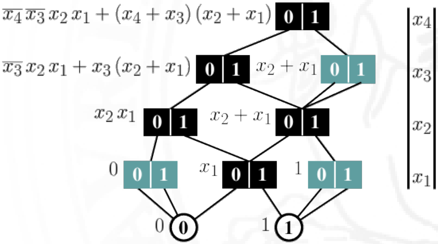


[1] Randy Bryant
Graph-based algorithms for boolean function manipulation *IEEE Transactions on Computers*, 1986 **CiteSeer most cited document!**

Binary Decision Diagram

In a canonical BDD with N variables:

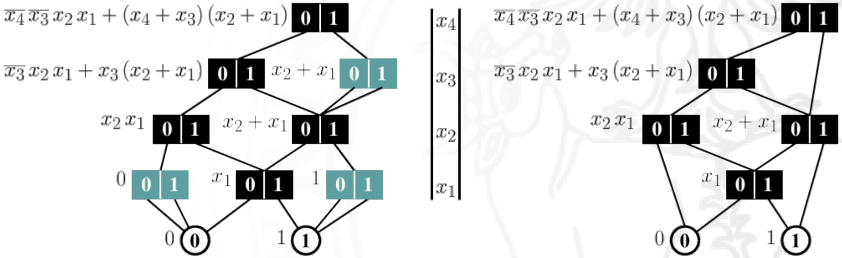
- an ordering is associated with the variables $(N, \ldots, 1)$
- the nodes are organized into $N + 1$ levels:
  - level N contains only one root node;
  - levels $N - 1, \ldots, 1$ contain one or more nodes, **no duplicates**;
  - level 0 contains only two terminal nodes, 0 and 1, corresponding to false and true.
- a non-terminal node has only two outgoing arcs one for value 0 and one for value 1



Binary Decision Diagram

A BDD is called:

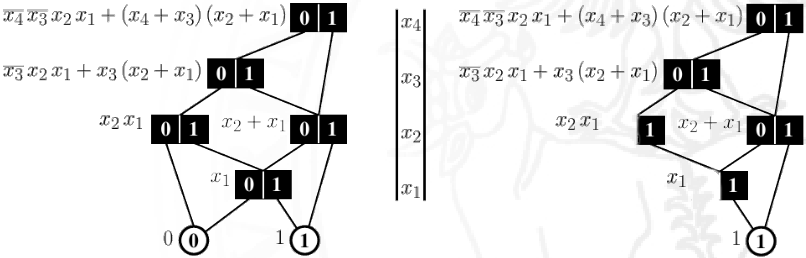
- **Quasi reduced**, if it does not contain duplicate nodes (No level skipping);
- **Fully reduced**, if it does not contain duplicate and redundant nodes (level skipping).



Binary Decision Diagram

A BDD is called:

- **Full storage**, each node stores all the variable values (e.g. vector);
- **Sparse storage**, each node stores only the variable value not directly connected to terminal node 0 (e.g list).



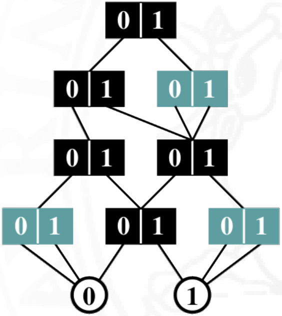
Binary Decision Diagram

We can encode a set $S = \{e_1, \ldots, e_n\}$ as an BDD P through its characteristic function:

$$e = (e^L, \ldots, e^1) \in S \Leftrightarrow v_P(e^L, \ldots, e^1) = 1$$

where v_P is a function that returns the terminal nodes reached by the P path identified through its input value $e^L, \ldots, e^1$.

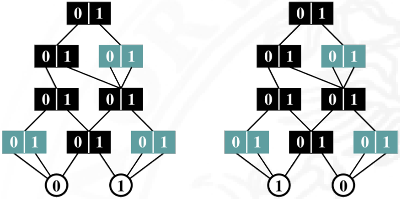
$$\{0011, 0101, 0110, 0111, 1001, \\ 1010, 1011, 1101, 1110, 1111\}$$



Binary Decision Diagram

The size of the set encoded on BDD is not directly related to the size of the BDD itself

$\{0011, 0101, 0110, 0111, 1001, 1010, 1011, 1101, 1110, 1111\}$ $\{0000, 0001, 0010, 0100, 1000, 1100\}$



The variable ordering affects the size of the BDD.

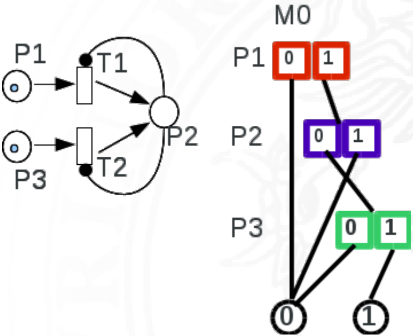
E.g. the logic function of $2n$ variables $x_1 \cdot x_2 + \ldots + x_{2n-1} \cdot x_{2n}$

- with ordering $(x_1, x_2, x_3, x_4 \ldots, x_{2n-1}, x_{2n}) \Rightarrow 2n + 2$ nodes;
- with ordering $(x_1, x_{2n-1}, x_3, x_{2n-3} \ldots, x_4, x_{2n-2}, x_2, x_{2n}) \Rightarrow 2^{n+1}$ nodes.

Find the **optimal ordering** that minimizes the BDD size is an **NP-complete** problem.

Symbolic state representation of safe PN

A simple example using a fully reduced BDD



Binary Decision Diagram

Main logical BDDs operators

- $+$ union;
- $*$ intersection;
- $==$ check for equality;

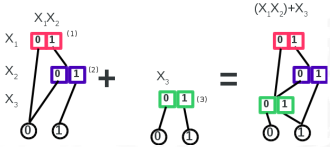
Main specific BDD operators:

- $\mathcal{EV}$ evaluates if a path is presented in the BDD;
- $\bullet$ post-image operation on the BDD paths with a **transition** function or relational product;
- $\mathcal{E}$ enumerates the paths in the BDD.

The green operators can be implemented in easy way!!

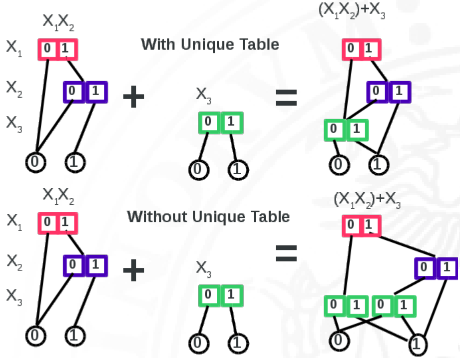
Binary Decision Diagram

An example of Union for BDD




```
procedure UnionBDD(BDDp, BDDq)
Union for Fully-Reduced
r = local BDD
    if ( p = 0 || q = 1 ) then return q;
    if ( p = 1 || q = 0 ) then return p;
    if ( p = q ) then return p;
    if (p.lvl > q.lvl) then
        r = UniqueTable.Insert(p.lvl,UnionBDD(p[0],q),UnionBDD(p[1],q));
    else
        if (p.lvl < q.lvl) then
            r = UniqueTable.Insert(p.lvl,UnionBDD(p,q[0]),UnionBDD(p,q[1]));
        else
            r = UniqueTable.Insert(p.lvl,UnionBDD(p[0],q[0]),UnionBDD(p[1],q[1]));
        end if
    end if
    return r;
end procedure
```

Binary Decision Diagram



Unique Table (UT)

To ensure no duplicate nodes, all decision diagram operations use a Unique Table (a hash table):

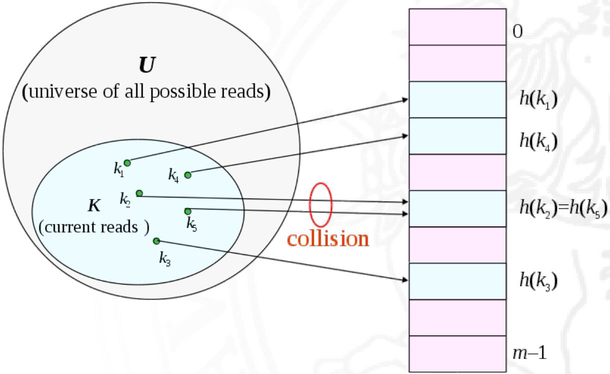
- Search key: the node's level and sequence of children's node ids;
- Return value: a node id.

With the UT, we avoid duplicate nodes

How to encode and search a node

How to efficiently find a node signature

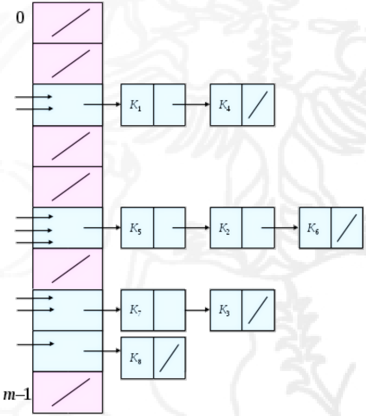
- Hash table is a good compromise in terms of memory and search cost.



How to encode and search node

How to manage collisions

- Typically a chaining collision policy is used:
  - store all elements with same hash value ($h()$) in a linked list.
  - store a pointer to the head of the linked list in the hash table slot.



Binary Decision Diagram

Union algorithm

Complexity O(product of the numbers of nodes in p and q)

Operation Cache (OC)

To achieve polynomial complexity, all operations use an Operation Cache (a hash table)[1]:

- Search key: OpCODE and sequence of operands' node ids;
- Return value: a node id.

With the OC, we consider every node combination instead of every path combination

[1] before computing OpCODE (node id1 , node id2), we search in the OC: If the search is successful, we avoid recomputing a result.

Binary Decision Diagram

```
procedure UnionBDD(BDDp, BDDq)
Union for Fully-Reduced
r = local BDD
    if ( p = 0 || q = 1 ) then return q;
    if ( p = 1 || q = 0 ) then return p;
    if ( p = q ) then return p;
    if OperationCache.Search(UNION,p,q,r) then
        return r;
    end if
    if (p.lvl > q.lvl) then
        r = UniqueTable.Insert(p.lvl,UnionBDD(p[0],q),UnionBDD(p[1],q));
    else
        if (p.lvl < q.lvl) then
            r = UniqueTable.Insert(p.lvl,UnionBDD(p,q[0]),UnionBDD(p,q[1]));
        else
            r = UniqueTable.Insert(p.lvl,UnionBDD(p[0],q[0]),UnionBDD(p[1],q[1]));
        end if
    end if
    OperationCache.insert(UNION,p,q,r);
    return r;
end procedure
```

Binary Decision Diagram

```
procedure INTERBDD(BDDp, BDDq)
Intersection for Fully-Reduced
r = local BDD
    if ( p = 0 || q = 1 ) then return p;
    if ( p = 1 || q = 0 ) then return q;
    if ( p = q ) then return p;
    if OperationCache.Search(INTERSECTION,p,q,r) then
        return r;
    end if
    if (p.lvl > q.lvl) then
        r = UniqueTable.Insert(p.lvl,InterBDD(p[0],q),InterBDD(p[1],q));
    else
        if (p.lvl < q.lvl) then
            r = UniqueTable.Insert(p.lvl,InterBDD(p,q[0]),InterBDD(p,q[1]));
        else
            r = UniqueTable.Insert(p.lvl,InterBDD(p[0],q[0]),InterBDD(p[1],q[1]));
        end if
    end if
    OperationCache.insert(INTERSECTION,p,q,r);
    return r;
end procedure
```

Intersection(p,q) differs from *Union(p,q)* only in the terminal cases:

Union	**Intersection**
if $p = 0 \cup q = 1$ then return q;	if $p = 1 \cup q = 0$ then return q;
if $q = 0 \cup p = 1$ then return p;	if $q = 1 \cup p = 0$ then return p;

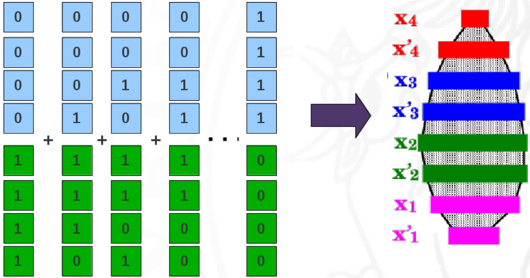
Binary Decision Diagram

How to encode relation on a set

How to encode a relation $R : I \leftarrow 2^I$ on a set $I = \{\langle x_L, \ldots, x_1 \rangle\}$:

$$R = \{(\langle x_L, \ldots, x_1 \rangle, \langle x'_L, \ldots, x'_1 \rangle)\}$$

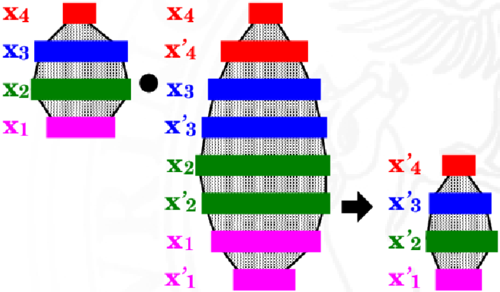
Since it is a set we can encode it on a BDD with 2L levels.



Binary Decision Diagram

The post image operator or the relational product.

Given a BDD encoding $Y = \{\langle x_L, \ldots, x_1 \rangle\}$, $Y \subseteq I$ and BDD encoding R is it possible to compute $Y' = \{\langle x'_L, \ldots, x'_1 \rangle : \exists \langle x_L, \ldots, x_1 \rangle \in Y \wedge (\langle x_L, \ldots, x_1 \rangle, \langle x'_L, \ldots, x'_1 \rangle) \in R\}$



Binary Decision Diagram

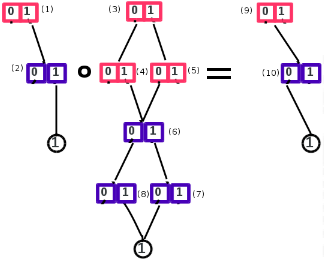
The post image operator or the relational product.
Given an L-level BDD on $(x_L, \ldots, x_1)$ rooted at p_* encoding a set $\mathcal{Y} \subseteq \mathcal{X}_{pot}$. Given a 2L-level BDD on $(x_L, x'_L, \ldots, x_1, x'_1)$ rooted at r_* encoding a function $\mathcal{N} : \mathcal{X}_{pot} \rightarrow 2^{\mathcal{X}_{pot}}$.

RelationalProduct(p_*, r_*) returns the root of the BDD encoding the set:

$$\{j : \exists i \in \mathcal{Y} \wedge j \in \mathcal{N}(i)\}$$

```
procedure RelationalProduct(BDDp, BDDq)
    RelationalProduct for Quasi-Reduced
    r,r',r" = local BDDs
    if ( p = 1 && q = 1 ) then return 1;
    if ( p = 0 || q = 0 ) then return 0;
    if OperationCache.Search(RelationalProduct,p,q,r) then
        return r;
    end if
    r' =Union(RelationalProduct(p[0],q[0][0]),RelationalProduct(p[1],q[1][0]));
    r"=Union(RelationalProduct(p[0],q[0][1]),RelationalProduct(p[1],q[1][1]));
    r = UniqueTable.Insert(p.lvl,r',r");
    OperationCache.insert(UNION,p,q,r);
    return r;
end procedure
```

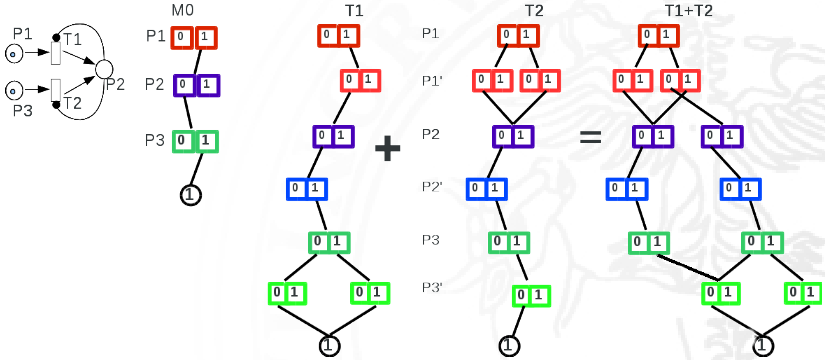
Binary Decision Diagram




```
procedure RelationalProduct(BDDp, BDDq)
    RelationalProduct for Quasi-Reduced
    r,r',r" = local BDDs
    if ( p = 1 && q = 1 ) then return 1;
    if ( p = 0 || q = 0 ) then return 0;
    if OperationCache.Search(RelationalProduct,p,q,r) then
        return r;
    end if
    r' =Union(RelationalProduct(p[0],q[0][0]),RelationalProduct(p[1],q[1][0]));
    r"=Union(RelationalProduct(p[0],q[0][1]),RelationalProduct(p[1],q[1][1]));
    r = UniqueTable.Insert(p.lvl,r',r");
    OperationCache.insert(UNION,p,q,r);
    return r;
end procedure
```

Symbolic state-space generation of PN

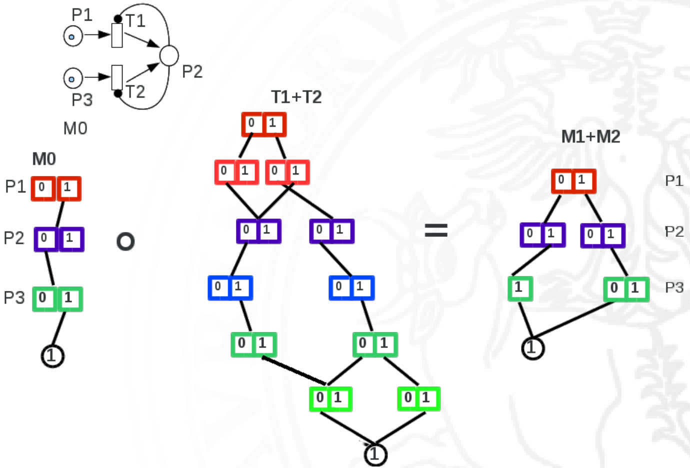
A simple example



Prime levels that do not change the variable values can be skipped

Symbolic state-space generation of PN

A simple example



Symbolic state-space generation of safe PN

We can store

- any set of markings $\mathcal{Y} \subseteq \mathcal{X}^{pot} = \mathbb{B}^{|P|}$ of a safe PN with a **$|P|$-level BDD**.
- the next state function over $\mathcal{X}^{pot}$, such as $\mathcal{N} : \mathcal{X}_{pot} \rightarrow 2^{\mathcal{X}_{pot}}$, with a **$2|P|$-level BDD**.

The state space $\mathcal{X}^{rch}$ is the fix-point of the iteration:

$$\mathcal{X}^{init} \quad \mathcal{N}(\mathcal{X}^{init}) \quad \mathcal{N}(\mathcal{N}(\mathcal{X}^{init})) \quad \mathcal{N}(\mathcal{N}(\mathcal{N}(\mathcal{X}^{init}))) \quad \ldots$$

The main operation is a repeated application of the relational product operator:

```
procedure GenerateRS(BDDX^init, BDDN)
    X = X^init;
    repeat
        O = X;
        O = Union(O,RelationalProduct(X,N));
    until (X! = O)
    return O;
end procedure
```

If state variable x^k has non-boolean domain, we use multiple boolean levels to encode it.

Explicit generation vs. symbolic generation

Explicit generation

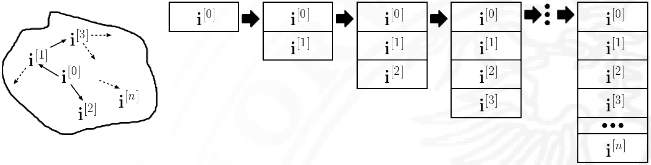
- **Explicit data structure**: each state requires a different memory location (bit, byte, word, array, etc.)$\rightarrow$ $O(|\mathcal{X}_{rch}|)$ memory.
- **Explicit algorithm**: states are manipulated one by one$\rightarrow$ $O(|\mathcal{X}_{rch}|)$ time.
  Memory requirements increase linearly as new states are found.

Symbolic generation

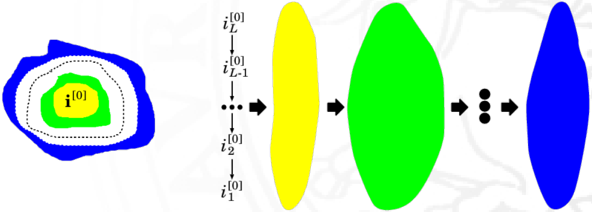
- **symbolic data structure**: each memory location may store information about multiple states $\rightarrow$ $O(|\mathcal{X}_{rch}|)$ memory only in the worst case.
- **symbolic data structure**: states are manipulated one set at a time $\rightarrow$ $O(|\mathcal{X}_{rch}|)$ time only in the worst case.
  Memory requirements grow and shrink as new states are found, peak not usually at the end

Explicit generation vs. symbolic generation

Explicit generation of the state space $\mathcal{X}_{rch}$ adds one state at a time, memory *O(states)* increases linearly, peaks at the end.



Symbolic generation of the state space $\mathcal{X}_{rch}$ with decision diagrams adds sets of states instead, memory *O(decision diagram nodes)*, grows and shrinks, usually peaks well before the end



Multi-way Decision Diagram

DD for integer functions

Multi-way Decision Diagram

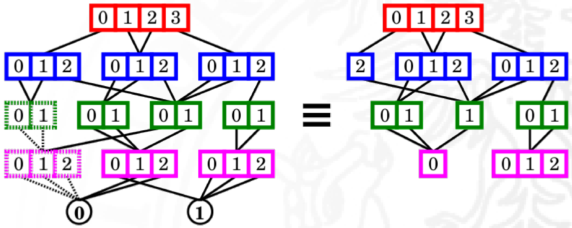
A **Multi-way Decision Diagram** (MDD) [1] is an acyclic directed graph used to represent functions of the form $f : \mathcal{V}_N \times \cdots \times \mathcal{V}_1 \to \{0,1\}$, where the set of possible values for variable i is $\mathcal{V}_i = \{0, 1, \ldots, |\mathcal{V}_i| - 1\}$.

$\mathcal{S}_4 = \{0,1,2,3\}$

$\mathcal{S}_3 = \{0,1,2\}$

$\mathcal{S}_2 = \{0,1\}$

$\mathcal{S}_1 = \{0,1,2\}$



$$\mathcal{S} = \left\{ \begin{matrix}0\\2\\1\\0\end{matrix}, \begin{matrix}1\\0\\0\\0\end{matrix}, \begin{matrix}1\\0\\1\\0\end{matrix}, \begin{matrix}1\\1\\0\\0\end{matrix}, \begin{matrix}1\\1\\1\\0\end{matrix}, \begin{matrix}1\\2\\1\\0\end{matrix}, \begin{matrix}2\\0\\0\\0\end{matrix}, \begin{matrix}2\\0\\1\\0\end{matrix}, \begin{matrix}2\\1\\0\\0\end{matrix}, \begin{matrix}2\\1\\1\\0\end{matrix}, \begin{matrix}2\\2\\1\\0\end{matrix}, \begin{matrix}3\\0\\1\\0\end{matrix}, \begin{matrix}3\\1\\1\\0\end{matrix}, \begin{matrix}3\\2\\0\\0\end{matrix}, \begin{matrix}3\\2\\0\\1\end{matrix}, \begin{matrix}3\\2\\0\\2\end{matrix}, \begin{matrix}3\\2\\1\\0\end{matrix}, \begin{matrix}3\\2\\1\\1\end{matrix}, \begin{matrix}3\\2\\1\\2\end{matrix} \right\}$$

[1] T. Kam, T. Villa, R. Brayton, and A. Sangiovanni-Vincentelli
Multi-valued decision diagrams: theory and applications *Multiple-Valued Logic*, 1998

Multi-way Decision Diagram

In a canonical MDD with N variables:

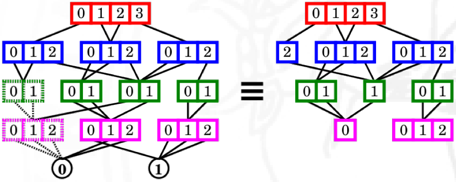
- an ordering is associated with the variables;
- nodes are organized into $N+1$ levels:
  - level N contains only one root node;
  - levels $N-1,\ldots,1$ contain one or more nodes, **no duplicates**;
  - level 0 contains only two terminal nodes, 0 and 1, corresponding to false and true.
- for $i>0$, a S node at level i has $|S_i|$ arcs pointing to nodes at level $i-1$.

$\mathcal{S}_4=\{0,1,2,3\}$

$\mathcal{S}_3=\{0,1,2\}$

$\mathcal{S}_2=\{0,1\}$

$\mathcal{S}_1=\{0,1,2\}$

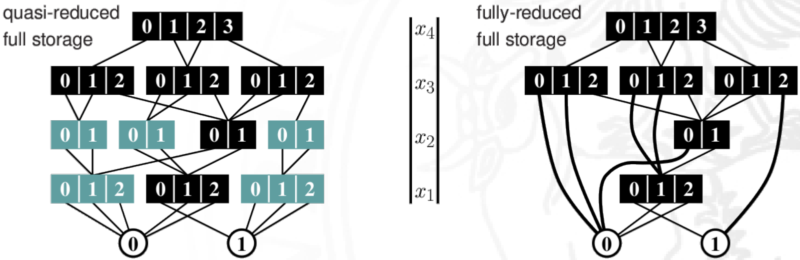


$$\mathcal{S}=\left\{\begin{smallmatrix}0\\2\\1\\0\end{smallmatrix},\begin{smallmatrix}1\\0\\0\\0\end{smallmatrix},\begin{smallmatrix}1\\0\\1\\0\end{smallmatrix},\begin{smallmatrix}1\\1\\0\\0\end{smallmatrix},\begin{smallmatrix}1\\1\\1\\0\end{smallmatrix},\begin{smallmatrix}1\\2\\1\\0\end{smallmatrix},\begin{smallmatrix}2\\0\\0\\0\end{smallmatrix},\begin{smallmatrix}2\\0\\1\\0\end{smallmatrix},\begin{smallmatrix}2\\1\\0\\0\end{smallmatrix},\begin{smallmatrix}2\\1\\1\\0\end{smallmatrix},\begin{smallmatrix}2\\2\\1\\0\end{smallmatrix},\begin{smallmatrix}3\\0\\1\\0\end{smallmatrix},\begin{smallmatrix}3\\1\\1\\0\end{smallmatrix},\begin{smallmatrix}3\\2\\0\\0\end{smallmatrix},\begin{smallmatrix}3\\2\\0\\1\end{smallmatrix},\begin{smallmatrix}3\\2\\0\\2\end{smallmatrix},\begin{smallmatrix}3\\2\\1\\0\end{smallmatrix},\begin{smallmatrix}3\\2\\1\\1\end{smallmatrix},\begin{smallmatrix}3\\2\\1\\2\end{smallmatrix}\right\}$$

Multi-way Decision Diagram

A MDD is called:

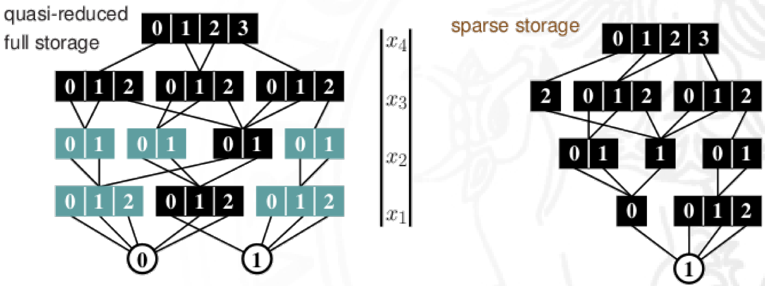
- **Quasi reduced**, if it does not contain duplicate nodes (No level skipping);
- **Fully reduced**, if it does not contain duplicate and redundant nodes (Maximum level skipping).



Multi-way Decision Diagram

A MDD is called:

- **Full storage**, each node stores all the variable values (e.g. vector);
- **Sparse storage**, each node store only the variable value not directly connected to terminal node 0 (e.g list).



Multi-way Decision Diagram

Main logical MDDs operators

- $+$ union;
- $*$ intersection;
- $==$ check for equality;

Main specific MDD operators:

- $\mathcal{EV}$ evaluates if a path is presented in the MDD;
- post-image operation on the MDD paths with a **transition** function or relational product;
- $\mathcal{E}$ enumerates the paths in the MDD.

Multi-way Decision Diagram

Union or intersection for MDD

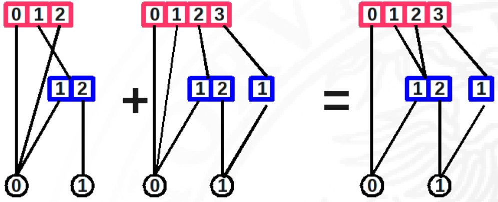
```
procedure UNIONMDD(MDDp, MDDq)
    Union for Quasi-Reduced
    r, r_1, ..., r_n = local MDD
    if ( p = 0 || q = 1 ) then return q;
    if ( p = 1 || q = 0 ) then return p;
    if ( p = q ) then return p;
    if OperationCache.Search(UNION,p,q,r) then
        return r;
    end if
    for i ∈ {1, ..., n} do
        r_i = UnionBDD(p[i],q[i]);
    end for
    r = UniqueTable.Insert(p.lvl, r_1, ..., r_n);
    OperationCache.insert(UNION,p,q,r);
    return r;
end procedure
```

Intersection(p,q) differs from *Union(p,q)* only in the terminal cases:

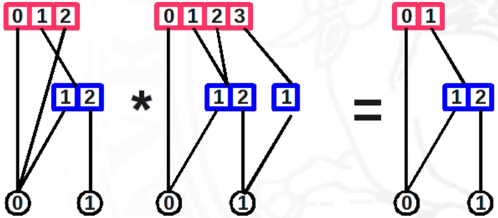
Union	**Intersection**
if $p = 0$ then return q;	if $p = 1$ then return q;
if $q = 0$ then return p;	if $q = 1$ then return p;

Multi-way Decision Diagram

An example of MDDs' union



An example of MDDs' intersection

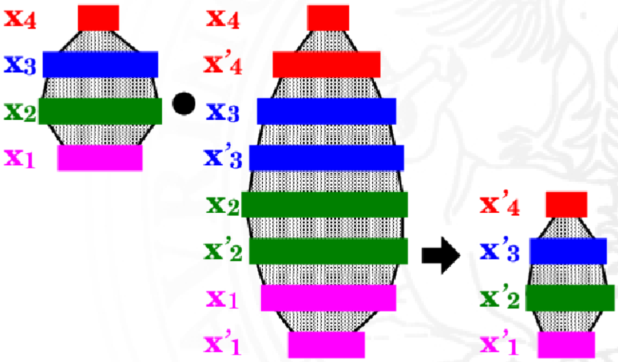


Multi-way Decision Diagram

The post image operator or the relational product.
Given an L-level MDD on $(x_L, \ldots, x_1)$ rooted at p_* encoding a set $\mathcal{Y} \subseteq \mathcal{X}_{pot}$. Given a 2L-level MDD on $(x_L, x'_L, \ldots, x_1, x'_1)$ rooted at r_* encoding a function $\mathcal{N} : \mathcal{X}_{pot} \rightarrow 2^{\mathcal{X}_{pot}}$.

RelationalProduct(L, p_*, r_*) returns the root of the MDD encoding the set:

$$\{j : \exists i \in \mathcal{Y} \wedge j \in \mathcal{N}(i)\}$$



Symbolic state-space generation of PN

We can store

- any set of markings $\mathcal{Y} \subseteq \mathcal{X}^{pot} = \mathbb{N}^{|P|}$ of a PN with a **$|P|$-level MDD**.
- the next state function over $\mathcal{X}^{pot}$, such as $\mathcal{N} : \mathcal{X}_{pot} \rightarrow 2^{\mathcal{X}_{pot}}$, with a **$2|P|$-level MDD**.

The state space $\mathcal{X}^{rch}$ is the fix-point of the iteration:

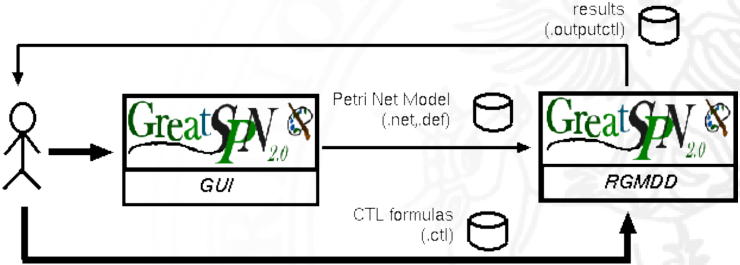
$$\mathcal{X}^{init} \quad \mathcal{N}(\mathcal{X}^{init}) \quad \mathcal{N}(\mathcal{N}(\mathcal{X}^{init})) \quad \mathcal{N}(\mathcal{N}(\mathcal{N}(\mathcal{X}^{init}))) \quad \ldots$$

The main operation is a repeated application of the relational product operator:

```
procedure GenerateRS(MDDX^init, MDDN)
    X = X^init;
    repeat
        O = X;
        O = Union(O,RelationalProduct(X,N));
    until (X! = O)
    return O;
end procedure
```

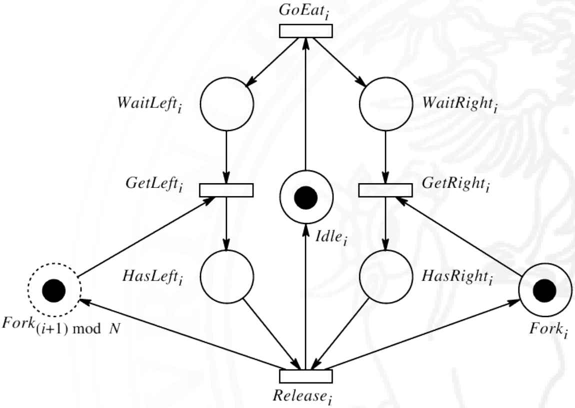
Experiments on Dining Philosophers

The symbolic algorithms have been implemented in GreatSPN using Meddly library (http://meddly.svn.sourceforge.net/)



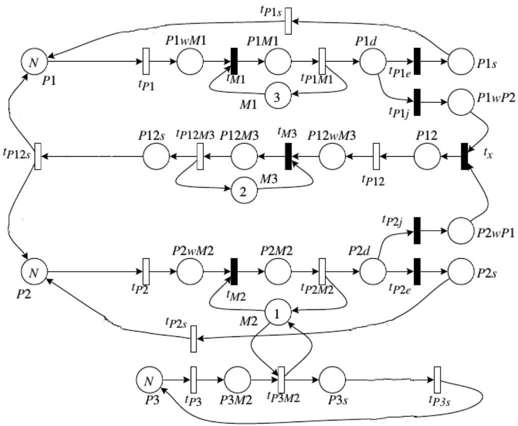
Experiments on Dining Philosophers

A single dining philosopher



Experiments on Flexible Manufacturing System

FMS



Some experimental results using GreatSPN

N	$\|\mathcal{S}\|$	GreatSPN BBT	GreatSPN File	GreatSPN T.	No Sat. Mem.	No Sat. T.
Dining philosophers Petri net						
7	2.4×10^4	1,175KB	1,027KB	8s	51KB	8s
8	1.0×10^5	4,977KB	4,976KB	45s	75KB	50s
9	4.3×10^5	21,082KB	23,717KB	345s	103KB	411s
10	1.8×10^6	89,304KB	104,654KB	31m	139KB	28m
11	7.8×10^6	—	—	—	185KB	82m
12	3.3×10^7	—	—	—	235KB	7h
Flexible manufacturing system Petri net						
4	$1,3 \times 10^5$	6,627KB	3,037KB	11s	363KB	14s
5	6.5×10^5	31,466KB	14,421KB	134s	775KB	63s
6	2.5×10^6	120,940KB	55,430KB	7m	1.470KB	3m
7	8.2×10^6	—	—	—	2.536KB	12m
8	2.3×10^7	—	—	—	4.070KB	32m

Table: Time and memory required for generation

Multi-way Decision Diagram

How to improve the symbolic state-space algorithm

Saturation: an iteration strategy

Since most events in a *globally-asynchronous locally-synchronous model* are highly **localized**, then we can exploit this to improve RS generation

Saturation: an iteration strategy

Let $Top(\alpha)$ be the highest MDD level on which event α depends, respectively **MDD node p at level k is saturated if it encode a fix-point w.r.t. any event α s.t.** $Top(\alpha) \leq k \Rightarrow$ **all MDD nodes reachable from p are also saturated.**

Saturation algorithm

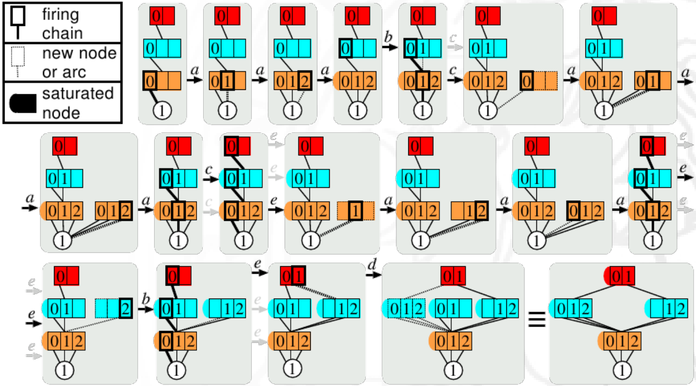
- build the L-level MDD encoding of M_0;
- saturate each node at level 1: fire in them all events α s.t. $Top(\alpha) = 1$;
- saturate each node at level 2: fire in them all events α s.t. $Top(\alpha) = 2$; (if this creates nodes at level 1, saturate them immediately upon creation)
- saturate each node at level 3: fire in them all events α s.t. $Top(\alpha) = 3$; (if this creates nodes at levels 2 or 1, saturate them immediately upon creation)
- ...
- saturate the root node at level L: fire in it all events α s.t. $Top(\alpha) = L$; (if this creates nodes at levels L-1, L-2,. . . , 1, saturate them immediately upon creation)

states are not discovered in breadth-first order

enormous time and memory savings for asynchronous systems

Saturation: an iteration strategy

level	event: a	event: b	event: c	event: d	event: e
3 2 1	$*$ $*$ $0 \to 1, 1 \to 2, 2 \to 0$	$*$ $0 \to 1, 2 \to 1$ $*$	$*$ $0 \to 1$ $1 \to 0$	$1 \to 0$ $*$ $*$	$0 \to 1$ $0 \to 2$ $0 \to 1$



Saturation: an iteration strategy

N	$\lvert\mathcal{S}\rvert$	GreatSPN BBT	GreatSPN File	GreatSPN T.	No Sat. Mem.	No Sat. T.	Sat. Mem.	Sat. T.
Dining philosophers Petri net								
7	2.4×10^4	1,175KB	1,027KB	8s	51KB	8s	36KB	0.13s
8	1.0×10^5	4,977KB	4,976KB	45s	75KB	50s	49KB	0.15s
9	4.3×10^5	21,082KB	23,717KB	345s	103KB	411s	64KB	0.18s
10	1.8×10^6	89,304KB	104,654KB	31m	139KB	28m	80KB	0.24s
11	7.8×10^6	—	—	—	185KB	82m	99KB	0.35s
12	3.3×10^7	—	—	—	235KB	7h	120KB	0.43s
30	6.4×10^{18}	—	—	—	—	—	829KB	10s
40	1.1×10^{25}	—	—	—	—	—	1,576KB	32s
50	2.2×10^{31}	—	—	—	—	—	2,364KB	1m
Flexible manufacturing system Petri net								
4	$1,3 \times 10^5$	6,627KB	3,037KB	11s	363KB	14s	76KB	0.01s
5	6.5×10^5	31,466KB	14,421KB	134s	775KB	63s	135KB	0.01s
6	2.5×10^6	120,940KB	55,430KB	7m	1.470KB	3m	218KB	0.01s
7	8.2×10^6	—	—	—	2.536KB	12m	353KB	0.02s
8	2.3×10^7	—	—	—	4.070KB	32m	515KB	0.13s
9	6.1×10^7	—	—	—	—	—	679KB	0.17s
10	1.4×10^8	—	—	—	—	—	899KB	0.18s
11	3.3×10^8	—	—	—	—	—	1,268KB	0.19s

Table: Time and memory (Kb) required for generation

Conclusion

In this presentation:

- we have introduced Binary Decision Diagram (BDD) and Multiway Decision Diagram (MDD) ;
- we have show how these data structures can be used to efficiently generate and storage the Reachability Set (RS) of PN;
- we have described how RS generation can be improved using the saturation technique.

Future presentation/work:

- how BDD/MDD can be used to perform model checking.

Acknowledgements

Some transparencies are adapted from the notes and transparencies of:

- Prof. Gianfranco Ciardo, Iowa State University.

  Department of Computer Science

  Ames, USA