Représentation symbolique d'espaces d'états

Yann Thierry-Mieg
Jan 2022
ASTRE
M2 SAR – Sorbonne Université

Reduced Ordered Binary Decision Diagrams

| (RO)BDD | → | BDD for Model-checking | → | BDD extensions |

Binary Decision Diagrams

Y. Thierry-Mieg   **3**   M2 –ASTRE – Sorbonne Université

- **Most cited document in computer science according to citeseer [in 2005, in 2017 down to place 61]:**
  1. **Graph-Based Algorithms for Boolean Function Manipulation – Bryant (1986)**
  In this paper we present a new data structure for representing Boolean functions and an associated set of…

- **Introduces Reduced Ordered Binary Decision Diagrams (RO)BDD**

- **What is a (RO)BDD ?**

A compact data structure to represent boolean functions

- An ordered (a<b<c) binary decision diagram for f :
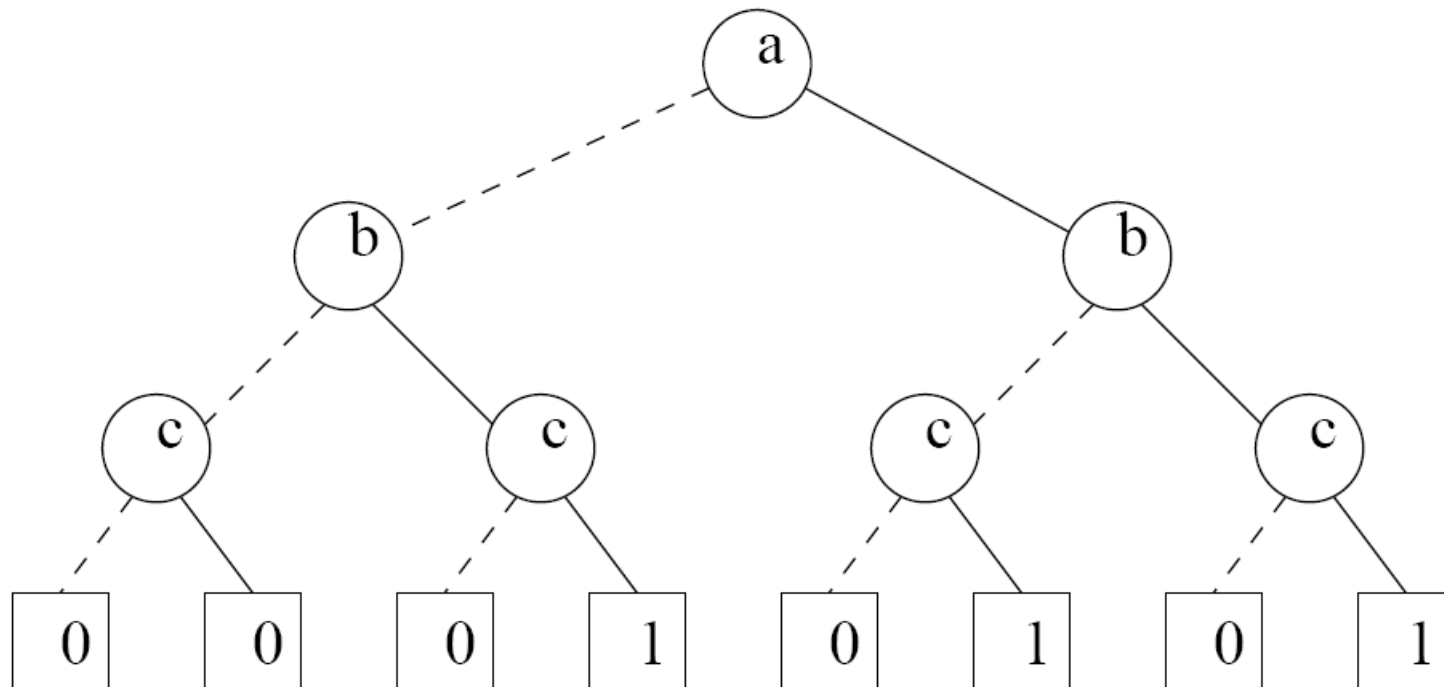
- **Reduction : single occurrence of terminals**
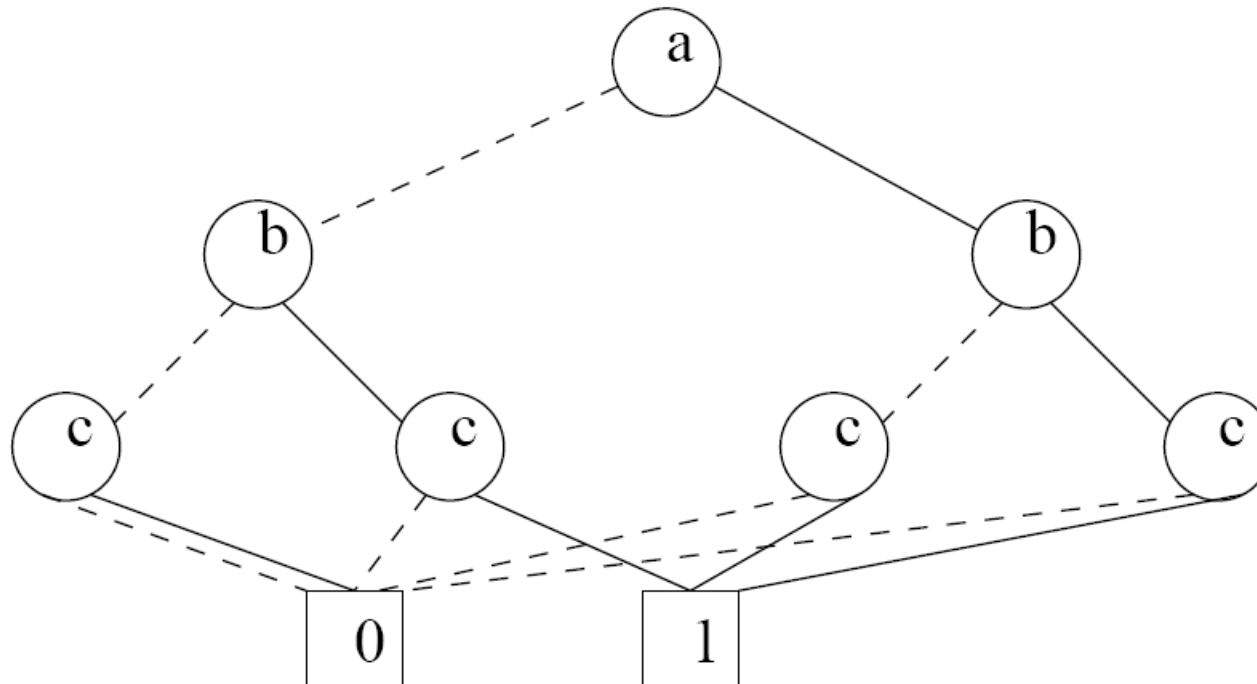
- **Recursively from terminals : single occurrence of any node**

  - **Uses a unicity table for nodes (hash table)**
  - **Node hash key based on : node + hash key of sons**
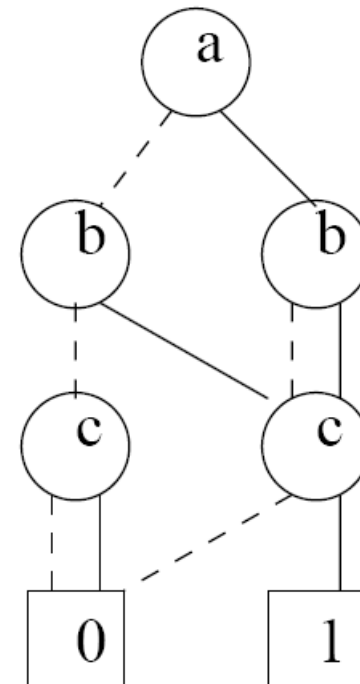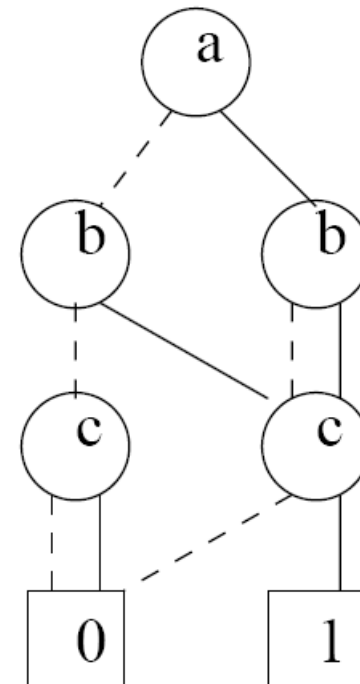
- **Recursively from terminals : single occurrence of any node**

  - **Uses a unicity table for nodes (hash table)**
  - **Node hash key based on : node + hash key of sons**

- **Remove useless nodes**
  - **Criterion : variable value does not influence truth value of formula**
  - **<=> both son arcs point to same node**
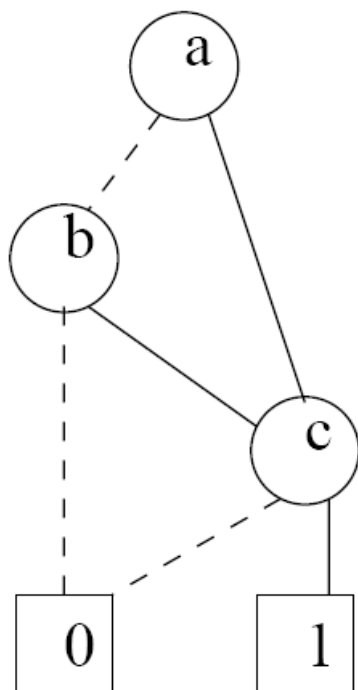
a	b	c	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

- **Variant :**
  - **Remove paths that lead to 0 (false)**
  - **Represents (only) the set of "true" values of f**

a	b	c	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Properties of ROBDD
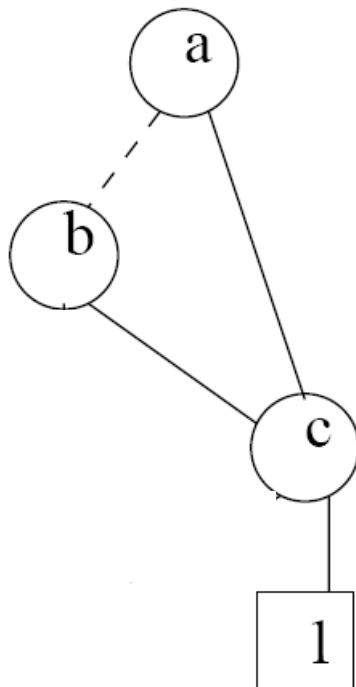
- Given a boolean expression y, or a function $f : \mathbb{B}^K \rightarrow \mathbb{B}$, there is a unique BDD encoding it (given a variable order $\mathbf{x}_K, \ldots, \mathbf{x}_1$)

- Many functions have a very compact encoding as a BDD

- The constant functions 0 and 1 are represented by the nodes $Zero$ and One, respectively

- Test whether a boolean expression is constantly true or false in $O(1)$ time, given its BDD encoding

- Test whether two boolean expressions are equivalent in $O(1)$ time, given their BDD encoding
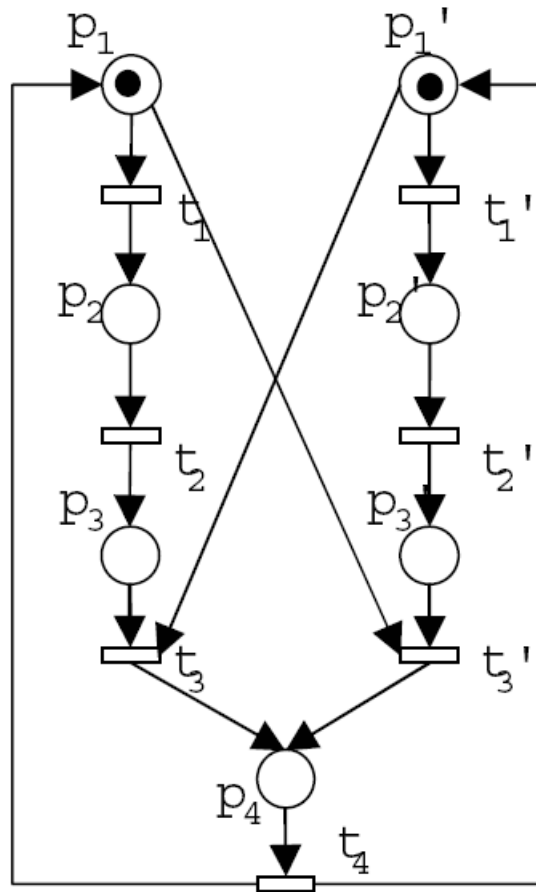
- The variable ordering affects the size of the BDD, consider $x_K \Leftrightarrow y_K \wedge \cdots \wedge x_1 \Leftrightarrow y_1$

  ○ with the order $(x_K, y_K, \ldots, x_1, y_1)$　　　　　　　　　　$O(K)$ nodes

  ○ with the order $(x_K, \ldots, x_1, y_K, \ldots, y_1)$　　　　　　　$O(2^K)$ nodes

- The BDD encoding of some functions is large (exponential) for any order

  ○ the expression for bit 32 of the 64-bit result of the multiplication of two 32-bit integers

- Finding the optimal ordering is an NP-complete problem
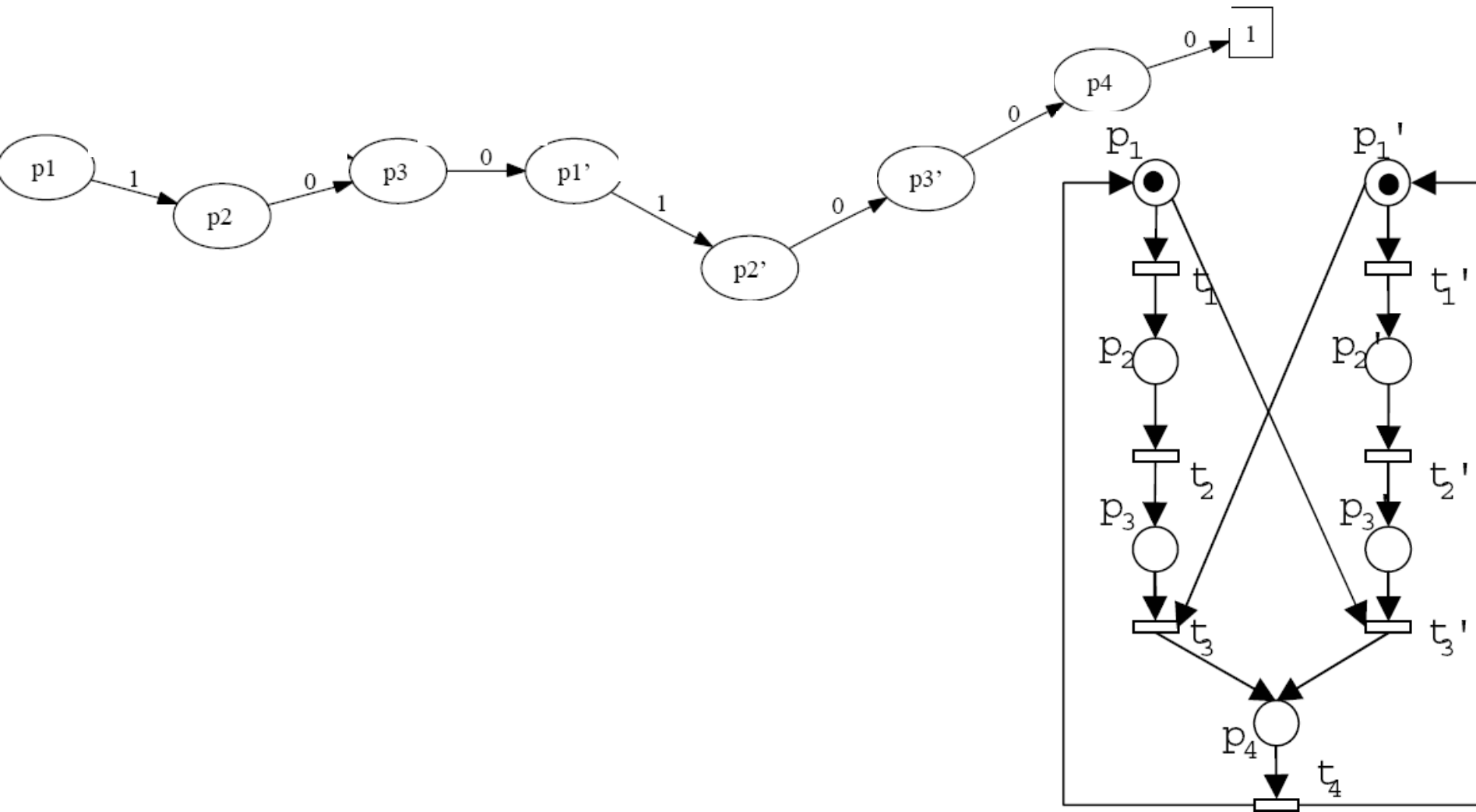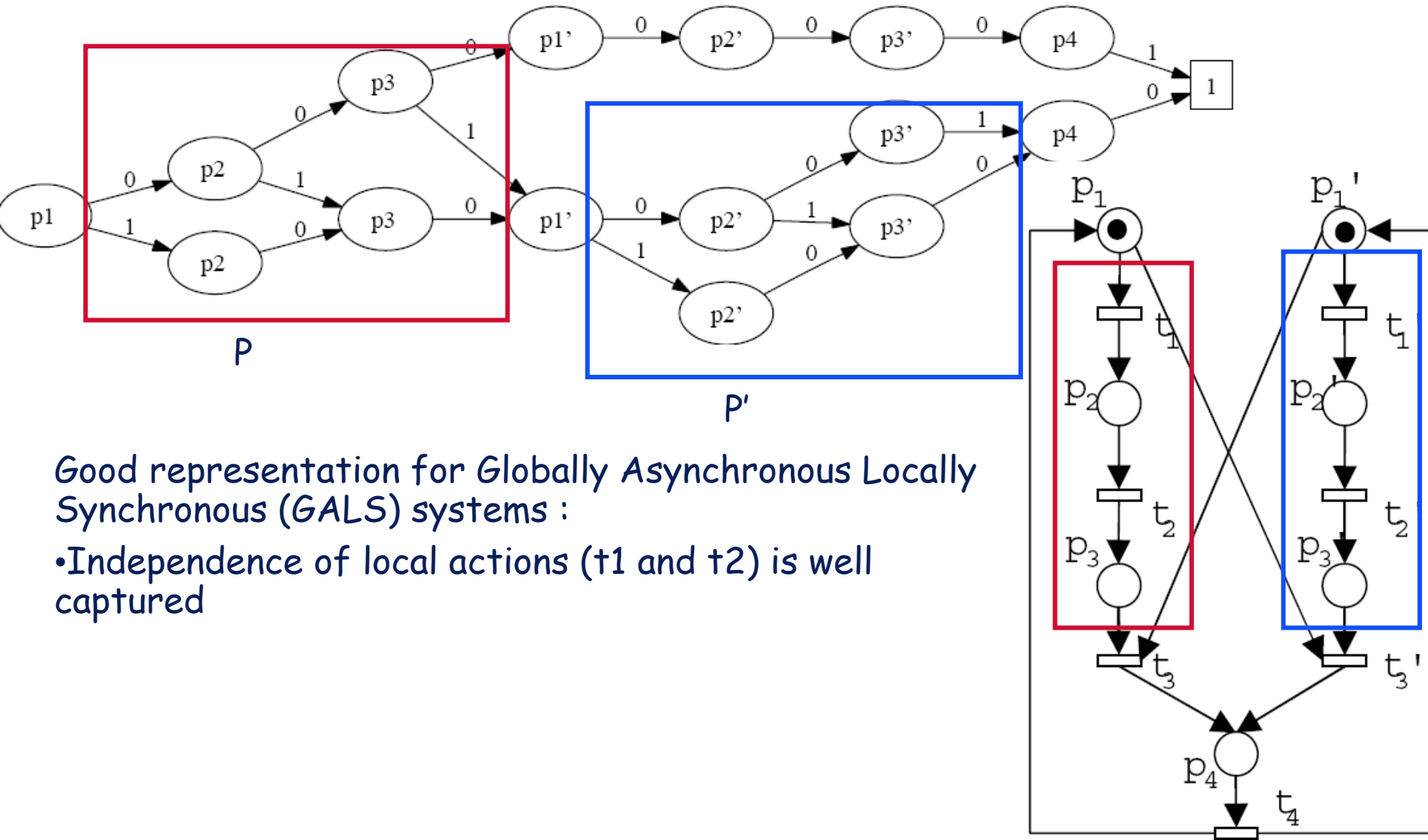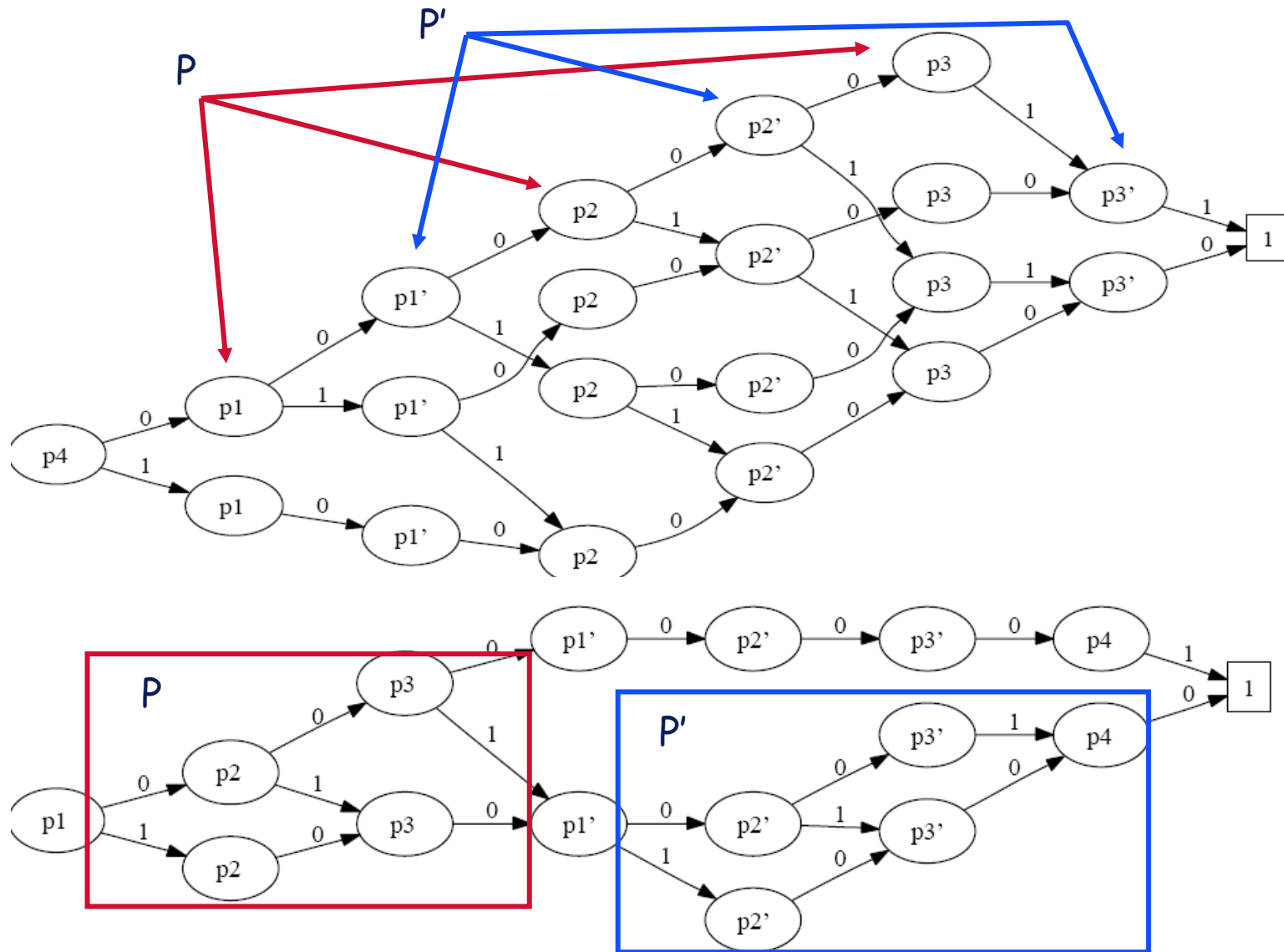
Model checking
and BDD

- Principle :
  - **A path in the structure represents a reachable state**
  - **A state S is described by the value of its state variables**
- Example :

A mutual exclusion protocol
for 2 process p and p'

P

P'

Good representation for Globally Asynchronous Locally Synchronous (GALS) systems :

•Independence of local actions (t1 and t2) is well captured

Growth in node size w.r.t. order

| Nb Proc	Nb States	Consecutive order	Interlaced order
2	10	16	21
5	244	40	260
10	59050	80	13321
15	1.434e+07	120	589838
20	3.486e+09	160	-out of ram-

- Linear growth vs exponential growth for poor ordering clearly visible
- Linear growth of representation but exponential growth of state-space size with appropriate ordering !!

- For some problems BDD based techniques allow to go much further than explicit representation techniques

- **Computations on BDD use a cache to limit complexity**

- **Cache is of the form**
  - Key:<operation, operand_1,..,operand_n >-> value:result
  - Where operands and result are BDD nodes

- **Example :**
  - Cache : contains <union, a, b > -> c if (a ∪ b) = c

- **union (a,b)**
  - If (a=0 or b=1) return b;
  - If (a=1 or b=0) return a;
  - If (a=b) return a;
  - If (<union,a,b> -> r in cache) return r;
  - BDD r= createBDD(
    - *0 => union(a[0],b[0]),*
    - *1=> union(a[1],b[1]))*
  - Cache.add(<union,a,b> -> r)
  - Return r;

- **createBDD uses a unicity table based on node structure**
  - Hash on value of son nodes
- **Operation cache =>**
  - complexity proportional to #nodes(a) x #nodes(b) => number of nodes
  - Without it complexity would be proportional to number of PATHS in the structure
- **Intersection differs from union only in terminal cases**
  - If (a = 1 or b = 0) return b
  - If (a = 0 or b = 1) return a

- **Classic algorithm is based on Breadth-first exploration BFS**
  - **Consider a system composed of k state variables (i.e. state space represented as a k-level BDD)**
  - **Transition relation represented using 2k variables**

    (i,j) = (i1,j1, ..ik,jk)

    System can go from i to j in one step
  - **A special synchronized product (relational product) operation is defined to apply such a transition to a system**

Transition BDD for Next(t1)

Paths to 0 terminal not represented

Useless variables removed
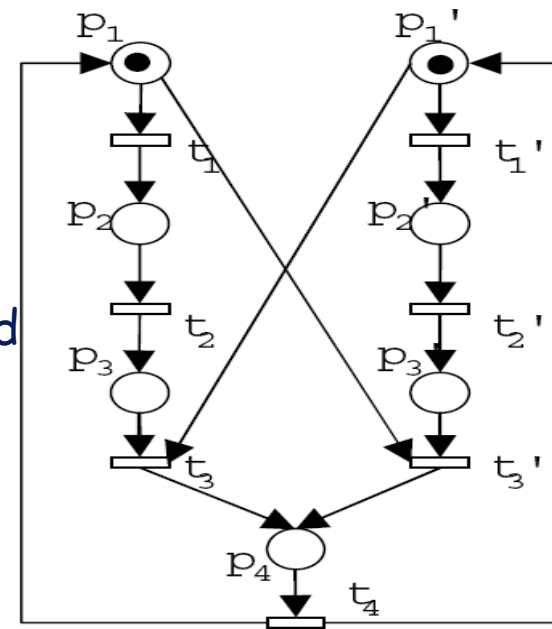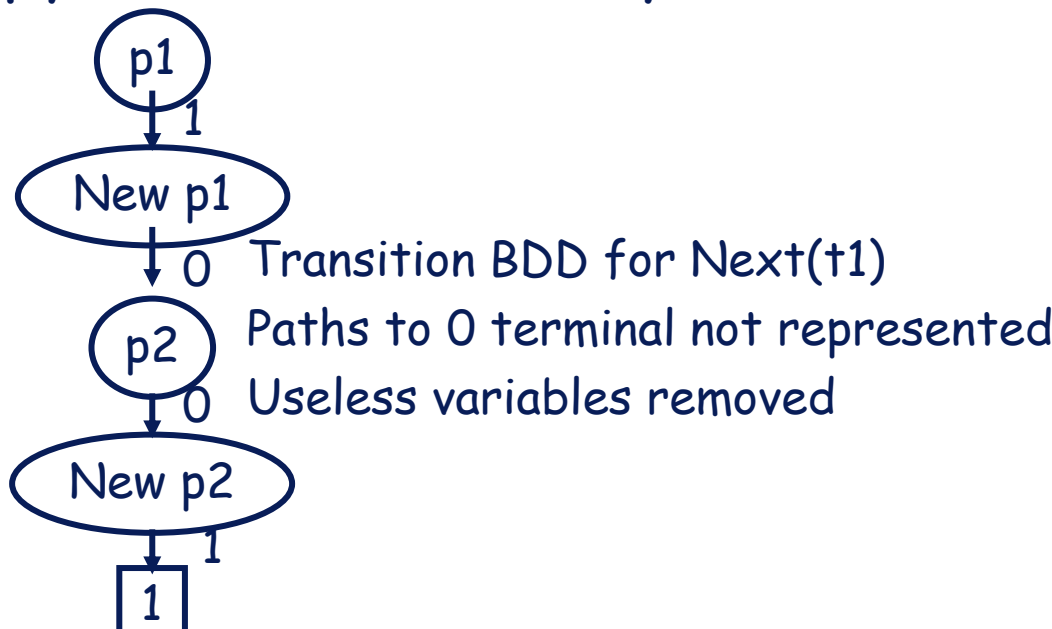
Given the *initial state* $\mathbf{s}$ and the *next-state function* $\mathcal{N}$, we obtain the *state space* $\mathcal{S}$:

$$\mathcal{S} = \mathcal{N}^*(\mathbf{s})$$

$\mathbf{s}$ $\mathcal{N}(\mathbf{s})$ $\mathcal{N}(\mathcal{N}(\mathbf{s}))$ $\cdots$ $\mathcal{N}^d(\mathbf{s})$
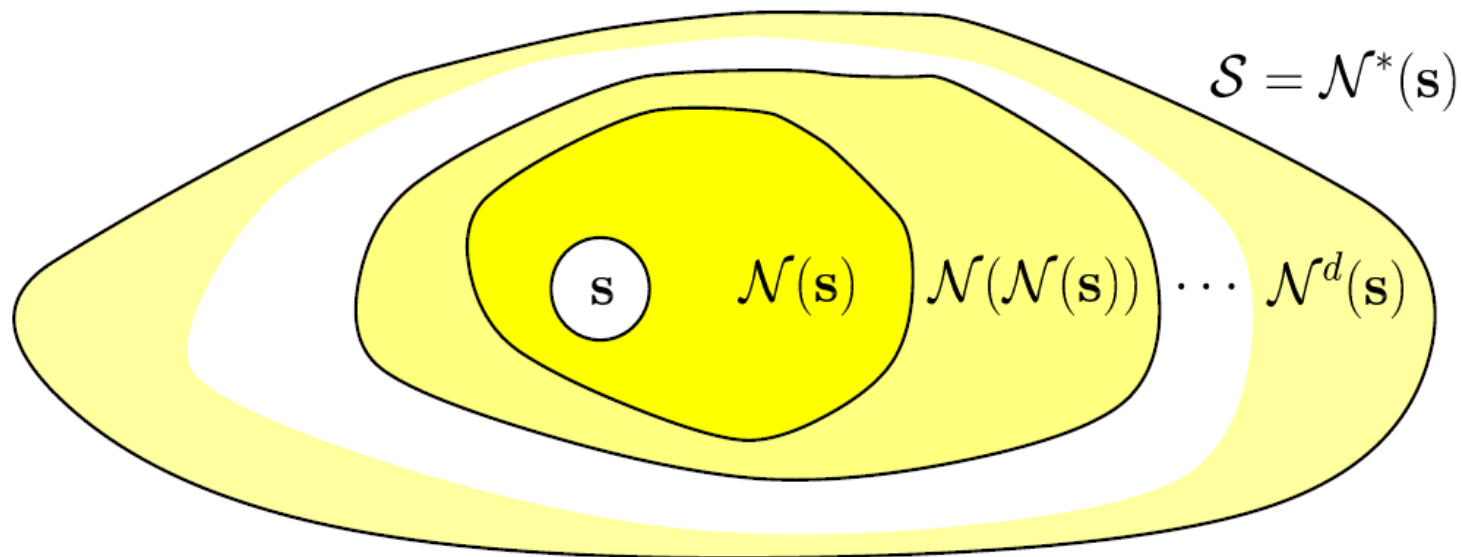
The *number of iterations* equals one plus the maximum distance d of any state from $\mathbf{s}$

The *peak* BDD size is usually achieved well before reaching the *final* BDD size at step d

- **BDD representing transitions can be combined using union**
  - **Full transition relation**
    - *NextAll = union (Next(t1)+..+Next(tn))*

- **Algorithm for BFS(s0)**
  - **S := S0**
  - **N := 0**
  - **While (N != S)**
    - N := S
    - S := S U NextAll(S)
  - **Return S**

"Symbolic Model Checking: 10 20 States and Beyond" (LICS'1990) Burch, Clarke, McMillan, Dill, Hwang

- **BFS performs better than the "intuitive algorithm"**
  - **Size of representation is not directly linked to number of states manipulated**
  - **Re-evaluating a transition on an already reached state is likely to cause a cache-hit thus has experimentally low cost.**

- **Algorithm for newBFS(s0)**
  - S := S0
  - N := S0   Only computes NextAll on newly reached states
  - While (N != 0)
    - N := NextAll(N) \ S
    - S := S U N
  - Return S

- **The idea is to cluster some transitions but keep an expression of the transition relation in parts**
  - **Next(i) for I in 1..nbClusters = union (Next(tj),..Next(tj+k))**
  - **Create one cluster for each process (requires structural information)**
- **Not quite BFS anymore as chainings may occur**
- **Solves problems experienced with the size of NextAll BDD**
- **Algorithm for chainBFS(s0)**
  - **S := S0**
  - **N := 0**
  - **While (N != S)**
    - N := S
    - For (i in 1..nbClusters)
      - S := S ∪ Next(i) (S)
  - **Return S**

Comparing the four approaches

Performances using Smart

| N	$\mathcal{S}$	New BFS	BFS	New Chain	Chain	New BFS	BFS	New Chain	Chain	final
			Time (sec)				**Memory (MB)**			
Dining Philosophers: $K=N$, $|\mathcal{S}_k|=34$ **for all** k										
50	2.2×10^{31}	37.6	36.8	1.3	1.3	146.8	131.6	2.2	2.2	0.0
100	5.0×10^{62}	644.1	630.4	5.4	5.3	>999.9	>999.9	8.9	8.9	0.0
1000	9.2×10^{626}	—	—	895.4	915.5	—	—	895.2	895.0	0.3
Slotted Ring Network: $K=N$, $|\mathcal{S}_k|=15$ **for all** k										
5	5.3×10^4	0.2	0.3	0.1	0.1	0.8	1.1	0.3	0.2	0.0
10	8.3×10^9	21.5	24.1	2.1	1.2	39.0	45.0	5.7	3.3	0.0
15	1.5×10^{15}	745.4	771.5	18.5	8.9	344.3	375.4	35.1	20.2	0.0
Round Robin Mutual Exclusion: $K=N+1$, $|\mathcal{S}_k|=10$ **for all** k **except** $|\mathcal{S}_1|=N+1$										
10	2.3×10^4	0.2	0.3	0.1	0.1	0.6	1.2	0.1	0.1	0.0
20	4.7×10^7	2.7	4.4	0.3	0.3	5.9	12.8	0.5	0.5	0.0
50	1.3×10^{17}	263.2	427.6	2.9	2.8	126.7	257.7	4.3	3.8	0.1
FMS: $K=19$, $|\mathcal{S}_k|=N+1$ **for all** k **except** $|\mathcal{S}_{17}|=4$, $|\mathcal{S}_{12}|=3$, $|\mathcal{S}_7|=2$										
5	2.9×10^6	0.7	0.7	0.1	0.1	2.6	2.2	0.4	0.2	0.0
10	2.5×10^9	7.0	5.8	0.5	0.3	18.2	14.7	2.3	1.3	0.0
25	8.5×10^{13}	677.2	437.9	12.9	5.1	319.7	245.3	42.7	21.2	0.1

Explicit generation of the state space $\mathcal{S}$ adds *one state* at a time

- memory O(states), increases linearly, peaks at the end
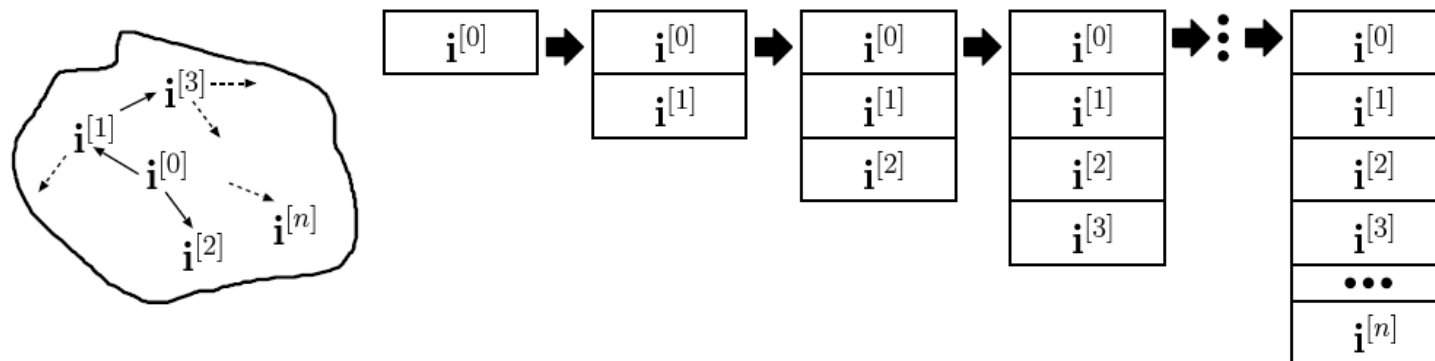
Symbolic generation of the state space $\mathcal{S}$ with decision diagrams adds *sets of states* instead

- memory O(decision diagram nodes), grows and shrinks, usually peaks before the end
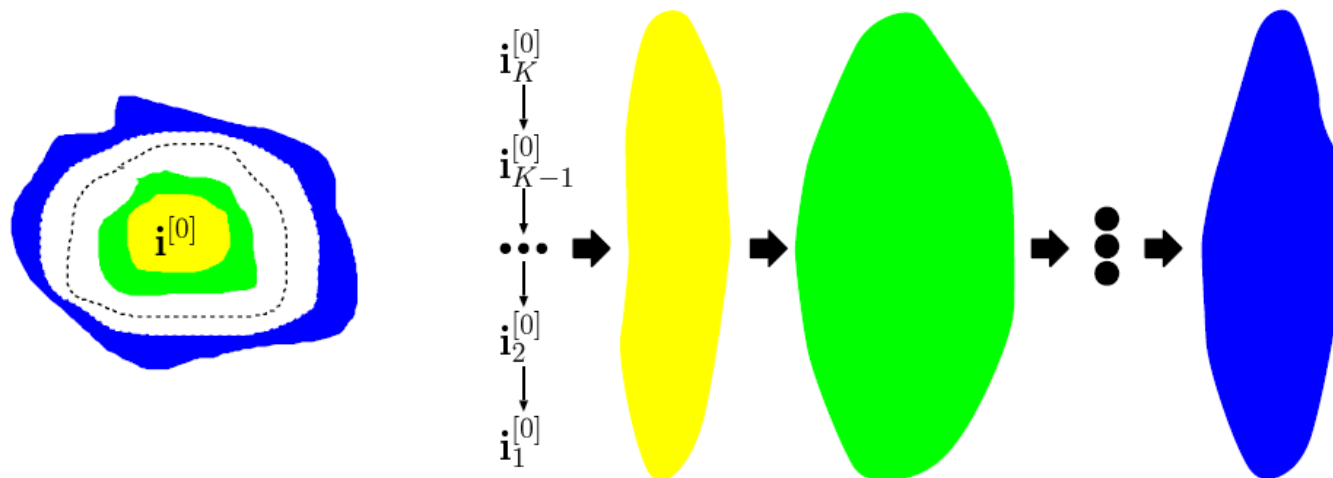
- **The state space representation allows to easily verify safety properties**
  - Can we reach a "bad" state
  - Can A and B both be true simultaneously
- **State space generation is the basis for more complex temporal logic properties such as CTL**
  - CTL properties can be expressed as nested fix points of the transition relation and its reverse $Next^{-1}$

Some BDD extensions

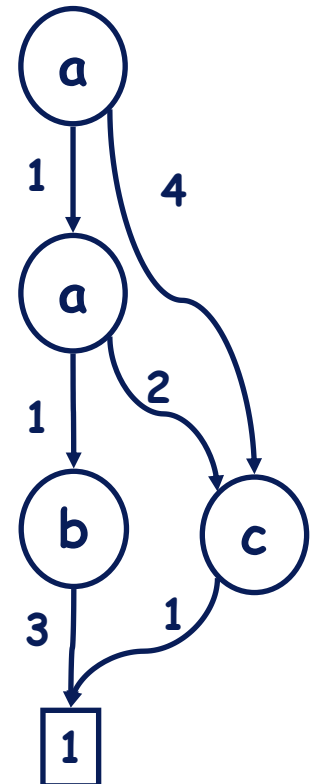(RO)BDD → BDD for Model-checking → BDD extensions

Decision Diagrams : Widely accepted in verification tools

- ## SMV (US):
  - FSM/Kripke structure
  - emblematic first symbolic enabled verification tool. Now uses (NuSMV 2-Italy) library Cudd.
- ## Uppaal (Den-Nor):
  - Hybrid systems
  - uses Difference Bounded Matrix diagrams to represent clocks
- ## Prism (UK) :
  - Stochastic process algebra
  - uses Matrix DD and Multi-terminal DD for stochastic verification
- ## Smart (US) :
  - Stochastic Petri nets
  - uses integer valued DD, both CTL and stochastic solution engine (+saturation)
- ## Red (Taiwan) :
  - Timed automata
  - Specific solution for real time systems

Integer valued Decision Diagrams

- **Some variants:**
  - **Multi-way DD (Ciardo&Miner Icatpn'99)**
  - **Data Decision Diagrams (Couvreur et al. Icatpn'02)**
- **Variables may have an integer domain instead of boolean domain**
  - **Usually zero-suppressed, to allow arbitrary variable domains provided the actual reachable set is finite**
  - **Using BDD, one has to decode integer state variables into their log_2(n) bit representation**
  - **Problem : complexity also linked to nb arcs/node, not only number of nodes**
- **Data Decision Diagram**
  - **No variable order (handled in the union operation to handle incompatibilities)**
  - **Homomorphisms to define the transition relation**

$$a \xrightarrow{4} c \xrightarrow{1} 1$$

$$a \xrightarrow{1} a \xrightarrow{2} c \xrightarrow{1} 1$$

$$a \xrightarrow{1} a \xrightarrow{1} b \xrightarrow{3} 1$$

Multi terminal Decision Diagrams

- Multi-terminal (MTDD [Fujita+ '97] or Algebraic DD [Bahar+ '93]):
- Instead of single terminal 1, use several terminals
- Allows to give a correspondence between a state and a characteristic it has
  - Not just presence or absence of a state
- Example 1: integer terminal
  - Terminal gives the distance (number of steps) from initial state of any state
  - union handles same path with different terminals => keep the smallest terminal
  - Useful for finding shortest witness or counter-example traces
- Example 2: Real valued terminal
  - Used in stochastic/probabilistic systems, gives the probability of being in a state
  - union handles the approximation (2 terminals x and y considered equal if |x-y| < epsilon)

CTL Model checking

- **Computation Tree Logic**

  - **(infinite) tree of all possible executions of a system**

  - **Superset of Boolean first order logic**
    - *AND, OR, NOT, TRUE, FALSE*

  - **Additional temporal operators**
    - *G : Generally (p holds in all states)*
    - *X p: neXt (p holds in a successor state by one step)*
    - *F p: Future (p holds in a state reachable in arbitrary number of steps)*
    - *p U q: Until (p holds until q holds)*

  - **Modalities : Exists, Always**
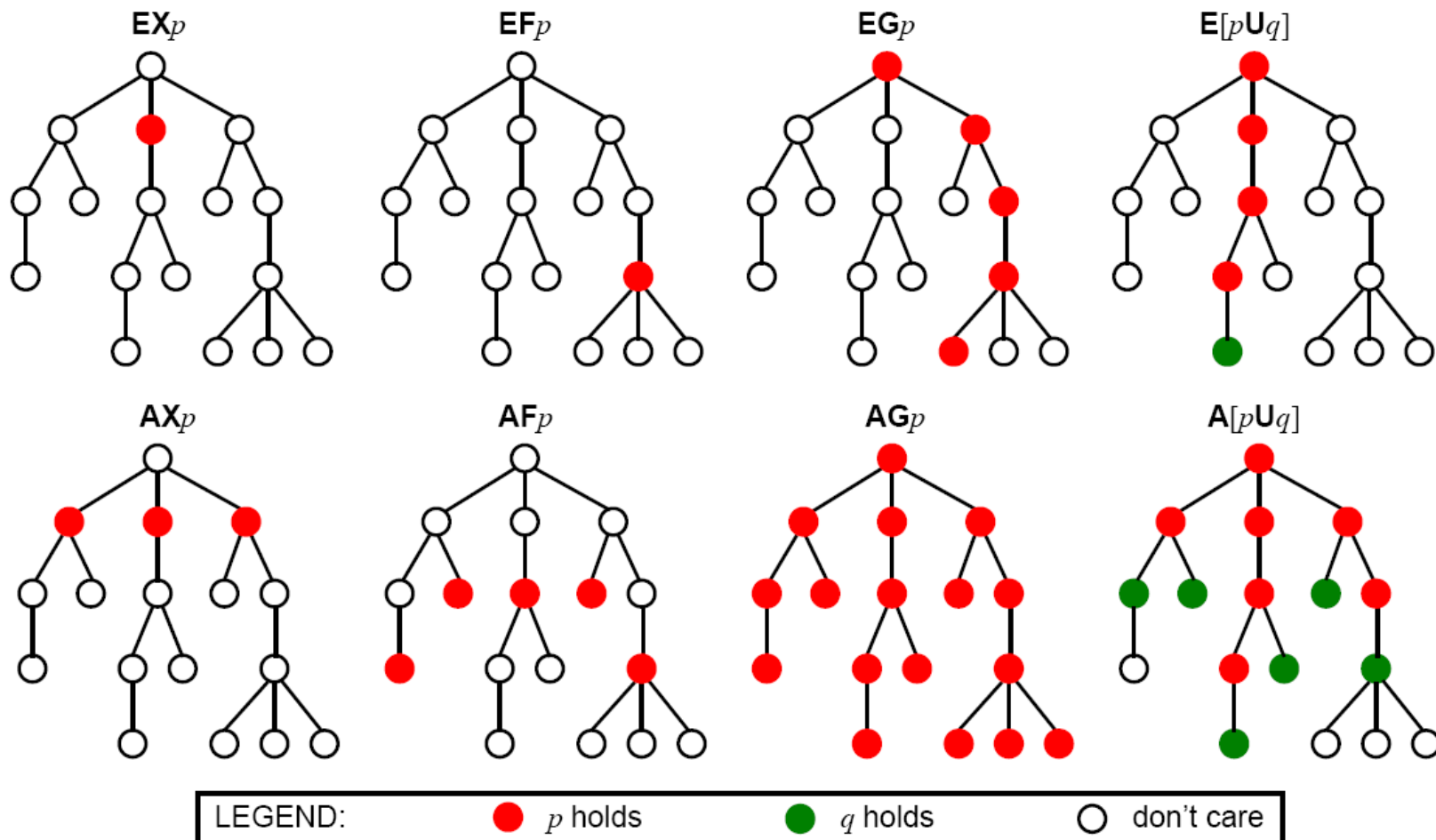
EX, EU, and EG are a complete set of CTL operators, since:

$$AXp = \neg EX \neg p \qquad EFp = E[true \cup p] \qquad E[pRq] = \neg A[\neg p \cup \neg q]$$
$$AFp = \neg EG \neg p \qquad A[p \cup q] = \neg E[\neg q \cup \neg p \wedge \neg q] \wedge \neg EG \neg q \qquad A[pRq] = \neg E[\neg p \cup \neg q]$$
$$AGp = \neg EF \neg p$$

- We have

  - **A compact representation for sets of states**

  - **A representation of transition relations, as a set of pairs of states**

- A model provides (So,Next,Lab)

  - **An initial state So**

  - **A successor relation Next, that gives the set of successors reachable in one step from a set of states**

  - **A labeling function, that tags individual states with the truth value of atomic propositions (e.g. x<3)**

- **Start from syntactic tree of the formula**
- **Compute all states S(p) satisfying a formula p recursively**
- **Terminal cases :**
  - **p=True : all reachable states**
  - **p=False : empty set of states**
  - **p=Atomic predicate (e.g. x<3): all reachable states satisfying p**
- **Boolean cases :**
  - **p AND q : intersect states S(p) and S(q)**
  - **p OR q : union states S(p) and S(q)**
  - **NOT p : all reachable states set minus S(p)**
- **Temporal operators :**
  - **Reduce to EX, EU, EG using equivalences**
  - **Algorithms on next slides**

- **Given states satisfying p, S(p)**

  - ## Compute predecessors of S(p)
    - *If a state is a predecessor of a state satisfying p, it satisfies neXt p*

  - ## Requires Pred, i.e. the invert of Next
    - *Easy in classic model, just revert pairs of states in the representation*
    - *Can be more difficult if computed dynamically : risk of unreachable states*
    - *Still possible in general by intersection with forward reachable set*

- **Let F be the set of states satisfying "f"**
  - F can be built by selecting states from the full state space
- **EX(F)**
  - S := Next^{-1} (F)
  - Return S

- **Let F be the set of states satisfying "f"**
- **EG(F)**
  - S := F
  - N := 0
  - **While (N != S)**
    - N := S
    - S := S ∩ Next^{-1} (S)
  - **Return S**

Initialize with states that verify f
Potentially all these states verify Gf

Remove some potential candidates state
If s verifies Gf, s verifies f
and successor verifies "f"

- **Let F be the set of states satisfying "f"**
- **EG(F)**

  Initialize with states that verify f
  Potentially all these states verify Gf
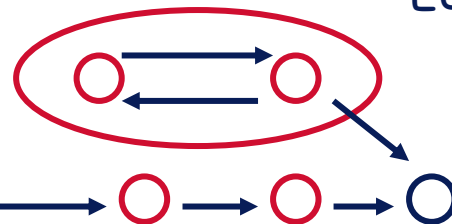
  - S := F
  - N := 0
  - **While (N != S)**

    Remove some potential candidates state
    If s verifies Gf, s verifies f
    and successor verifies "f"
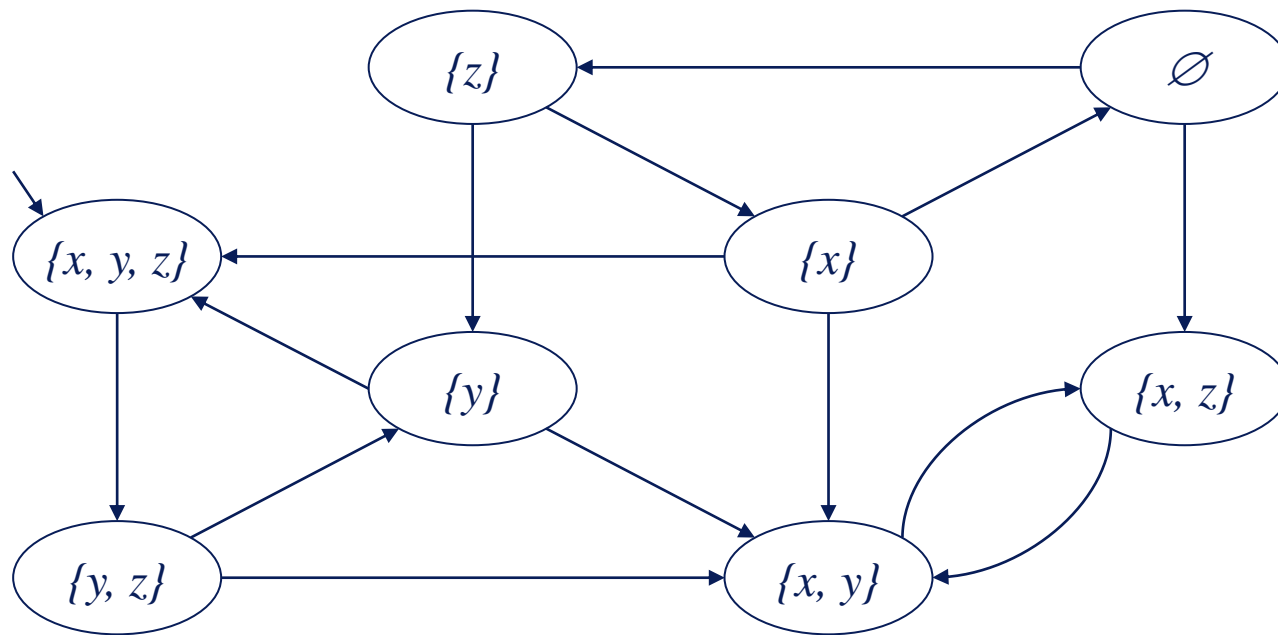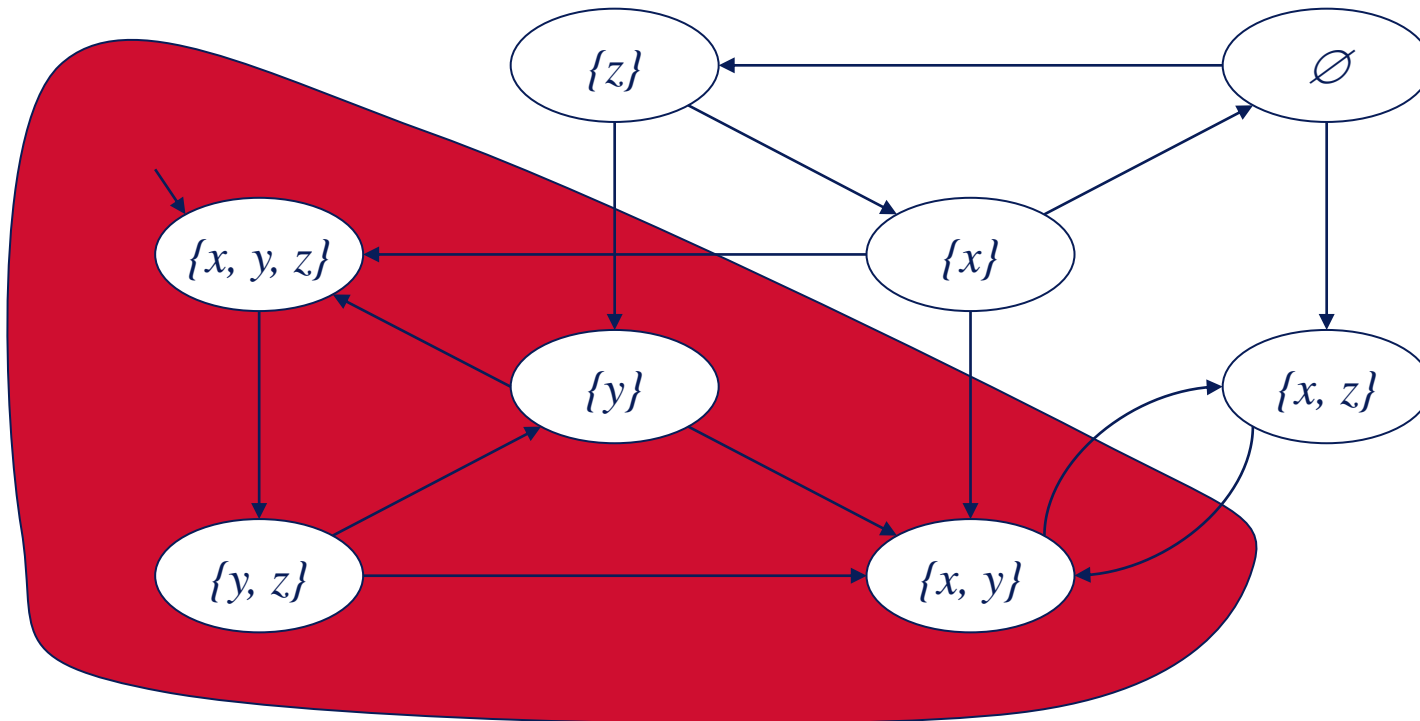
    - N := S
    - S := S ∩ Next^{-1} (S)
- **Return S**
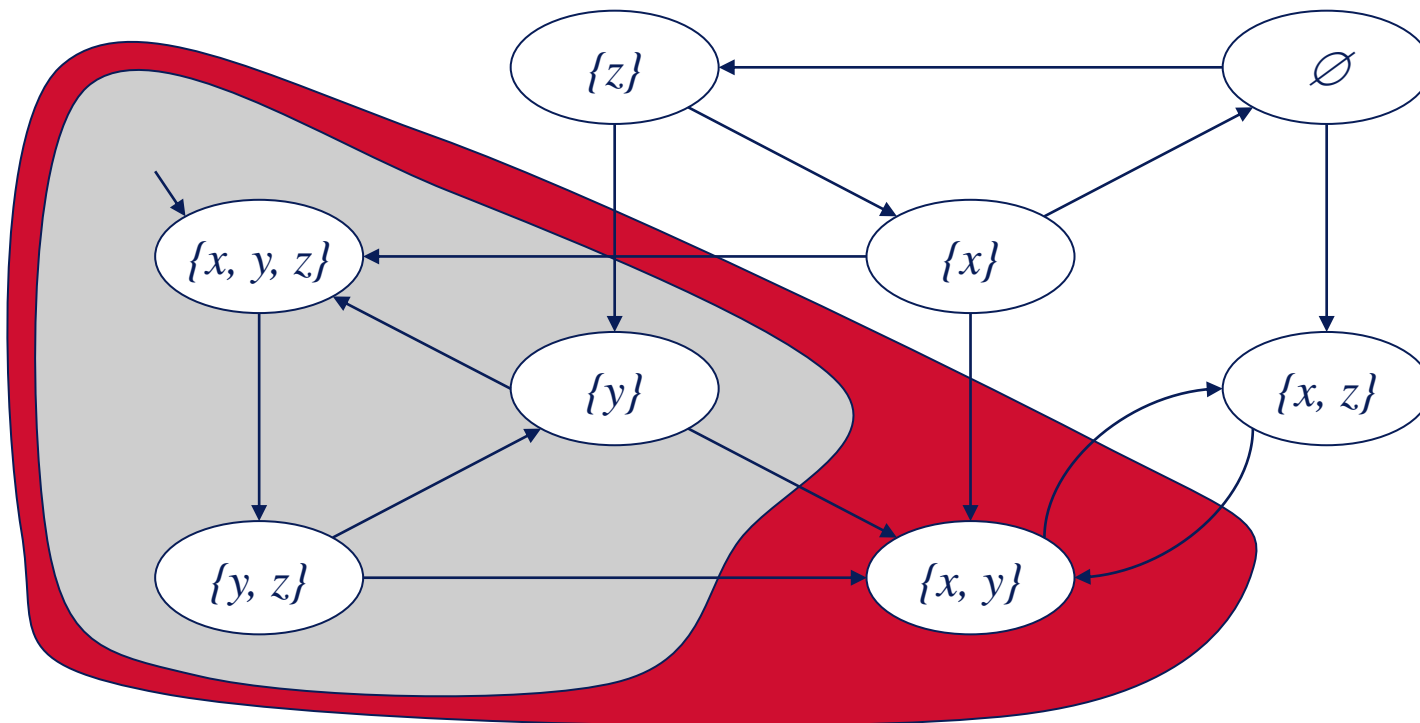
- **Let F be the set of states satisfying "f"**
- **EG(F)**

Initialize with states that verify f
Potentially all these states verify Gf

  - **S := F**

  - **N := 0**

Remove some potential candidates state
If s verifies Gf, s verifies f
and successor verifies "f"

  - **While (N != S)**

    - $N := S$

    - $S := S \cap Next^{-1}(S)$

  - **Return S**

$$(Next^{-1} \cap Id)^* \circ F$$

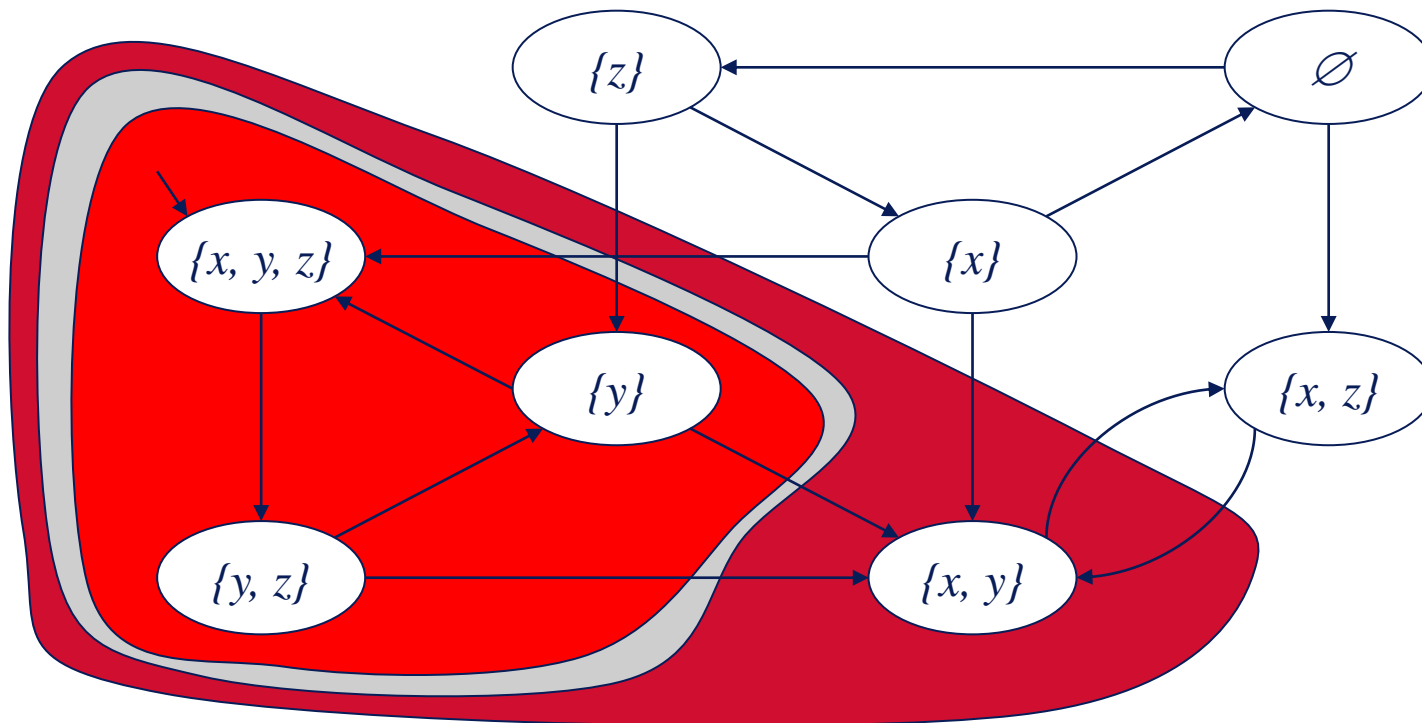

EGf

Illustration for *EG y*

Illustration for *EG y*

States not satisfying *y*
have been excluded

$S := S \cap Next^{-1}(S)$

States having all its successors outside π^l have been excluded

Illustration for *EG y*

The fixed point has been reached

- **Let F and G be the set of states satisfying "f" and "g"**
- **EU(F,G)**

  Initialize with states that verify g

  - S := G
  - N := 0

  Keep only predecessors that verify f

  - **While (N != S)**
    - N := S
    - S := S ∪ (F o Next^{-1} (S))
  - **Return S**

$$(F \text{ o } Next^{-1} + Id)^* \text{ o } G$$

Illustration for *z EU y*
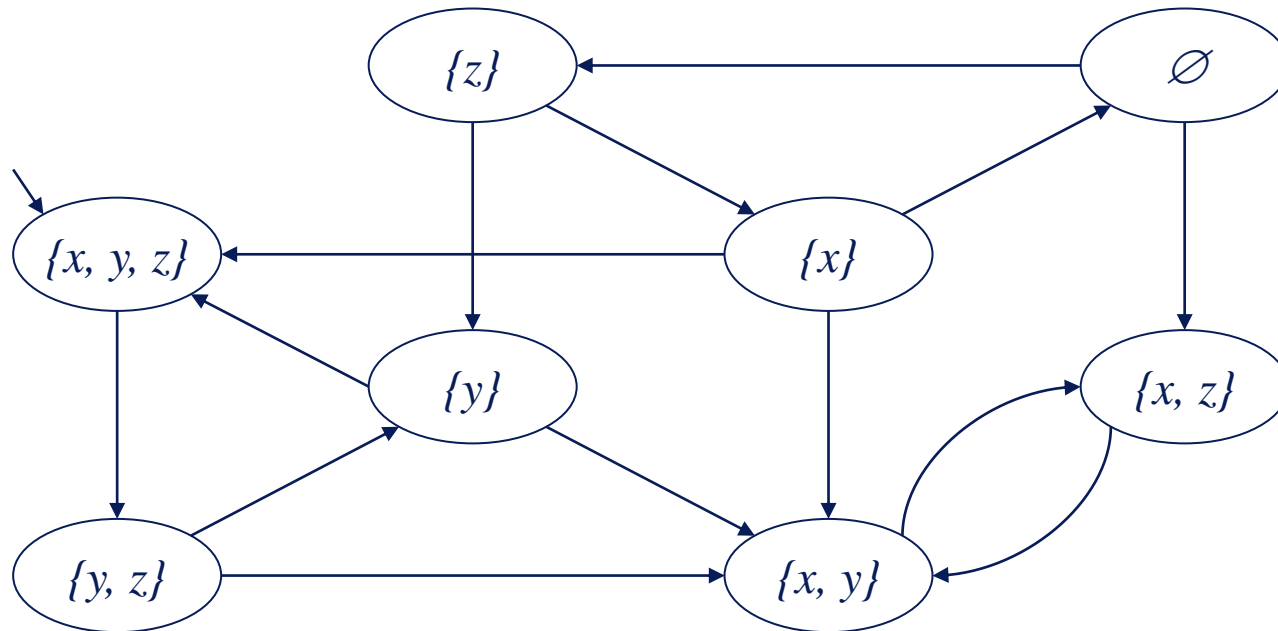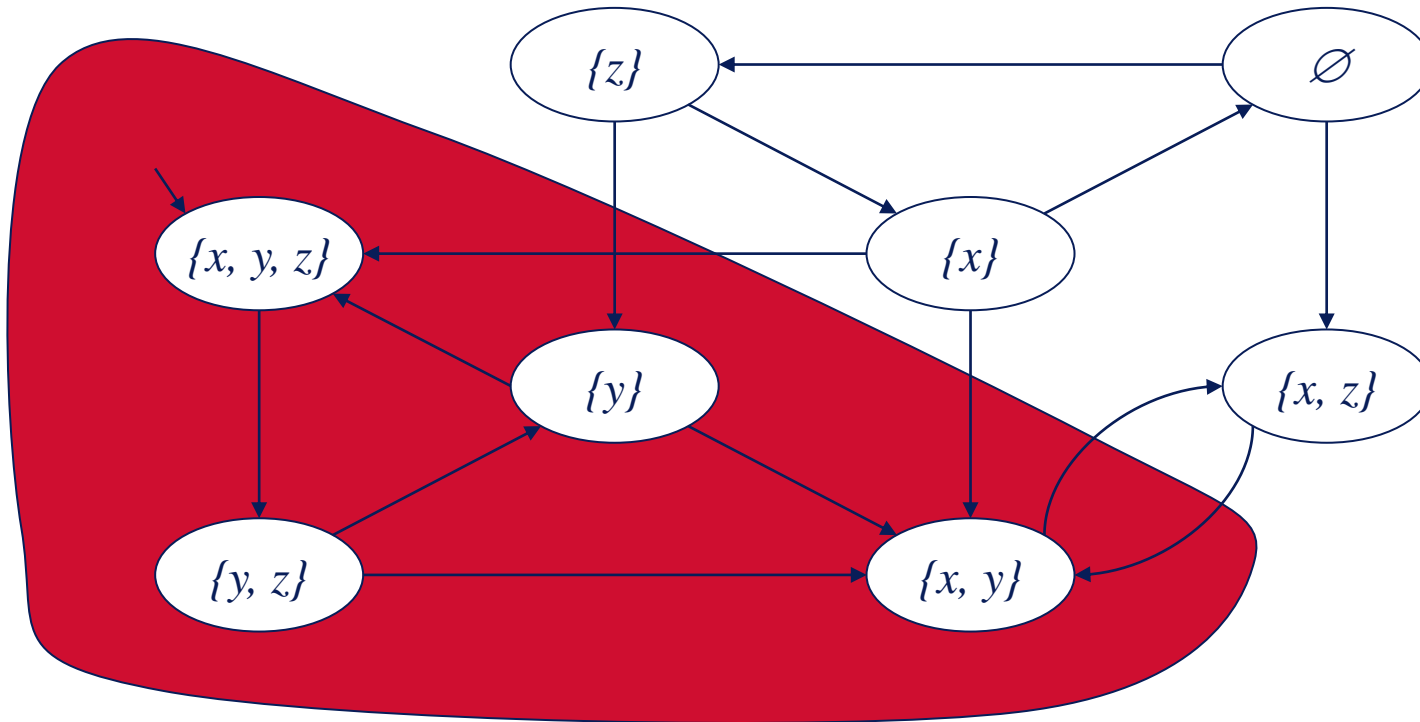
States satisfying *y*
have been added

Illustration for *z EU y*
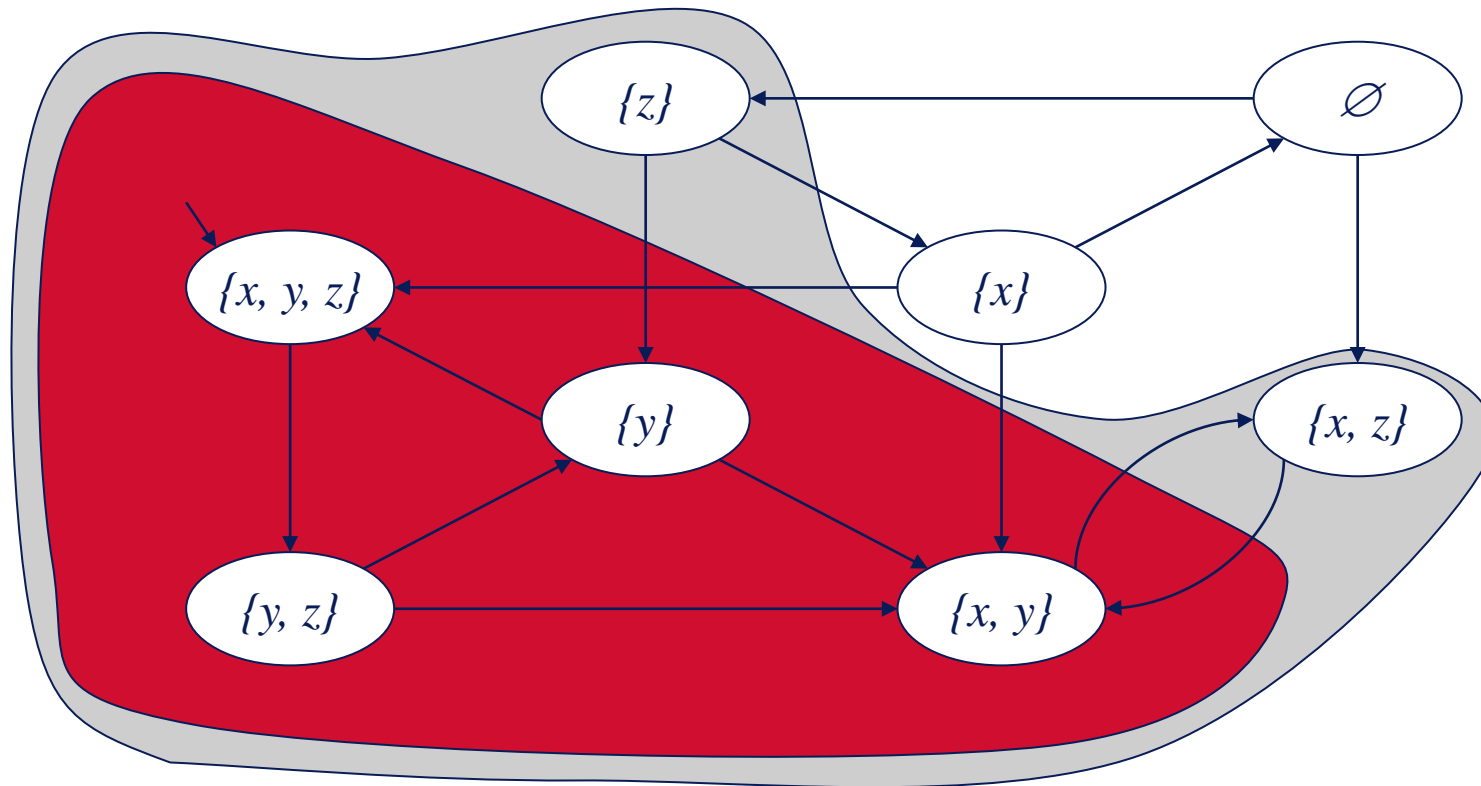
$$S := S \cup (F \circ Next^{-1} (S))$$
Add Predecessors satisfying "z"

States satisfying z and having at least a successor in ξ^1 have been added

Illustration for *z EU y*
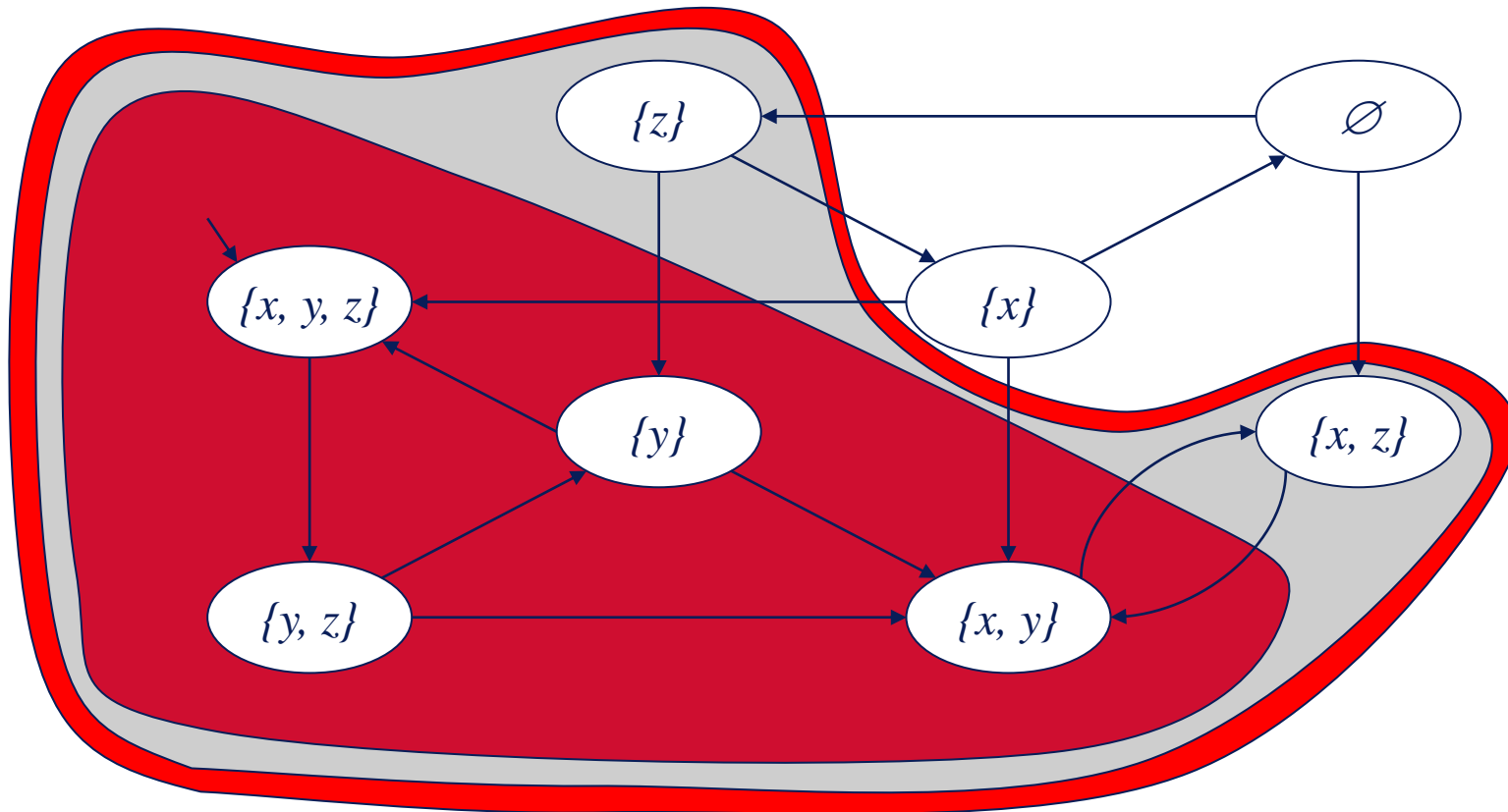
$S := S \cup (F \circ \text{Next}^{-1} (S))$
Add Predecessors satisfying "z"

The fixed point has been reached

- CTL (Branching time) can specify safety properties and some liveness properties
- CTL can be efficiently implemented (linear complexity w.r.t. to the Kripke structure), provided a good management of sets of states.
- Fairness needs to augment the capability of CTL model checkers (SCC searches are needed).
- CTL fair model checkers can be used to verify CTL and also LTL formula.
- CTL does not provide a single counter example when the property does not hold. The output is the set of states that satisfy the formula (maybe huge).
  - Witness paths or counter examples can still be exhibited
- CTL model checkers cannot answer before labeling the initial state with the truth value of the formula.

A Guarded Action Language to express system semantics

Yann Thierry-Mieg
Joint work with
S. Baarir, B. Berard, M.Colange, F. Kordon, D. Poitrenaud
Nov. 2013 – ENS Cachan
Journée AFSEC

- In MDD approaches

  - **Build a Domain Specific Language**
  - **Use model transformation for specific targets**

- Choosing a target formalism

  - **Expressive enough to capture your semantics**
  - **Efficient solution engine**

- We propose ITS/GAL formalism

  - **Allows to express discrete state semantics**
  - **Symbolic model-checking**

- **GAL : a « DSL to express Semantics »**
  - **Simple to use, easy C-like syntax**
  - **Straightforward Petri net style concurrent semantics**
  - **Integer variables and arrays + arbitrarily nested array expressions**
  - **Efficient symbolic solution engine**
  - **Subsumed by Instantiable Transition Systems (ITS), allowing hierarchical composition of GAL modules**
- **Meant to be a back-end target in a transformation process.**
- **Define your semantics in GAL.**

```
GAL system {
    // Variable declarations
    int variable = 5 ;
    array [2] tab = (1, 2) ;

    transition t1 [variable > 9] {
        tab [0] = tab [1] * tab [0] ;
        variable = variable * 5 ;
    }

    transition t2 [variable == 23] label "a" {
        tab [1] = 0 ;
    }
}
```

- All variables are 32 bit integers or arrays of fixed size.
- Any variable must be initialized
  - int a = 0;
  - array [3] tab = (0,0,0);

- Terminal expressions are signed constants, parameters, variables, array access with arbitrary index expression

  - **3,-2,$MAX,x,tab[x+1],tab[tab[$MAX-x]]**

- All C operators supported

  - **Bitwise : &, |, ^, <<, >>, ~**

  - **Integer : +, -, *, /, %, ****

- Boolean expressions

  - **Basics : true, false, &&, ||, !**

  - **Comparisons of integers: ==,!=, <,<=, >,>=**

- x = (y == 255) * 100;   // x is 0 or 100

- **<lhs=rhs>** assign integer expression rhs to variable designated by lhs.
- **<s1;..;sn>** sequence of statements, **<nop>** the empty sequence
- **<ite(c,t,f)>** an if-then-else statement
- **<for(min, max, b)>** a limited form of iteration
- **<abort>** return the empty set (!)
- **<call(a)>** call a label (i.e. an arbitrary transition with label a) of « self »
- **<fixpoint(b)>** fixpoint statement

GAL Transitions

- Tuple : **<label, guard, body>**
- Fire : In any state where guard is enabled, process body statement(s) atomically
- Tau (empty) label for local transitions
- Labeled transitions are not fireable by Locals outside of call or synchronization
- Guard is a boolean expression
- Body is a statement

• **Allow easier configuration of a model**

```
GAL paramSystem ($N = 2, $K = 1) {
    int variable = $N ;
    array [2] tab = ($N + $K, $N - 1) ;
    transition t1 [variable > $N] {
        tab [$K] = tab [1] * tab [0] ;
        variable = variable * 5 ;
    }
    transition t2 [variable == $N] label "a" {
        tab [1] = 0 ;
    }
}
```

```
GAL paramDef ($N=2) {
    typedef paramType = 0..$N;
    typedef paramType2 = 0..1;
    int variable = 0;

    // a transition compactly modeling ($N+1)*2
    // basic transitions
    transition trans (paramType $p1, paramType2 $p2)
            [$p1 != $p2] {
        variable = $p1 + $p2;
    }
}
```

```
GAL iteExample {
    int variable = 0 ;


    transition invert [variable == 0 || variable == 1] {
        if (variable == 0) {
            variable = 1;
        } else {
            variable = 0;
        }
    }
}
```

Equivalent to xor : variable = variable ^ 1

For loop

- **Limited iteration**

```
GAL forLoop  {
    typedef Dom = 0..2;

    array [3] tab  = (0,0,0);

    transition forExample [true] {
        for ($i : Dom) {
            tab[$i] = $i;
        }
    }
}
```

```
GAL forLoop_inst {
    array [3] tab = (0, 0, 0) ;

    transition forExample [true] {
        tab [0] = 0 ;
        tab [1] = 1 ;
        tab [2] = 2 ;
    }
}
```

```
GAL callExample {
    int variable = 0 ;

    transition NDassignX [variable == 0 || variable == 1] {
        self."setX" ;
    }

    transition callee1 [true] label "setX" {
        variable = 1 ;
    }

    transition callee2 [true] label "setX" {
        variable = 0 ;
    }
}
```

```
GAL abortExample ($EFT = 1, $LFT = 3) {
    int a = 1 ;
    int b = 0 ;
    int t.clock = 0 ;

    transition t [a >= 1 && t.clock >= $EFT] {
        a = a - 1 ;
        b = b + 1 ;
        t.clock = 0 ;
    }
```
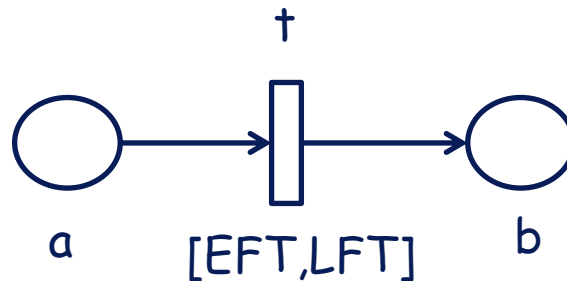
```
transition elapse [true] label "elapse" {
    // is t enabled ?
    if (a >= 1) {
        // is t's clock strictly less than
        // its latest firing time ?
        if (t.clock < $LFT) {
            // if yes increment t clock
            t.clock = t.clock + 1 ;
        } else {
            // otherwise, time cannot elapse,
            //  kill exploration
            abort ;
        }
```

```
GAL sortEx {
  typedef index = 0..3;                              // 0 to  n-2
  array [5] tab = (3,1,2,4,5);
  int tmp = 0;


  transition swap (index $i) [ tab[$i] > tab[$i+1] ]  label "sort"
  { tmp = tab[$i];  tab[$i] = tab[$i+1]; tab[$i+1] = tmp; tmp = 0;   }


  transition sorted  label "sort" [true] {
        for ($i : index) {
                if (tab[$i] > tab[$i+1]) { abort; } } }


  transition sort [true] {
        fixpoint { self."sort"; }
  }                                                    Fonction de Canonisation…

}
```
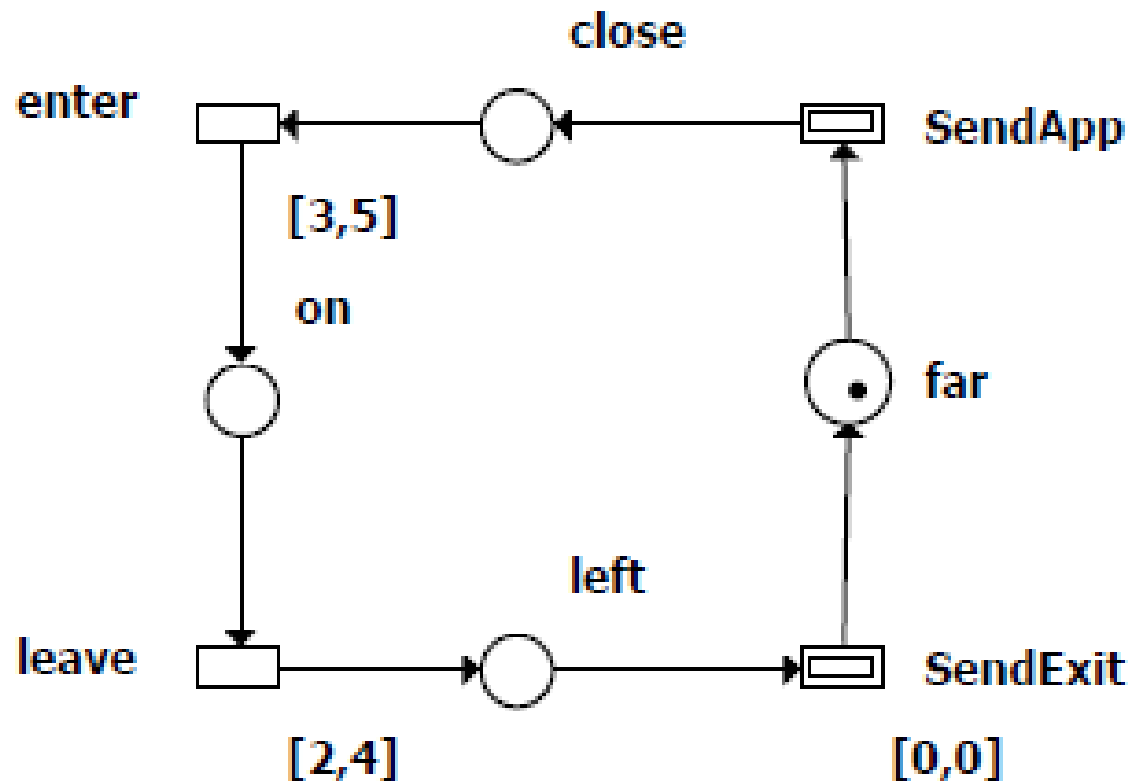
- Petri nets
- Discrete Time Petri nets
- Colored Petri Nets
- Divine (Promela-like) models
- CCSL clock logic
- …

- Each place => a variable
- Each transition => a transition
  - **Guard tests enabling conditions**
  - **Actions update state variables**
- Easy to support many extensions of PN
  - **Test arcs**
  - **Reset arcs**
  - **Inhibitor**
  - **Capacity places**
  - **…**

- **Place -> integer variable,**
  - **initial value=initial marking**
- **Transitions -> define variable t.clock**
  - **unless [0,0] or [0,inf[**
- **Time elapse -> additional transition labeled « elapse »**
  - **Sequence, for each transition t**

```
If (enabled(t)) {
    If ( t.clock < lft(t) ) {
        t.clock=t.clock+1;
    } else {
        abort;
    }
}
 General case
```

```
If (enabled(t)) {
    abort;
}




[0,0] urgent case
```

```
If (enabled(t)) {
    If ( t.clock < eft(t) ) {
        t.clock=t.clock+1;
    }
}


[a,inf[ infinite lft case
```
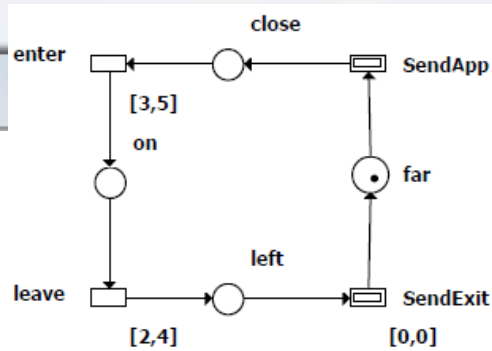
- **Transition t -> transition <l,g,b>**
  - **l: Label is copied from input transition,**
  - **g: enabled(t) and t.clock >= eft(t)**
  - **b:**
    - *update place markings according to arc types and inscriptions,*
    - *reset current clock,*
    - *reset any disabled transition clocks (call(reset))*
- **Reset disabled transitions: <reset,true,b> for each transition t :**

```
If (! enabled(t)) {
            t.clock=0;
}
 General case
```

```


  If no clock do nothing
```

```
GAL train {
    int far = 1 ;
    int close = 0 ;
    int on = 0 ;
    int left = 0 ;
    int enter.clock = 0 ;
    int leave.clock = 0 ;

    transition SendApp [ far >= 1 ] label "SendApp" {
        far = far - 1 ;
        clo
        sel
    }

    transitio
        clo
        on
        ent
        sel
    }

    transitio
        on
        lef
        lea
        sel........ ;
    }

    transition SendExit [ left >= 1 ] label "SendExit" {
        left = left - 1 ;
        far = far + 1 ;
        self.reset ;
    }
}
```

```
    transition elapse [ True ] label "elapse" {
        if (close >= 1) {
            if (enter.clock < 5) {
                enter.clock = enter.clock + 1 ;
            } else {
                abort ;
            }
        }
```

```
                                        [
                                        eave.clock + 1 ;
```

```
                                        eset" {

                                        ,
            if (! on >= 1) {
                leave.clock = 0 ;
            }
        }
    }
}
```

transition enter [close >= 1 && enter.clock >= 3]{

    close = close - 1 ;

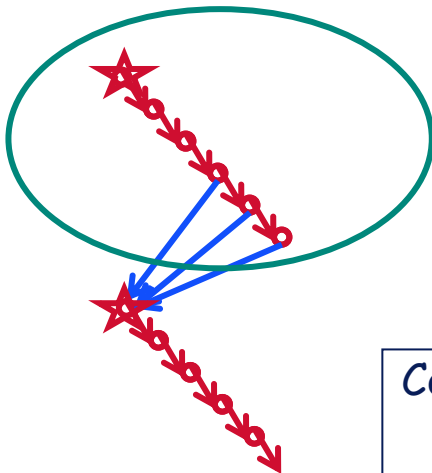   **self**.reset ;

    on = on + 1 ;

    enter.clock = 0 ;

}

- **Essential states construction [Popova]**

  - **Letting time elapse cannot disable transitions**
  - **Let time progress, only consider states that immediately follow a discrete transition**

↘ Time Elapse

↘ Discrete Transition

☆ Essential State

Compute green set :
        let time elapse if it can
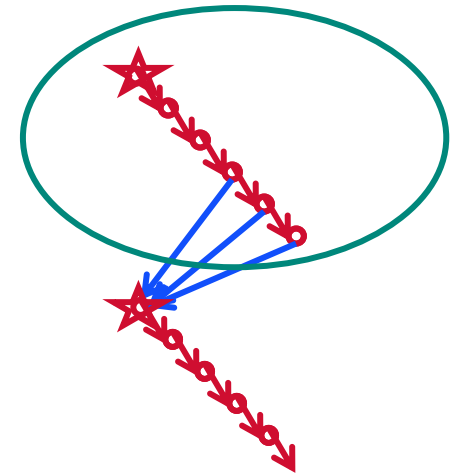   cumulate states
Fire transitions (succ) from resulting set

```
GAL tpnModel ($EFT = 3, $LFT = 5) {
    int a = 1 ;
    int b = 0 ;
    int t.clock = 0 ;

    transition t [a >= 1 && t.clock >= $EFT] label "succ"
    {
        a = a - 1 ;
        b = b + 1 ;
        t.clock = 0 ;
    }
```

```
transition elapseIfpossible
   [ a >= 1 && t.clock < $LFT] label "elapseIP" {
      t.clock = t.clock + 1 ;
}
  transition id [true] label "elapseIP" {
}
transition nextState [true] {
  fixpoint {
    self."elapseIP" ;
  }
  self."succ" ; // Any discrete transition
 }
}
```

- Each place => an array of dimension proportional to domain
  - **Uncolored places => size 1**
- Formal parameters => typedef of a range
- Each transition => transition with parameters

```
Class
  Quality is 1..M;
  Products is 1..N;
  Options is 1..N;
Var
  p in Products;
  o1, o2, o3 in Options;
  x in Quality;
```
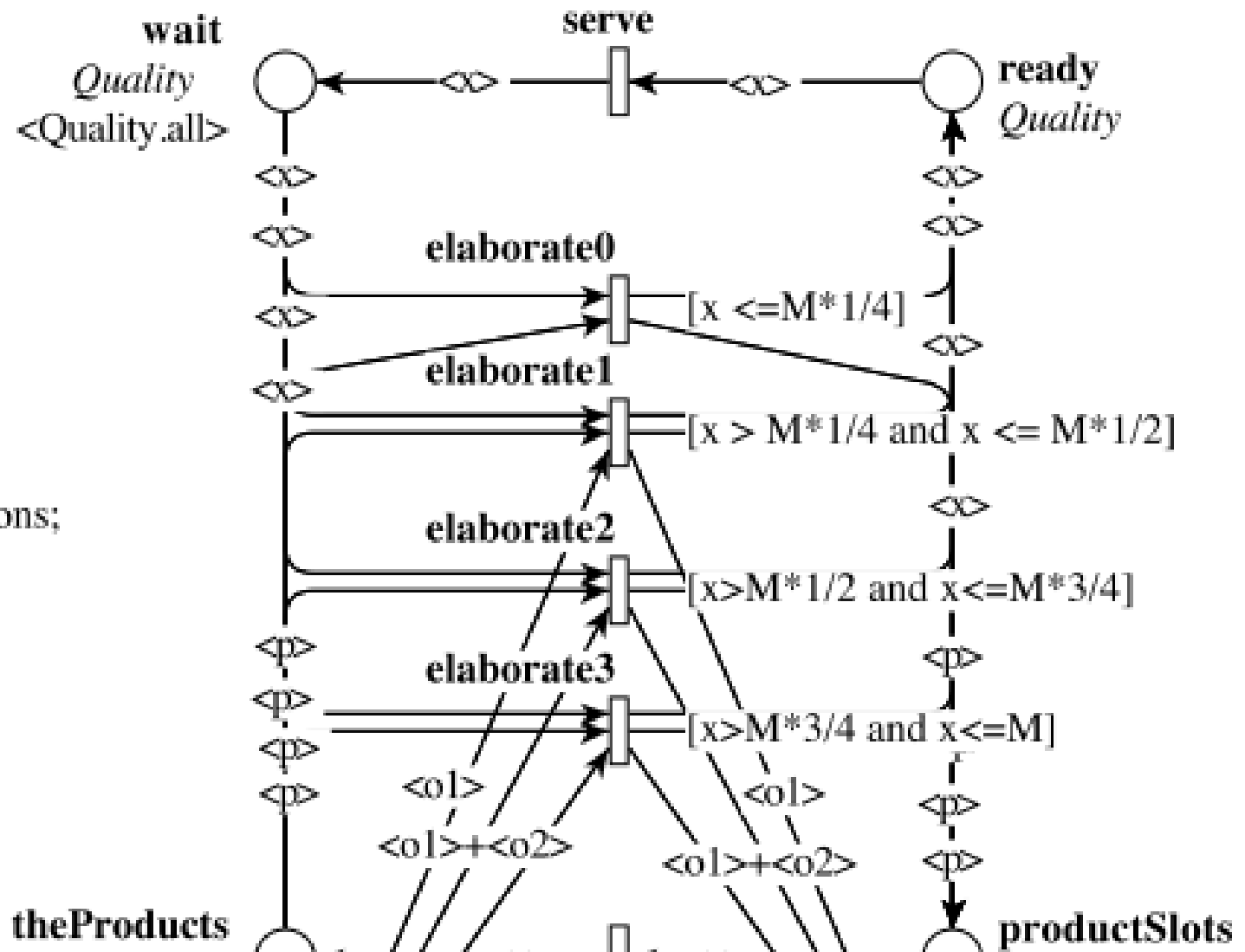
```
GAL DrinkVending2 {
typedef Options = 0 .. 1 ;
typedef Products = 0 .. 1 ;
typedef Quality = 0 .. 7 ;
array [8] ready = (0, 0, 0, 0, 0, 0, 0, 0) ;
array [8] wait = (1, 1, 1, 1, 1, 1, 1, 1) ;
array [2] theProducts = (1, 1) ;
array [2] productSlots = (0, 0) ;
array [2] theOptions = (1, 1) ;
array [2] optionSlots = (0, 0) ;
```
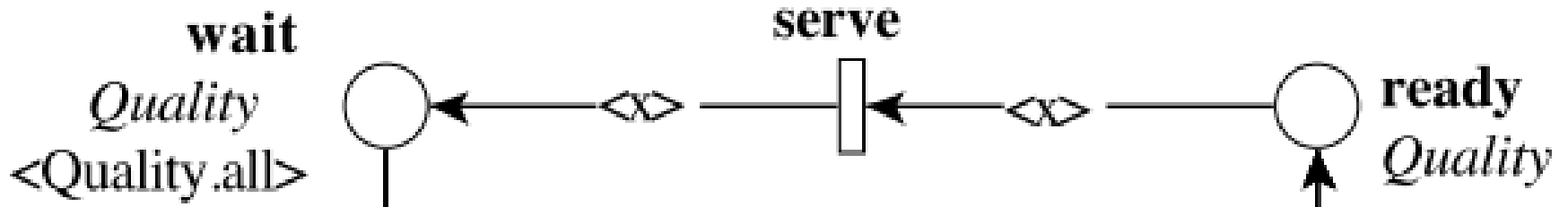
Color Domains
M=8, N=2

Places

```
transition serve (Quality $x) [ready [$x] >= 1] {
        ready [$x] = ready [$x] - 1 ;
        wait [$x] = wait [$x] + 1 ;
}
```

```
byte id;
byte t[3] = { 255 ,255 ,255  };
process P_0 {
        state NCS, try, wait, CS;
        init NCS;
trans
        NCS -> try
{ guard id == 0; effect t[0] = 2;},
        try -> wait
{ effect t[0] = 3, id =0 +1; },
        wait -> wait
{ guard t[0] == 0; effect t[0] = 255;}, …
```

Global Variables

Process + Channels

- Translate Divine concepts to GAL

  - **Process state => variable**
  - **Divine variables and arrays => GAL equivalent**
  - **Guards, Instructions => GAL equivalent**
  - **Synchronizations => use GAL call semantics**
  - **Channels => GAL arrays + variable for size**

- Symbolic data structures: BDD, MDD

  - **k boolean variables => 2^k**

- Encode transitions with sets : $2^k \times 2^k$

- Use the support (only k' vars) of transitions

  - **Build clusters of transitions**

- Reorder evaluations in fixpoint computation (chaining, saturation)

- But :

  - **Exponential worst case complexity when k' grows**

 - **In general, necessary to invoke an explicit solver for each new state in $2^{k'}$**

- **A system to relate explicit and symbolic engines**
- **Interface :**

  - **System is a fixed set of integer variables**

  - **A transition is declared as an opaque function through its *support, i.e.* set of variables impacted**

- **Algorithm for each transition :**

  - **Store as a DD projection of encountered states on support**

  - **Execute (explicit) transitions on new states only, store resulting states and transition DD**

- Computing the support
  - A[x + y] => pessimistic assumptions
  - x=x+1; y=y+1 => Atomic sequence of updates produce artificially large support
- What if we could compute this on the fly ?
  - Carry the expression in a dedicated operation
  - Traverse a state -> path
  - Resolve variables as they are encountered
  - Drop pessimistic assumptions ASAP
- But we must still reason with sets !

- **Partial expression evaluation**
  - **f = a + b**
  - **States : s1:(a=1, b=0)   s2:(a=0,b=1)**
  - **If both a and b are known : f(s1)=f(s2),**
          s1 and s2 are equivalent
  - **If only a is known, f(s1)=1+b   f(s2)=0+b**
          s1 and s2 are NOT equivalent
- **Algorithm discovers variable values and builds equivalence classes on the fly**
  - **Split a node into a partition w.r.t the value of the expression**
  - **Cache the partition to reuse result in different computations**

Performance evaluation (BEEM)
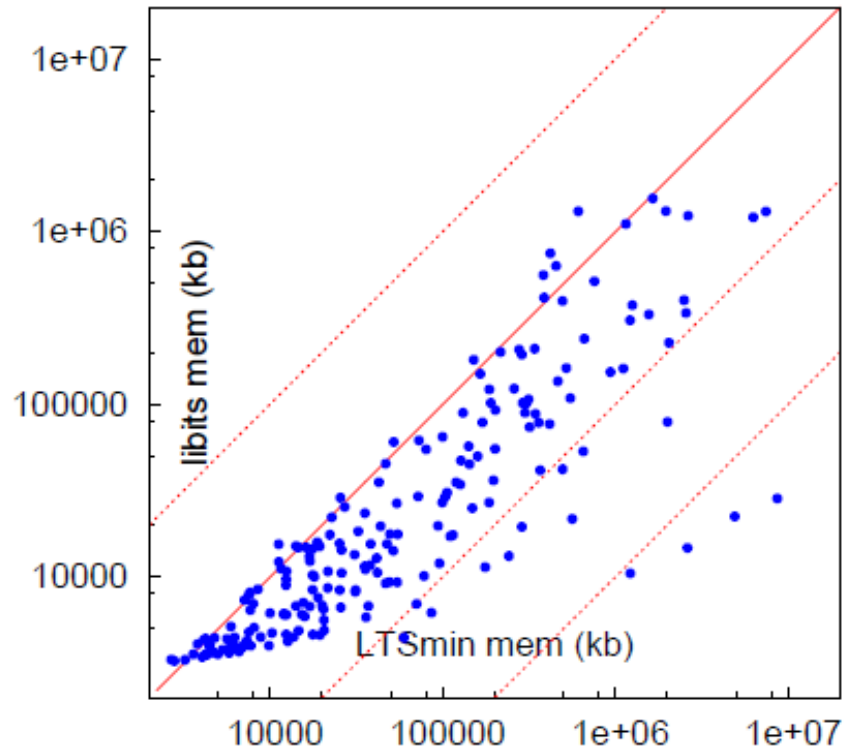
(a) comparison on peak memory
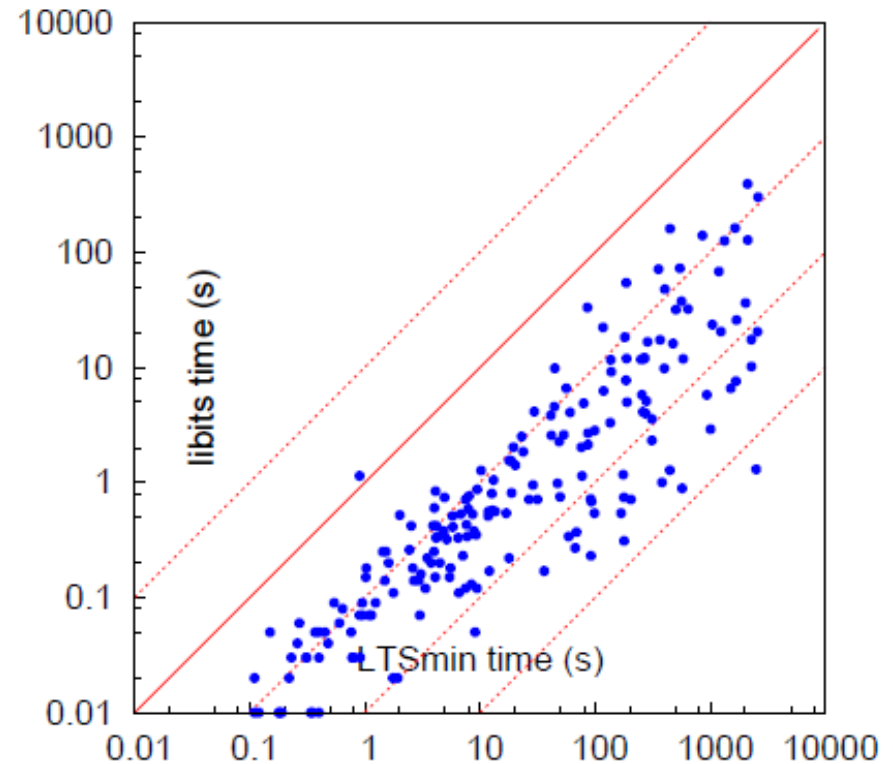
(b) comparison on time

- reachability properties: 4 cores, 900s wall-clock, 1Gb (HWMCC)
- there are difficult instances for both tools

UNSAT
SAT

instances	456
libits	376
super_prove	282
both	258
none	56
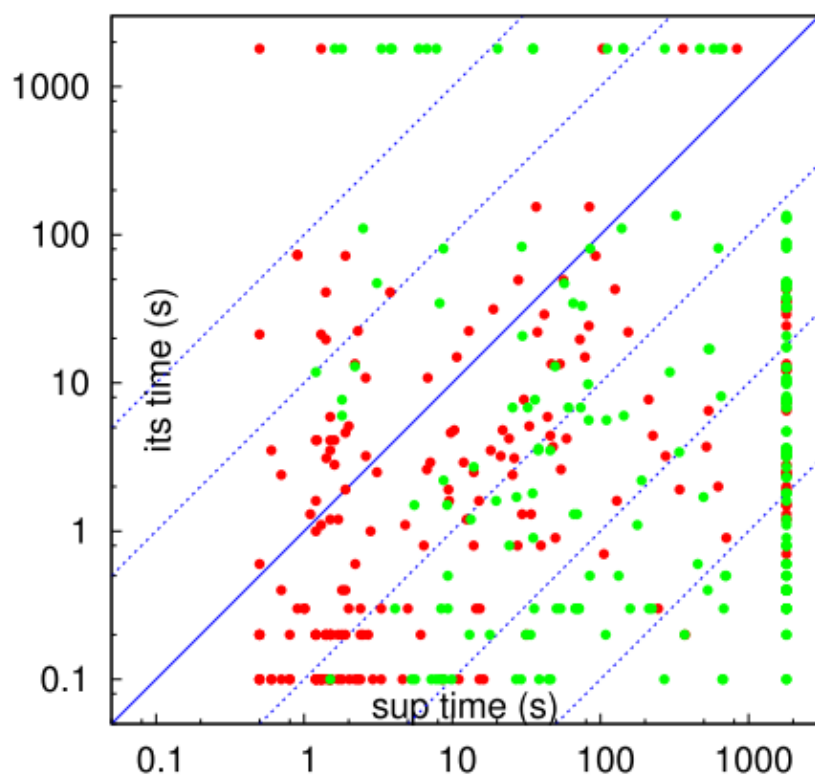
- **GAL modeling**

  - **A natural model for many discrete semantics**
  - **Efficient symbolic solution :**
    - *More transparency = more optimizations*

- **ITS Composite for compositional modeling**

  - **Modular and hierarchical specifications**
  - **Efficient support for symmetric models**

- **Model checking engine**

  - **Reachability (shortest traces)**
  - **CTL (Forward algorithms, traces)**
  - **LTL with Spot (Fully symbolic or hybrid)**
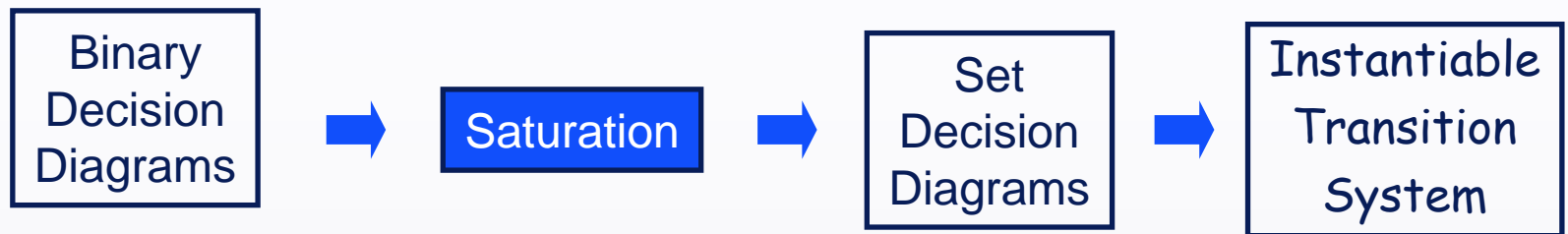
GAL Eclipse plugin (Thanks Xtext !)

SDD and ITS-tools are distributed as an open-source LGPL/GPL C++ source and pre-compiled tools :

http://ddd.lip6.fr

Eclipse plugin for GAL/ITS manipulation and CTL model-checking

http://coloane.lip6.fr/night-updates

The Saturation Algorithm for Decision diagrams

| Binary Decision Diagrams | → | **Saturation** | → | Set Decision Diagrams | → | Instantiable Transition System |

- Model-checking using decision diagrams => (nested) transitive closures over the transition relation
- Optimizing complexity of this operation critical to efficiency
- [BCM'92] based on <u>BFS style iterations</u>, n iterations required where n is depth of "deepest" state

$s0$ $N(s0)$ $N^2(s0)$
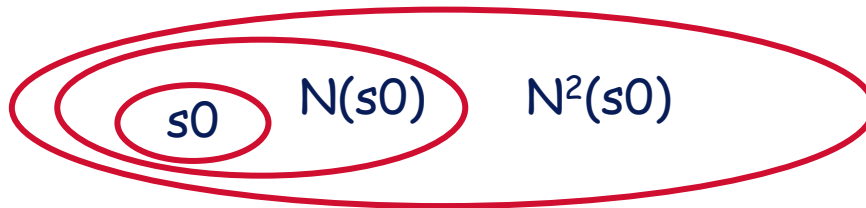
- [Roig'95] <u>Chaining</u> may converge faster, based on clusters of transitions, no longer strict BFS

$s0$ $t1(s0)$ $t2(t1(s0))$ $t2(t1(t2(t1...(s0)))..))$
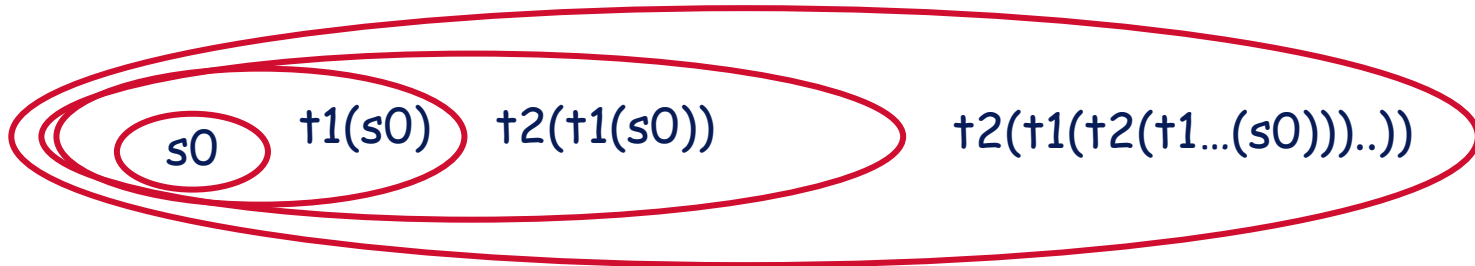
- [Ciardo'01] <u>Saturation</u> is empirically 1 to 3 orders of magnitude better

- **Saturation algorithm: [Ciardo et al. TACAS'01]**
  - Fire transitions from the leaves (terminals) up to root
  - Go to ancestor of a node iff. The current node is saturated : all events that only affect this variable and variables below it have been fired until a fixpoint is reached
  - Each time a node is affected by an event, resaturate it.
- **Not BFS anymore, firing order of events follows data structure**
  - Huge reduction of time and space complexity
  - Good tackling of intermediate peak size effect
- **However :**
  - Definition of saturation algorithm is complex
  - Cannot be implemented directly with public API of DD libraries

Our contribution : Automatic saturation

- The transitive closure or *fixpoint* noted * is a unary operator
- Evaluated by $h^*(d)$:
  - repeat : d = h(d)
  - until : d == h(d)
- Evaluation may not terminate
  - depends on the homomorphism
  - if it does, evaluation described as finite composition :
    - $h^* (d) = h \circ h \circ ... h (d)$
  - Thus h^* is a homomorphism
- To cumulate states, use of a common construction :
  - $(h + id)^*$
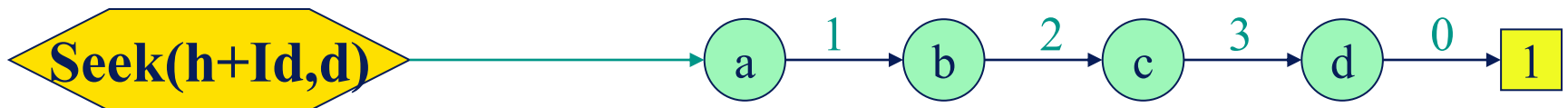- Allows to implement a leaf to root saturation strategy

Fixpoint : an example

$$\text{Seek}(h,v)(e,x) = \begin{cases} h^* \circ e \xrightarrow{X} \text{Id} & \textit{if } v=e \\ e \xrightarrow{X} \text{Seek}(h,v) & \textit{otherwise} \end{cases}$$

$$\text{Seek}(h,v)(1) = T$$

$$\text{Max}(z)(e,x) = \begin{cases} e \xrightarrow{X} \text{Id} & \textit{if } x<z \\ 0 & \textit{otherwise} \end{cases}$$

$$\text{Max}(z)(1) = T$$

$$h = \text{Max}(3)\circ \text{Inc}(d) \qquad \text{// Increment } d \text{ up to 2}$$

Seek(h+Id,d) → (a) —1→ (b) —2→ (c) —3→ (d) —0→ [1]

Fixpoint : an example

$$\text{Seek}(h,v)(e,x) = \begin{cases} h^* \text{ o } e \xrightarrow{X} \text{Id} & \text{if } v=e \\ e \xrightarrow{X} \text{Seek}(h,v) & \text{otherwise} \end{cases}$$

$$\text{Seek}(h,v)(1) = T$$

$$\text{Max}(z)(e,x) = \begin{cases} e \xrightarrow{X} \text{Id} & \text{if } x<z \\ 0 & \text{otherwise} \end{cases}$$

$$\text{Max}(z)(1) = T$$

$$h = \text{Max}(3)\text{o Inc}(d) \qquad // \text{ Increment } d \text{ up to 2}$$

a single traversal of these nodes

Fixpoint : an example
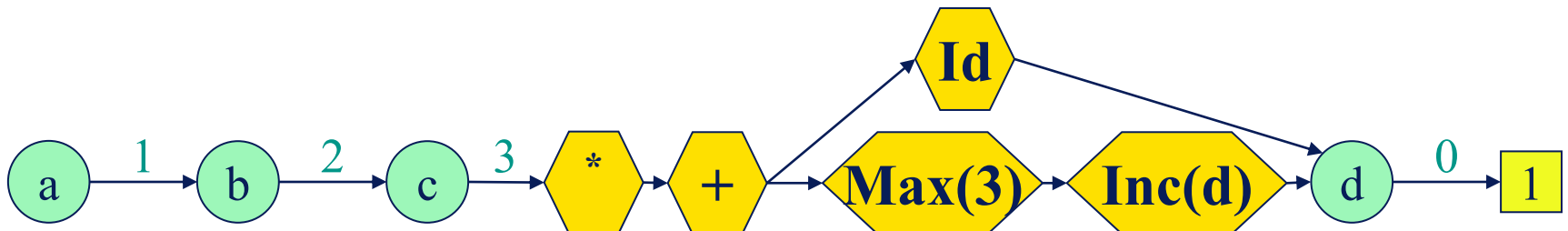
$$Seek(h,v)(e,x) = \begin{cases} h^* \text{ o } e \xrightarrow{X} Id & \textit{if } v=e \\ e \xrightarrow{X} Seek(h,v) & \textit{otherwise} \end{cases}$$

$$Seek(h,v)(1) = T$$

$$Max(z)(e,x) = \begin{cases} e \xrightarrow{X} Id & \textit{if } x<z \\ 0 & \textit{otherwise} \end{cases}$$

$$Max(z)(1) = T$$

$$h = Max(3) \text{ o } Inc(d) \qquad \textit{// Increment } d \textit{ up to 2}$$

$$\text{Seek(h,v)(e,x)} = \begin{cases} h^* \circ e \xrightarrow{\text{X}} \text{Id} & \textit{if } v=e \\ e \xrightarrow{\text{X}} \text{Seek(h,v)} & \textit{otherwise} \end{cases}$$

$$\text{Seek(h,v)(1)} = T$$

$$\text{Max(z)(e,x)} = \begin{cases} e \xrightarrow{\text{X}} \text{Id} & \textit{if } x<z \\ 0 & \textit{otherwise} \end{cases}$$

$$\text{Max(z)(1)} = T$$

$$h = \text{Max(3)} \circ \text{Inc(d)} \qquad \text{// Increment } d \text{ up to 2}$$
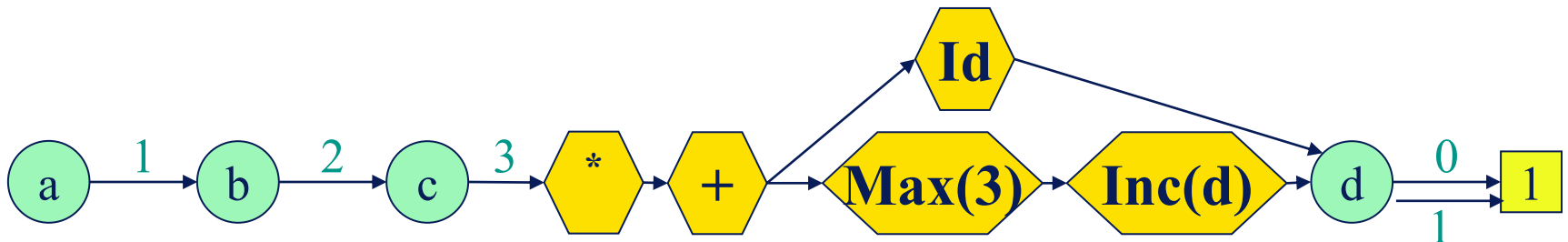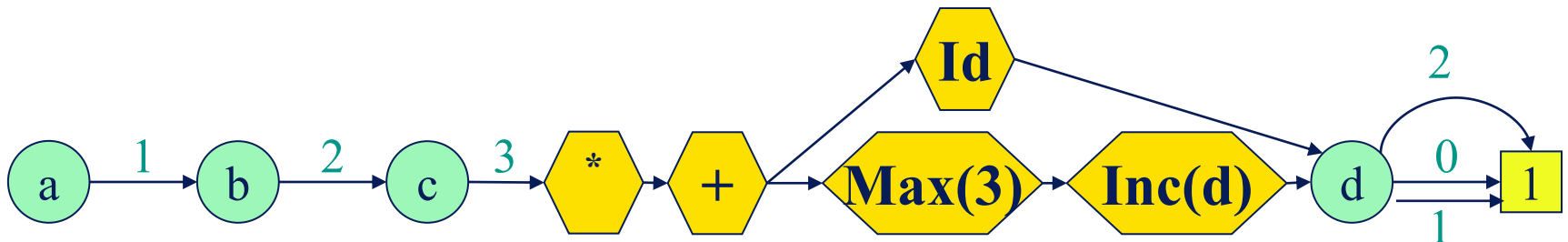
Fixpoint : an example

$$\text{Seek}(h,v)(e,x) = \begin{cases} h^* \circ e \xrightarrow{X}> \text{Id} & \textit{if } v=e \\ e \xrightarrow{X}> \text{Seek}(h,v) & \textit{otherwise} \end{cases}$$

$$\text{Seek}(h,v)(1) = T$$

$$\text{Max}(z)(e,x) = \begin{cases} e \xrightarrow{X}> \text{Id} & \textit{if } x<z \\ 0 & \textit{otherwise} \end{cases}$$

$$\text{Max}(z)(1) = T$$

$$h = \text{Max}(3) \circ \text{Inc}(d) \qquad \textit{// Increment } d \textit{ up to 2}$$
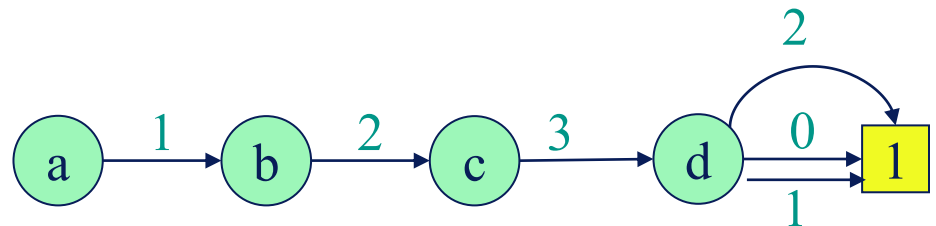
Fixpoint : an example

$$Seek(h,v)(e,x) = \begin{cases} h^* \text{ o } e \xrightarrow{X} Id & \textit{if } v{=}e \\ e \xrightarrow{X} Seek(h,v) & \textit{otherwise} \end{cases}$$

$$Seek(h,v)(1) = T$$

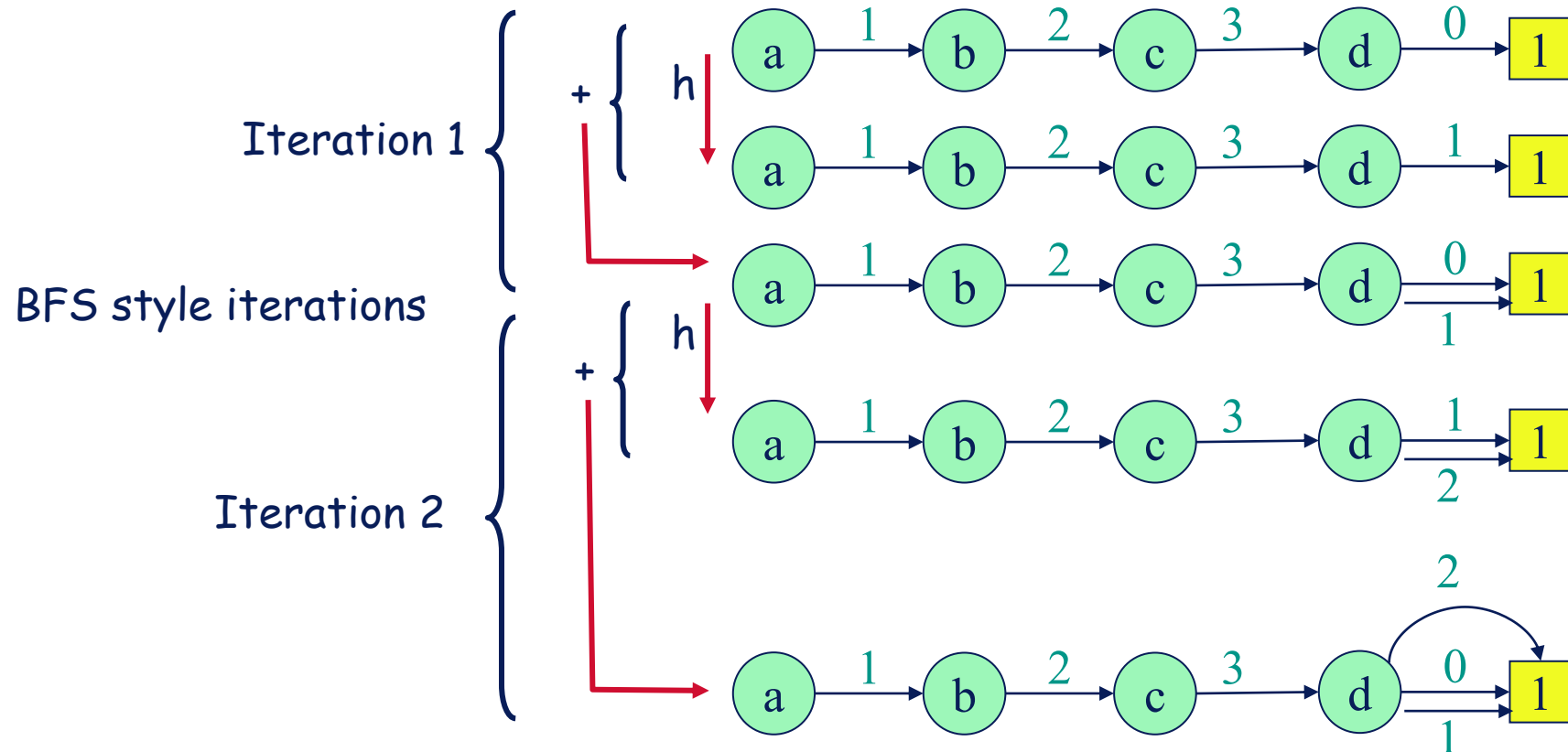$$Max(z)(e,x) = \begin{cases} e \xrightarrow{X} Id & \textit{if } x{<}z \\ 0 & \textit{otherwise} \end{cases}$$

$$Max(z)(1) = T$$

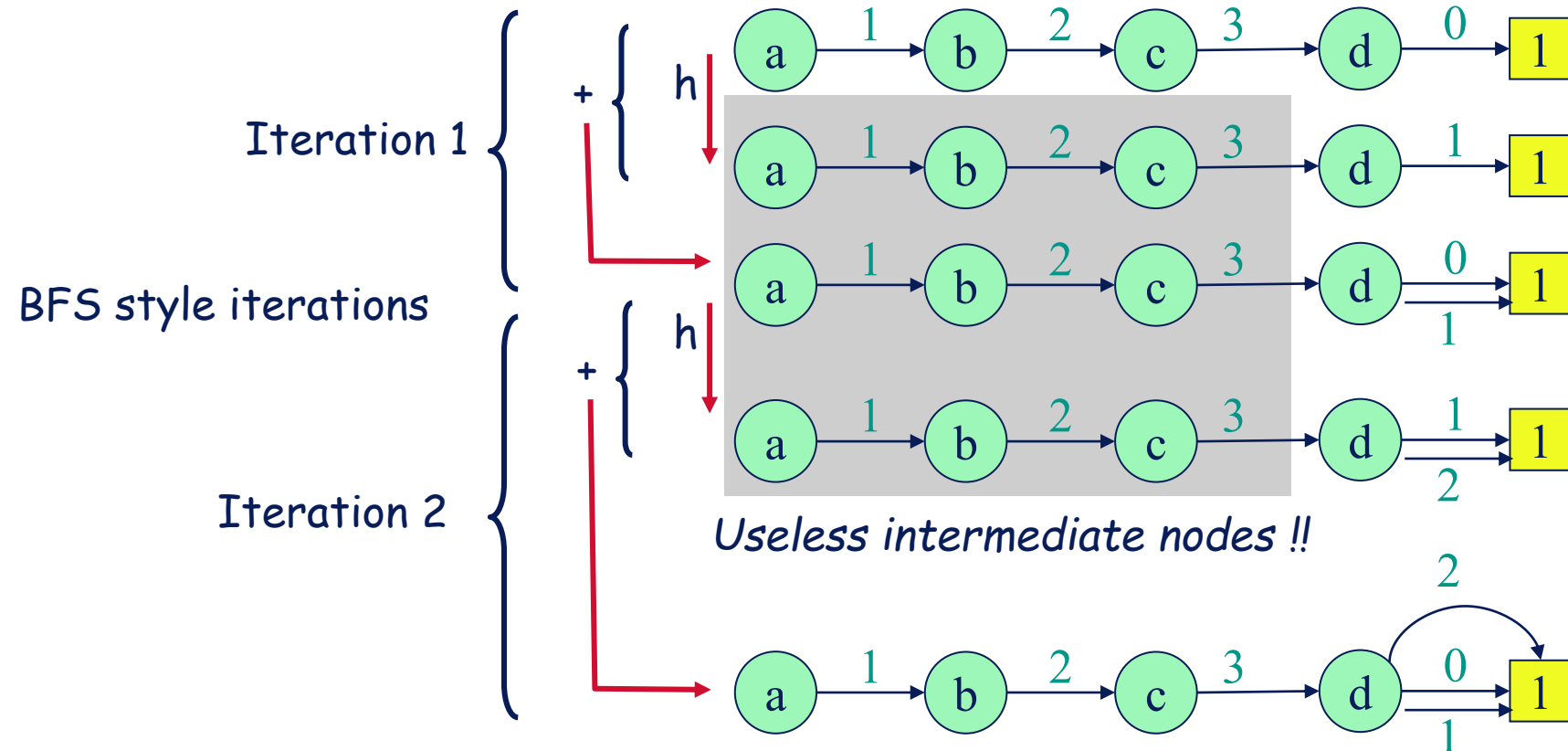$$h = Max(3) \text{o } Inc(d) \qquad // \text{ Increment } d \text{ up to 2}$$

- **Transitive closure or fixpoint allows:**
    - **single traversal of the top of the tree**
    - **less intermediate nodes**

Fixpoint conclusions

- **Transitive closure or fixpoint allows:**
  - **single traversal of the top of the tree => cost of + and h**
  - **less intermediate nodes**

Iteration 1

BFS style iterations

Iteration 2

Useless intermediate nodes !!

- **Nested transitive closure or fixpoint = saturation allows:**
  - *single traversal of the top of the tree => cost of + and h*
  - **less intermediate nodes**

Useless intermediate nodes !!
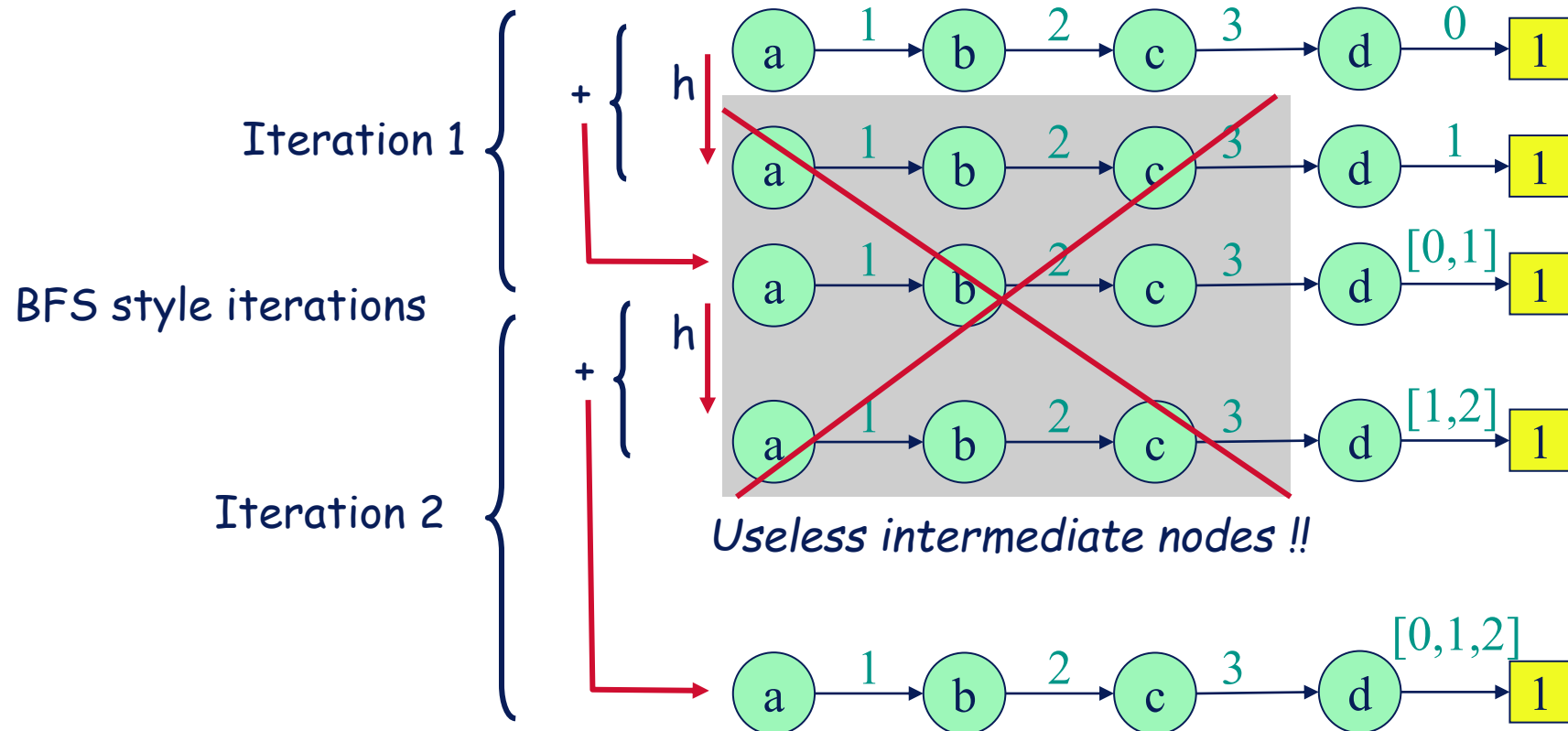
- **Transitive closure allows more efficiency**
  - **Manual Saturation "à la Ciardo" (Tacas'01 and '03) using * operator**
  - **Organize events by highest variable affected**

Model	N	Nb. States	final nodes	PNDDD no sat total nodes	PNDDD no sat time (s)	PNDDD sat total nodes	PNDDD sat time (s)
Dining Philosophers	50	2.23e+31	1387	13123	11.6	10739	0.09
	100	4.97e+62	2787	26823	54.19	21689	0.18
	200	2.47e+125	5587	54223	234	43589	0.39
	1000	9.18e+626	27987	–	–	218789	2.1
Slotted Ring Protocol	10	8.29e+09	1281	35898	83.07	45970	0.8
	15	1.46e+15	2780	118054	595	132126	2.26
	50	1.72e+52	29401	–	–	3.58e+06	61.58
Flexible Manufacturing System	10	2.50+09	580	8604	2.06	11202	0.17
	25	8.54e+13	2545	50489	28.75	85962	1.58
	50	4.24e+17	8820	231464	240.4	490062	9.78
	80	1.58e+20	21300	–	–	1.72e+06	37.06
Kanban	10	1.01e+09	257	26862	20.47	5837	0.06
	50	1.04e+16	3217	–	–	209117	3.96
	100	1.73e+19	11417	–	–	1.32e+06	28.09
	200	3.17e+22	42817	–	–	9.23e+06	238.95