FLOLAC 2025

Hardware Model Checking

Prof. Chung-Yang (Ric) Huang (黃鐘揚)
Design Verification Lab, EE/GIEE, NTU

2025.08.12

About me...

黃鐘揚 (Ric), PhD

台大電機系/電子所教授
優拓資訊Yoctol.AI創辦人
台灣數據智慧發展協會理事
晶睿通訊 & 迪樂科技集團獨立董事
前台大創創中心、創創學程副主任
Sr. RD Mgr, Verplex/Cadence

NTU GIEE Design Verification Lab

Formal Verification of IC Designs

AI-Powered Design for Verifiability

Quantum Circuir Synthesis

3D-IC Feasibility Analysis

Outline

1. Challenges in design verification
2. What is formal verification & Model Checking?
3. BDD-based verification techniques
4. Introduction to Boolean Satisfiability (SAT)
5. Why modern SAT prevails?
6. Bounded Model Checking (BMC)
7. Abstraction and Refinement
8. Interpolation-based over-approximated image computation
9. Property-Directed Reachability (PDR)

Disclaimer:

這份講義將我在臺大電子所將近半個學期的課程內容濃縮在將近 210 頁的投影片裡頭

可能有點硬，但我會盡量 top-down 的講解來讓大家一窺 Model Checking 的技術。聽不懂的地方請隨時發問，講不完的話我也不會勉強，會以讓大家有辦法吸收為原則

In this talk, we will focus on

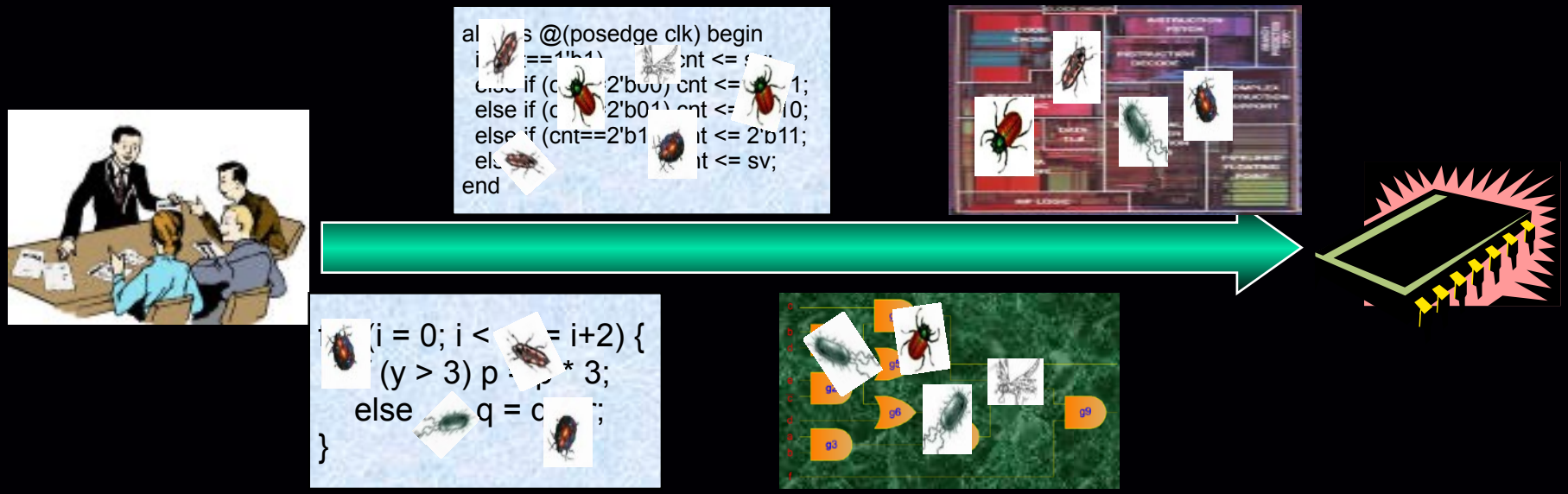
"**Hardware Verification**",

and by "hardware" we actually mean "(Digital) ICs".

Hardware verification is to verify whether the "design" of an IC is correct (i.e. bug-free).

What is Design Verification?

To verify the correctness of your design

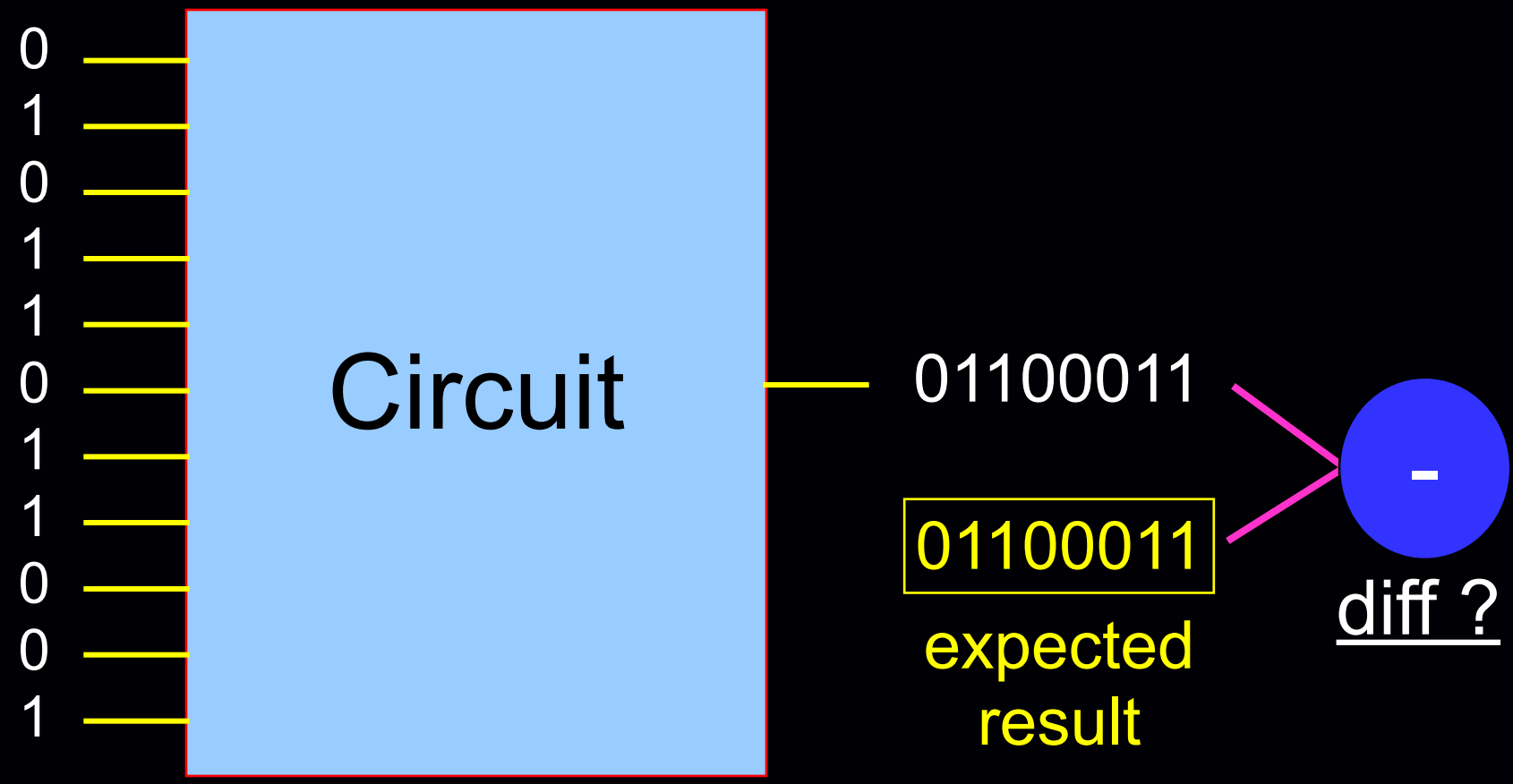


Bugs may exist everywhere in the design...

➔ Find as many design bugs as possible !!

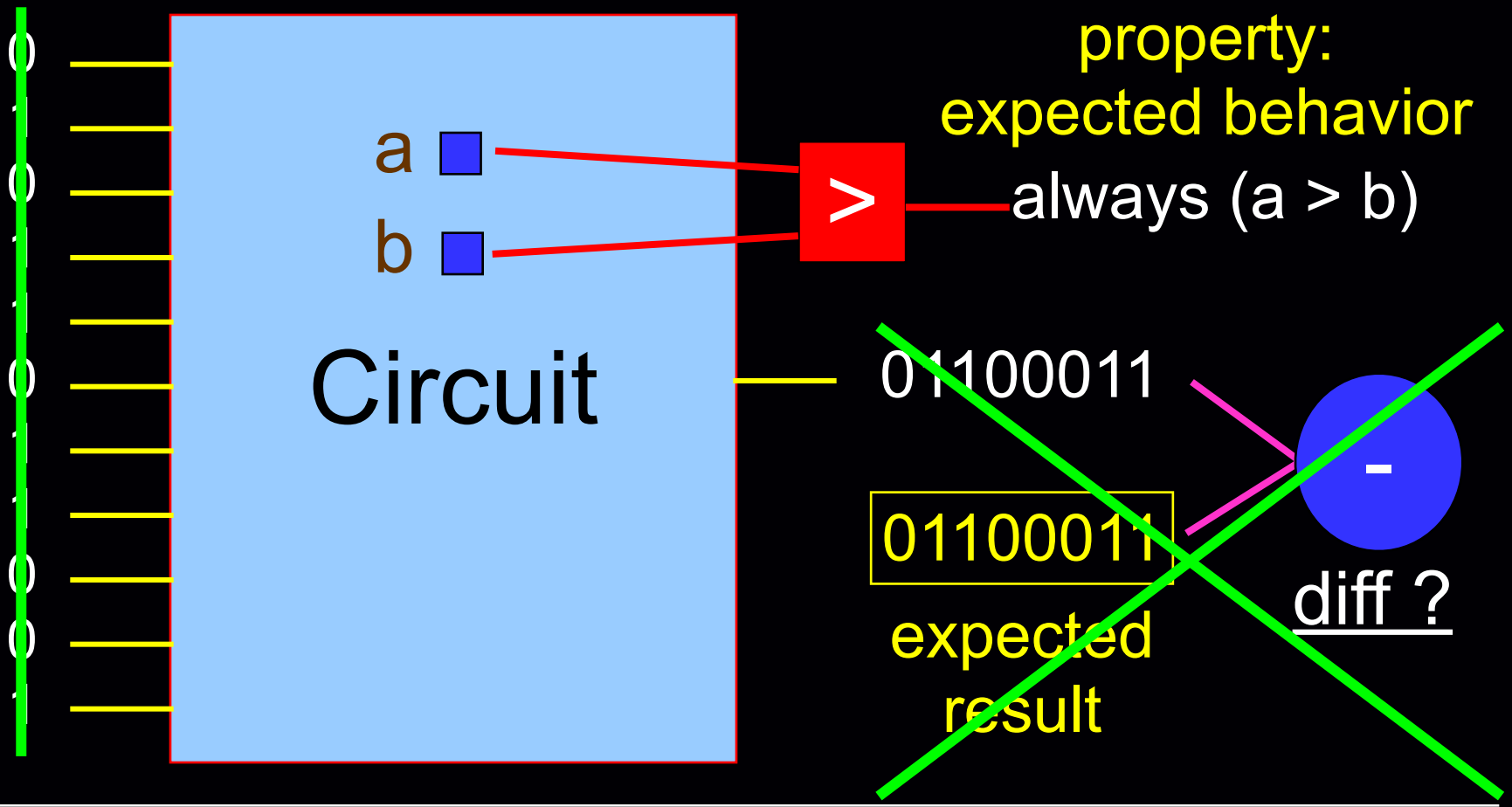
Approaches for Design Verification

1. Simulation-based approach



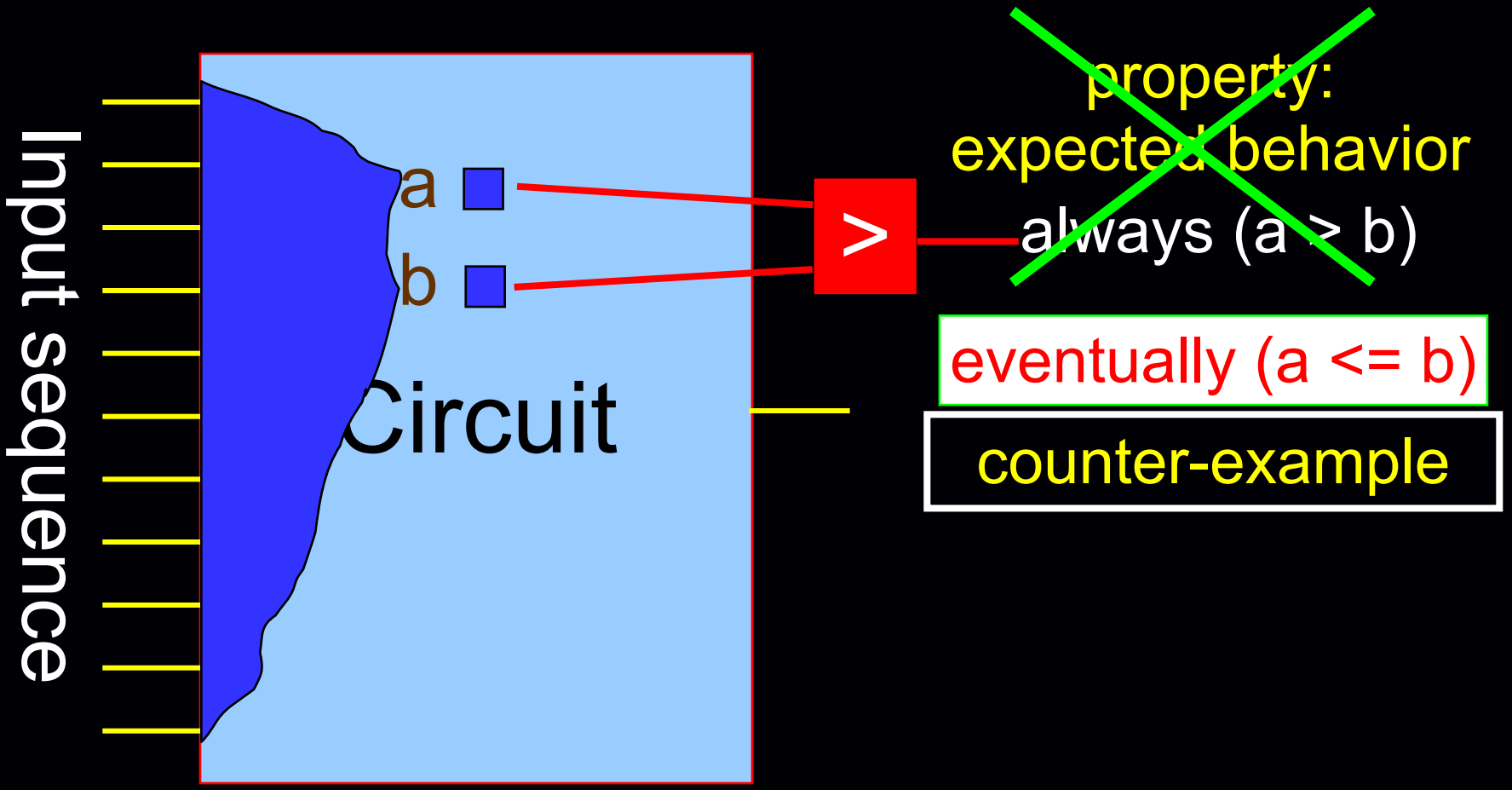
Approaches for Design Verification

2. Formal verification approach



Approaches for Design Verification

2. Formal verification approach



Challenges in Design Verification

- For simulation-based approaches
  - Doubly-exponential growth
  - Being the player and the judge
  - Corner-case bugs are extremely hard
  - Difficult to witness "eventuality" and "deadlock" bugs
- For formal verification approaches
  - Even worse scalability issues
  - High learning curves
  - When proof is incomplete... what do we get?
  - Over- and under-constrained issues

Focuses of the talk: "How it works?", "What to expect?"

Definition of "Formal Verification"

"The expression 'formal verification', as it appears in the literature, refers to a variety of (often quite different) methods used to prove that a model of a system has certain specified attributes. What distinguishes 'formal' verification from other undertakings also called 'verification' is that 'formal' verification conveys a promise of mathematical certainty. The certainty is that if a model is formally verified to have a given attribute, then no behavior or execution of the model ever can be found to contradict this"

Robert Kurshan, "Computer-Aided Verification of Coordinating Processes"

When formally verifying the design,
we usually formally "model"
the design under verification as a
"**Finite State Concurrent System**".

"**Model Checking**"
is the most widely studied and used
verification technique.

Model Checking Problem

- Let M be the state transition graph obtained from the concurrent system.
- Let *f* be the specification expressed in temporal logic.

- Model Checking
  - Find all states s of M such that M, s $\models$ *f*

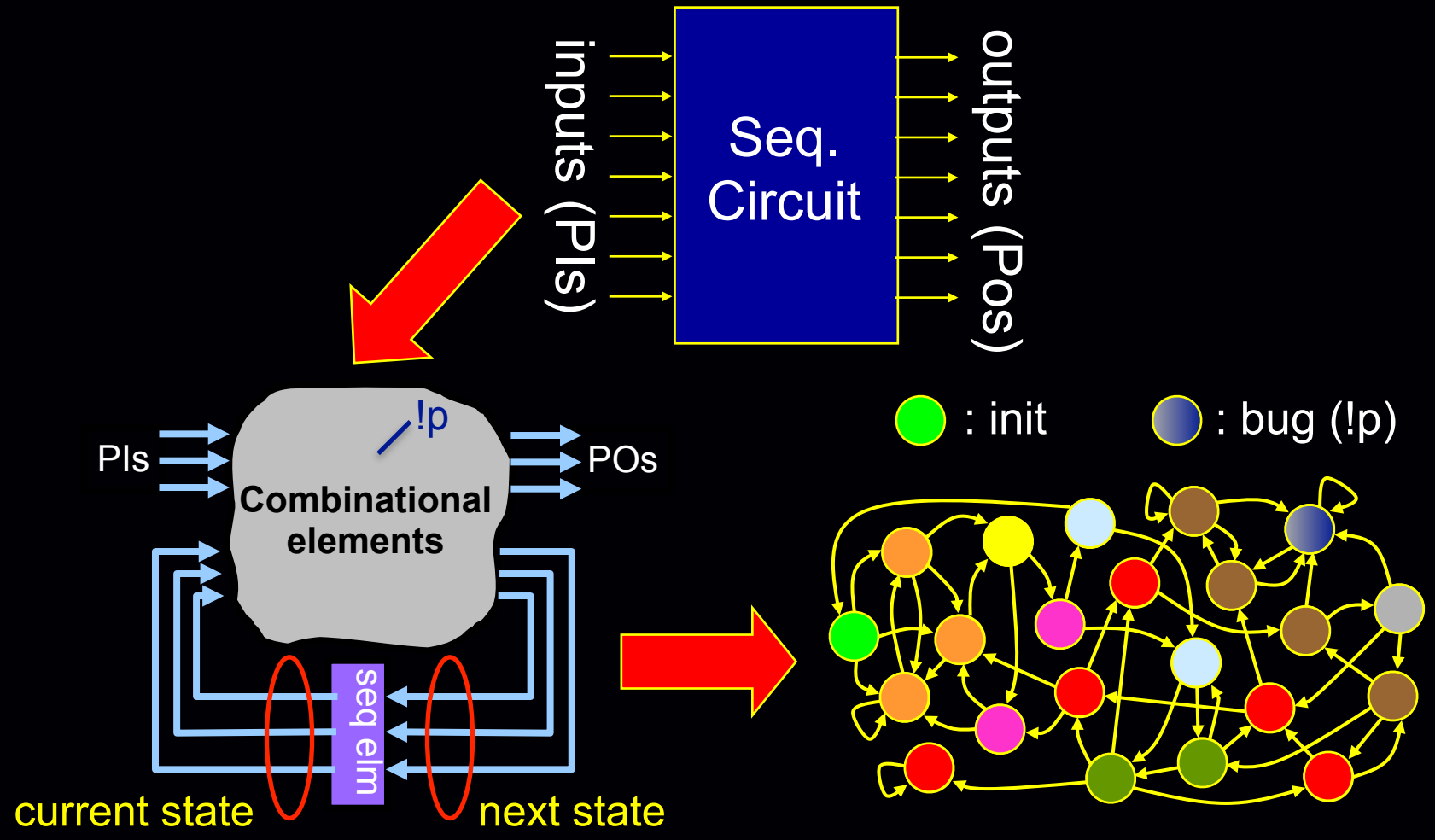
The Process of Model Checking

1. Modeling
   - Convert a design into a formalism accepted by a model checking tool
   - Parsing, compilation, abstraction, reduction, etc
2. Specification
   - What are the properties the design must satisfy?
   - e.g. Temporal logic
3. Verification
   - Try to prove that the model is compliant with the specification
   - If not, manual debugging is usually required

"Model Checking", E.M. Clarke, et al.

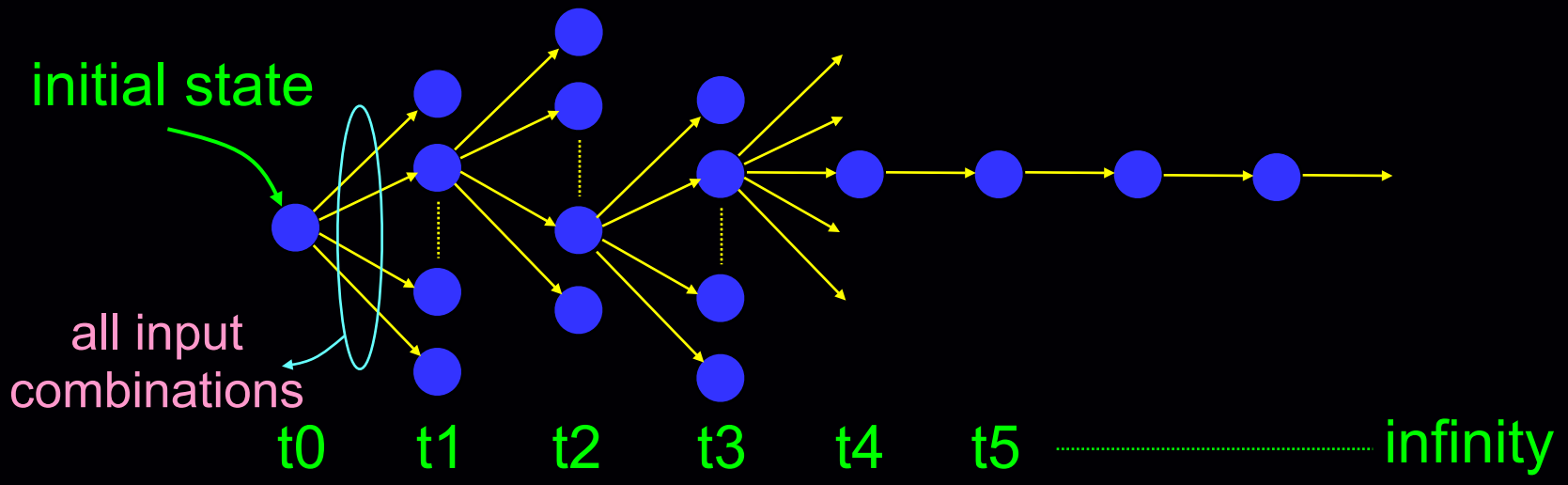
Formal Modeling of a Sequential Circuit

- It can be easily shown that a synchronous digital circuit can be converted to a kripke structure

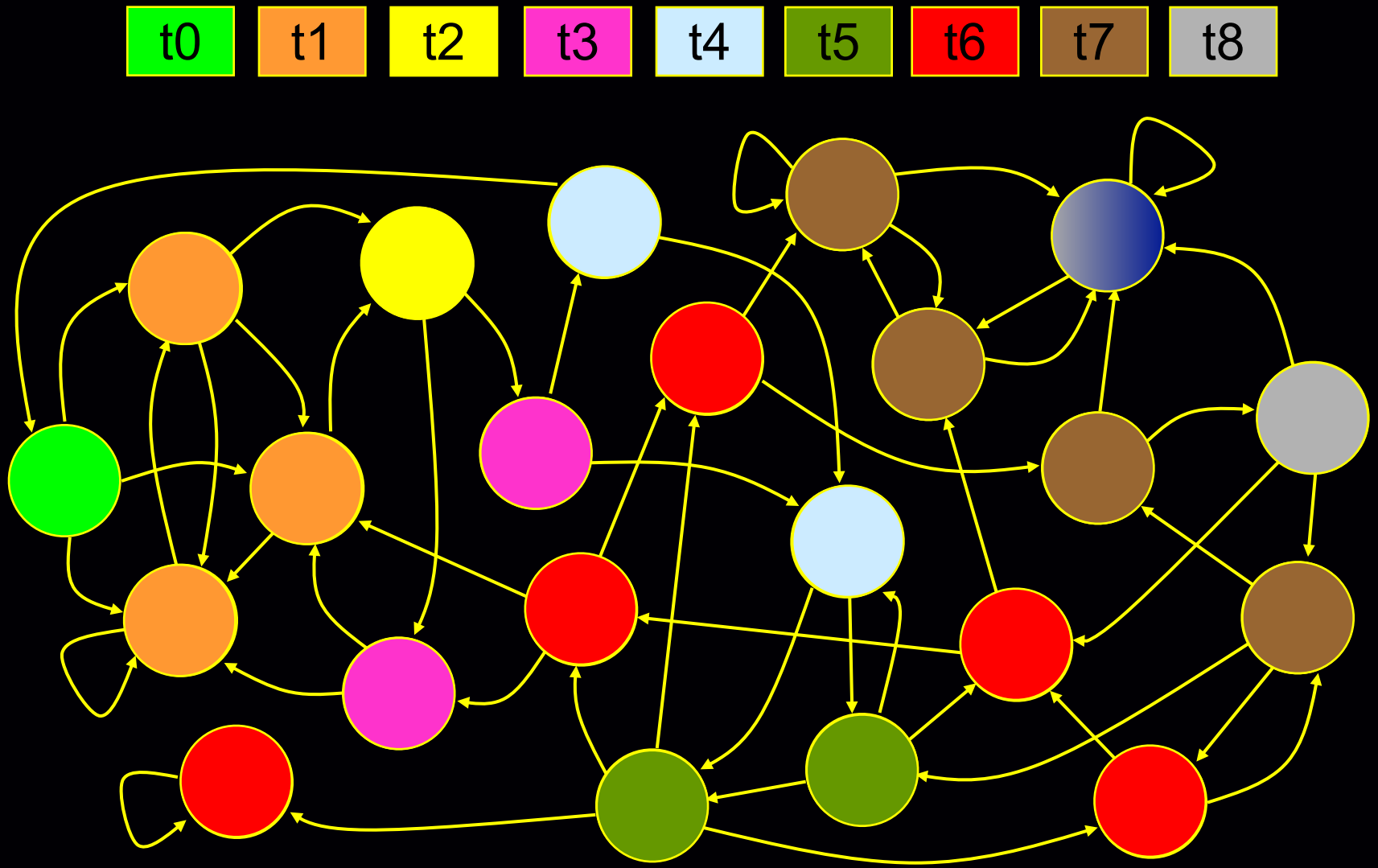


"Mathematical Certainty" in Formal Verification

- Space exhaustiveness
  - Verify all input combinations of the system
- Time exhaustiveness
  - Verify system behavior from initial state to time infinity

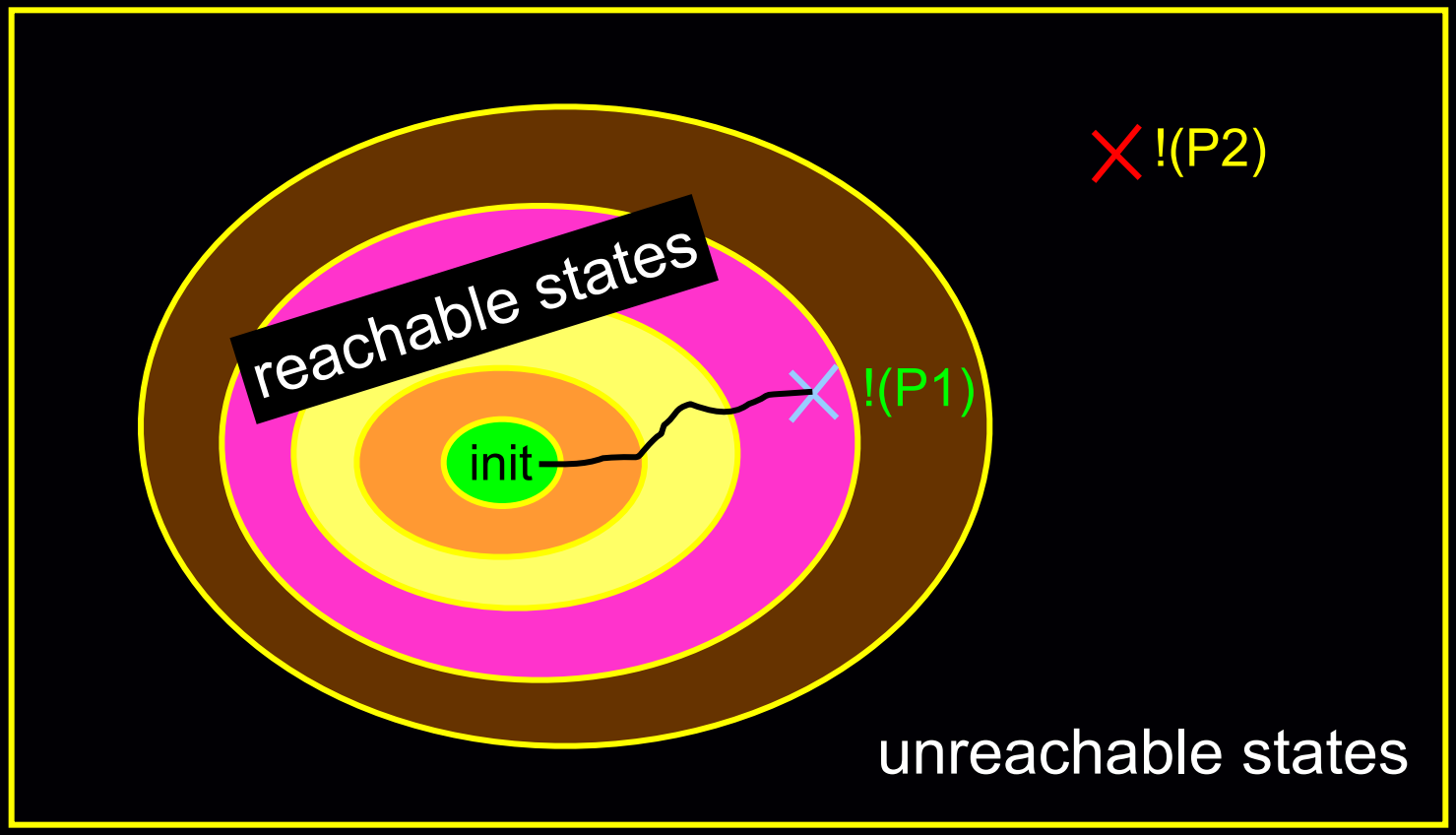


Set of Reachable States



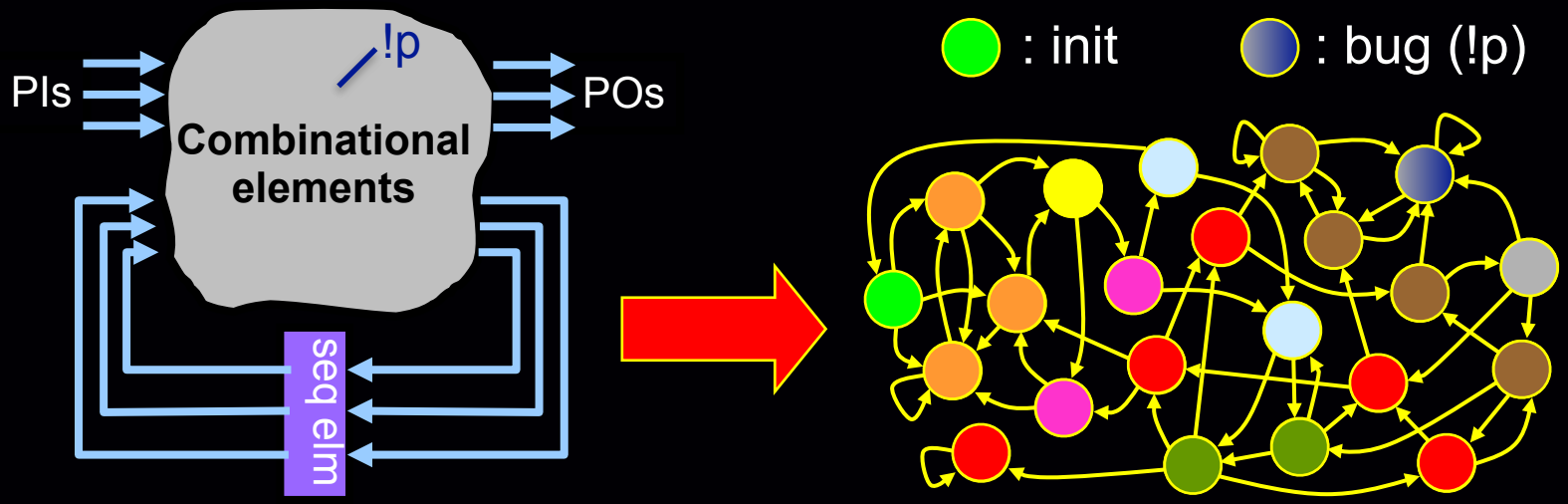
Boolean State Space

Boolean space of n state variables (2^n)



➔ assert(p1) ≡ false; assert(p2) ≡ true

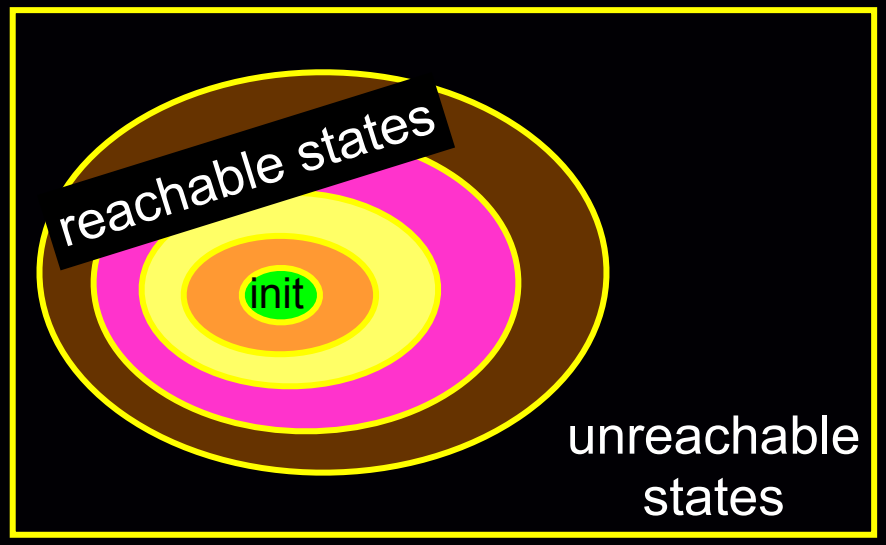
So, how does formal verification "prove" an assertion?



- (FACT) The numbers of states and traces are exponential
- Need an efficient "data structure" to:
  - Record the set of states (for each time step)
  - Perform the checking on whether the bug (!p) is reached
  - Or, able to show that the bug (!p) is unreachable

Encoding of the State Space

Boolean space of n state variables (2^n)



Conceptually, if we treat the whole design as a BIG finite state machine, we can concatenate all the registers:

```
reg  [a:0] aaa;
reg  [b:0] bbb;
...
```

into a BIG register:

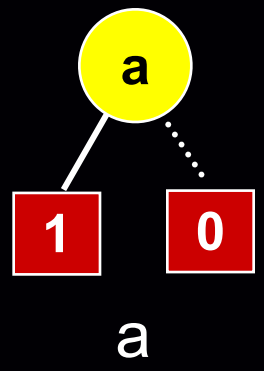
```
reg  [n-1] state;
```

and encode the global state as an n-bit Boolean string.

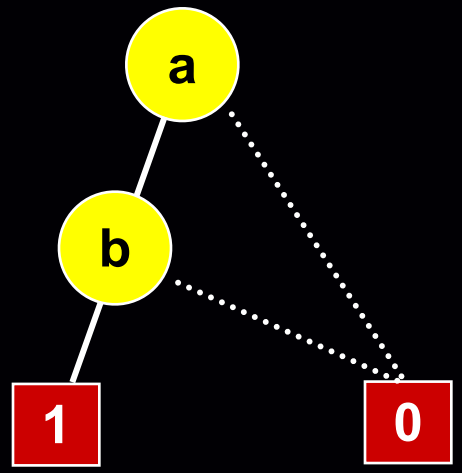
- Essentially, we can define a Boolean function **f** to indicate whether a state is in the set of reachable states
- How to efficiently record the set of reachable states?
  - An array of bit strings?
  - Hash table?
  - A binary tree?

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

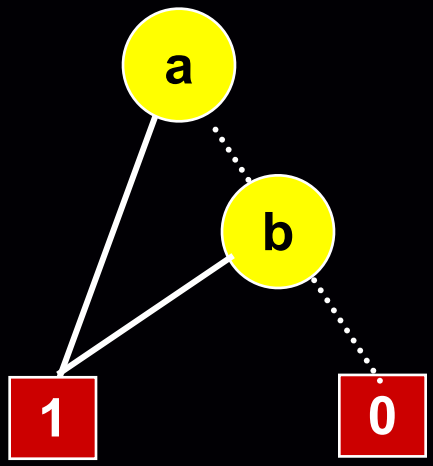
Recall: Binary Decision Diagram (BDD)



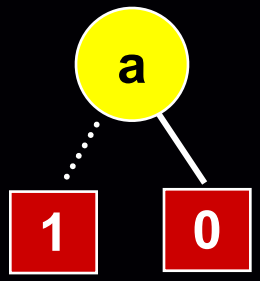
a



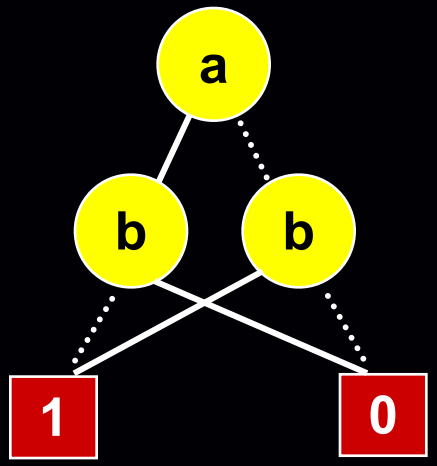
a && b



a || b

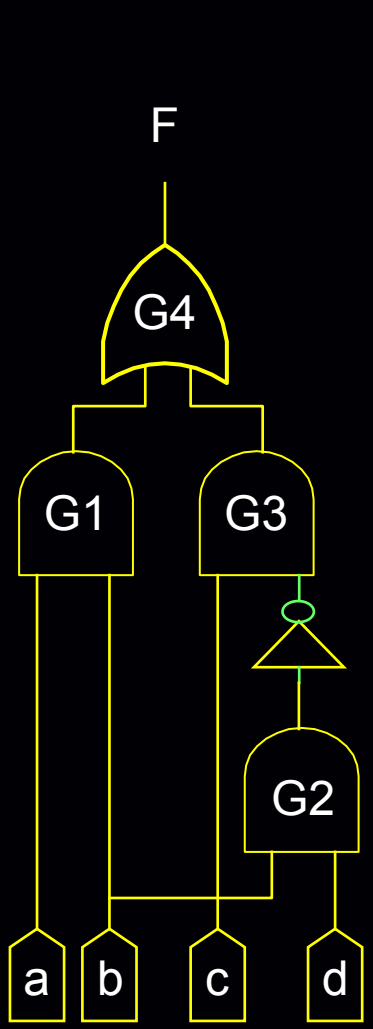


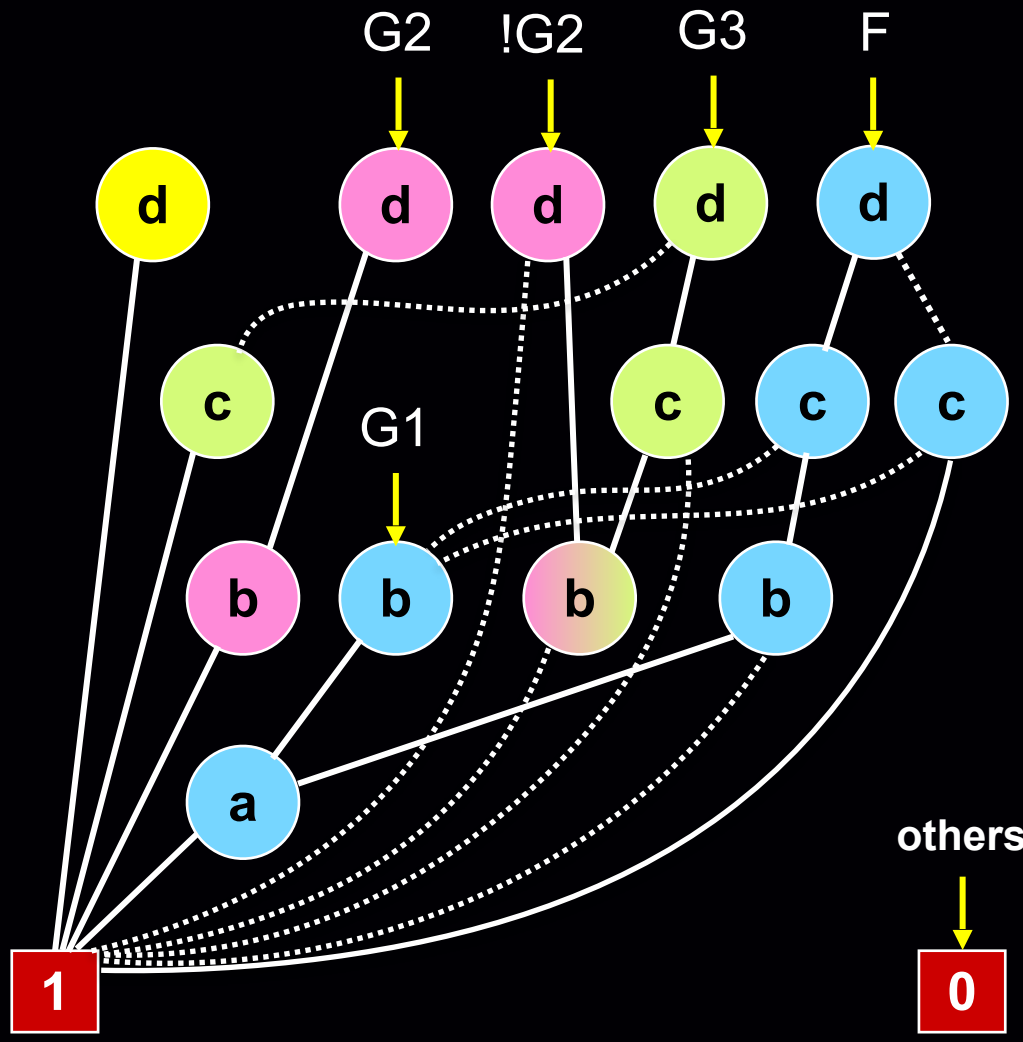
!a

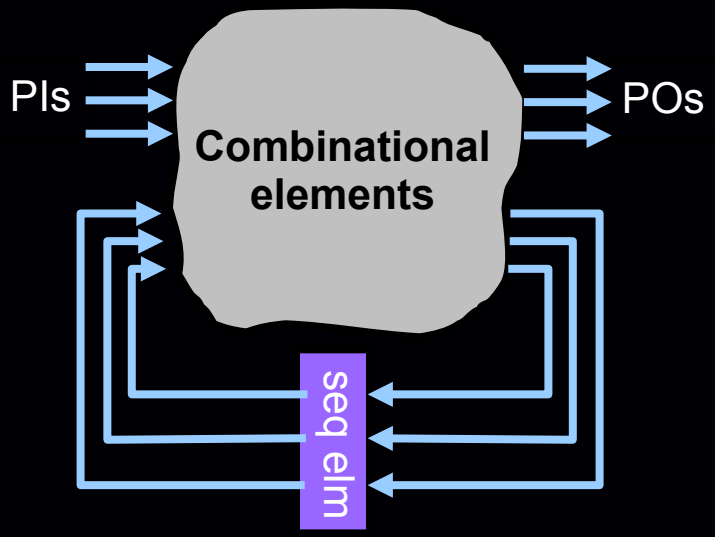


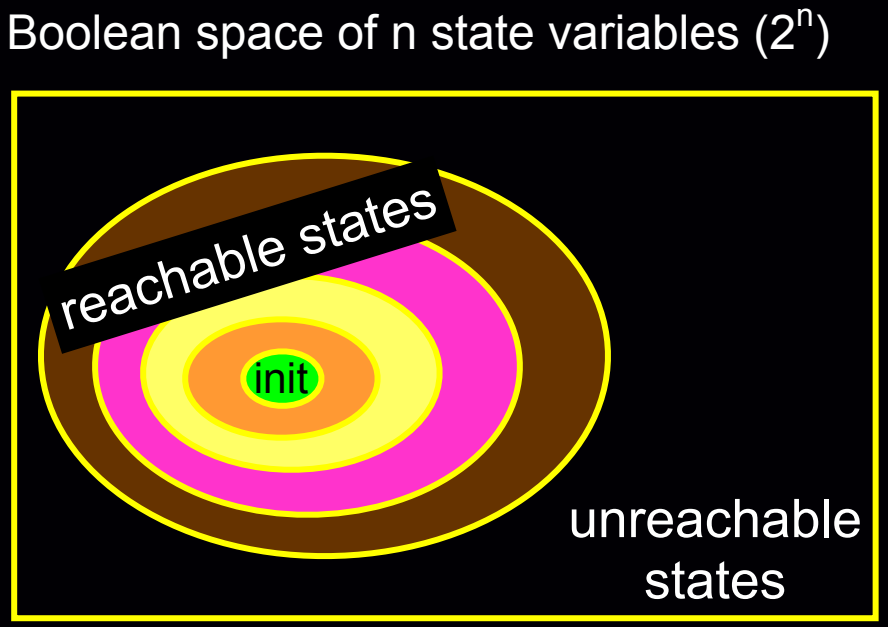
a ⊕ b

BDD to Represent a Boolean circuit







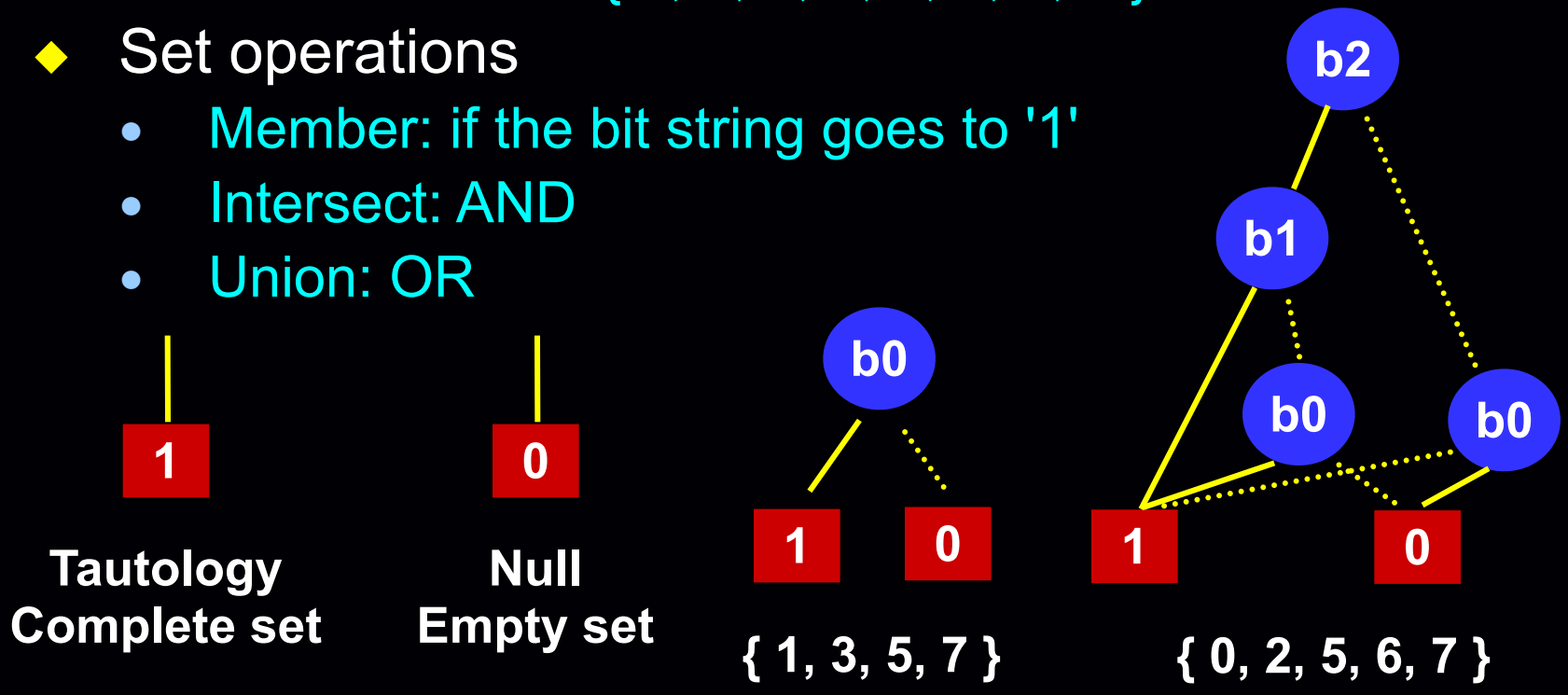


We have learnt that BDD can efficiently represent a Boolean function/circuit.

How to use BDD to represent the set of researchable states?

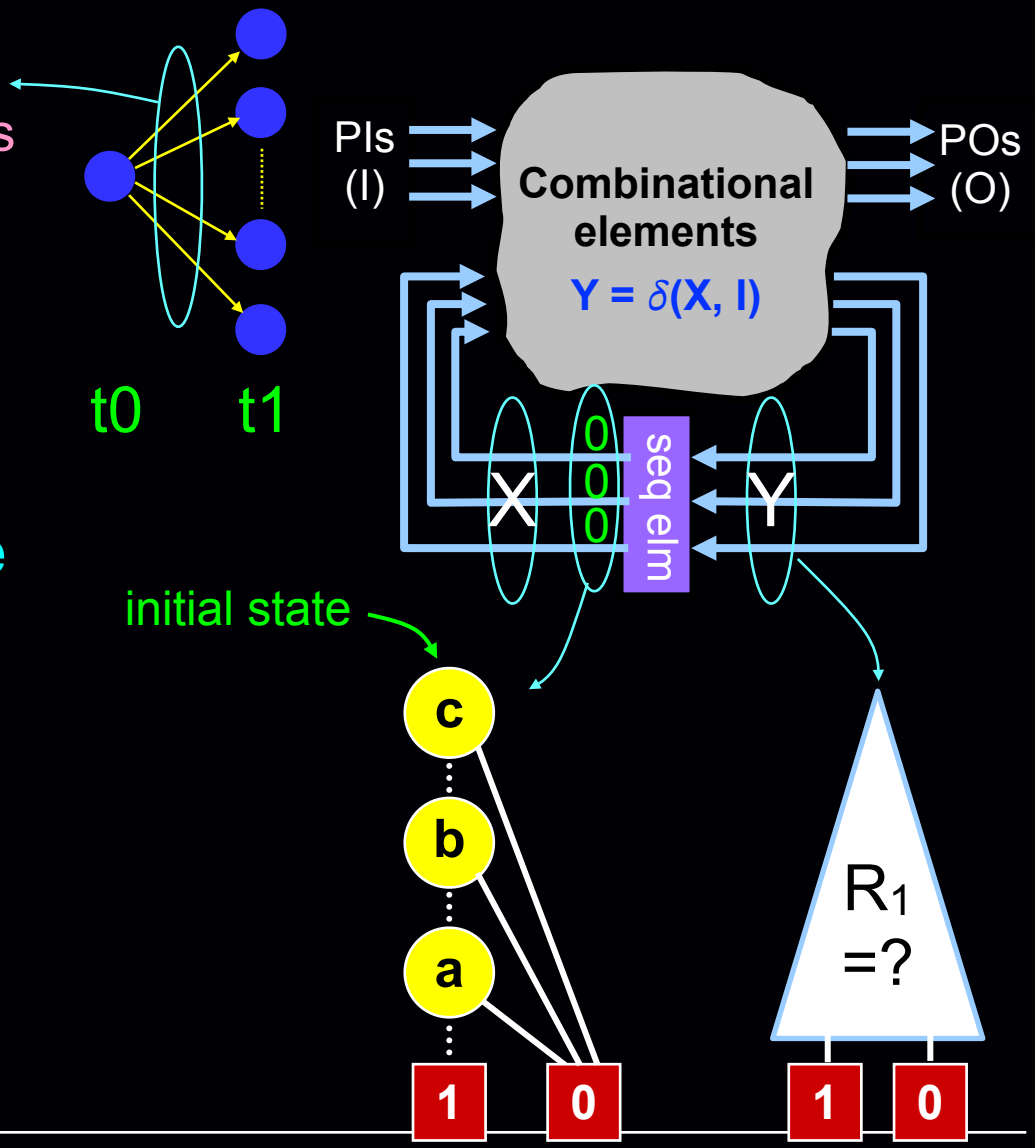
BDD to Represent a Set

- Other than Boolean function, BDD can also represent sets over Boolean variables
  - e.g. Use 3-variable BDDs to represent any subset of the numbers in { 0, 1, 2, 3, 4, 5, 6, 7 }
- Set operations
  - Member: if the bit string goes to '1'
  - Intersect: AND
  - Union: OR



Computing the Set of Reachable States

- Given initial state R_0 = (0, 0, 0) and BDD(R_0)
  - How to compute the set of next reachable states R_1 in t1 and represent it in BDD?
  - How does it relate to "Y=δ(X, I)"?



For BDD to represent
the set of reachable states R(S),

we need to introduce the concept of
"(state) transition relationship (TR)",
and how to use BDD to represent TR.

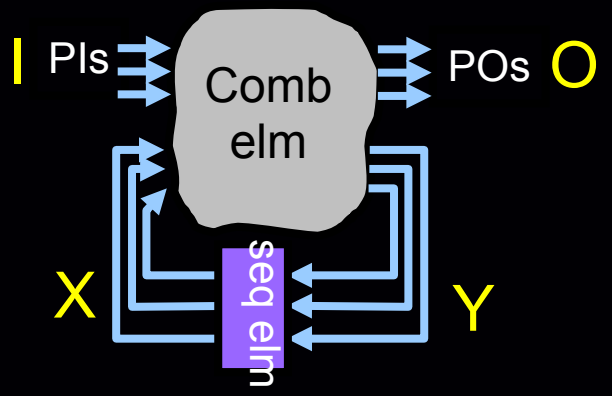
From Transition Functions to Transition Relationship, to Set of Reachable States

- Assume current set of reachable states are represented as R(X) // Init states = ?
- We would like to compute next set of reachable states as R'(Y)
  - What's the relationship between R(X) and R'(Y)?
  - What's the relationship between X and Y?
- Transition Functions: Y = δ(X, I)
  - For each state variable, $y_i = \delta_i(X, I)$

  ➔ We can build BDDs for y_i's
- Define: Transition Relationship: TR(Y, X, I)
  - For each legal transition (X, I) →Y, the relation TR(Y, X, I) = 1

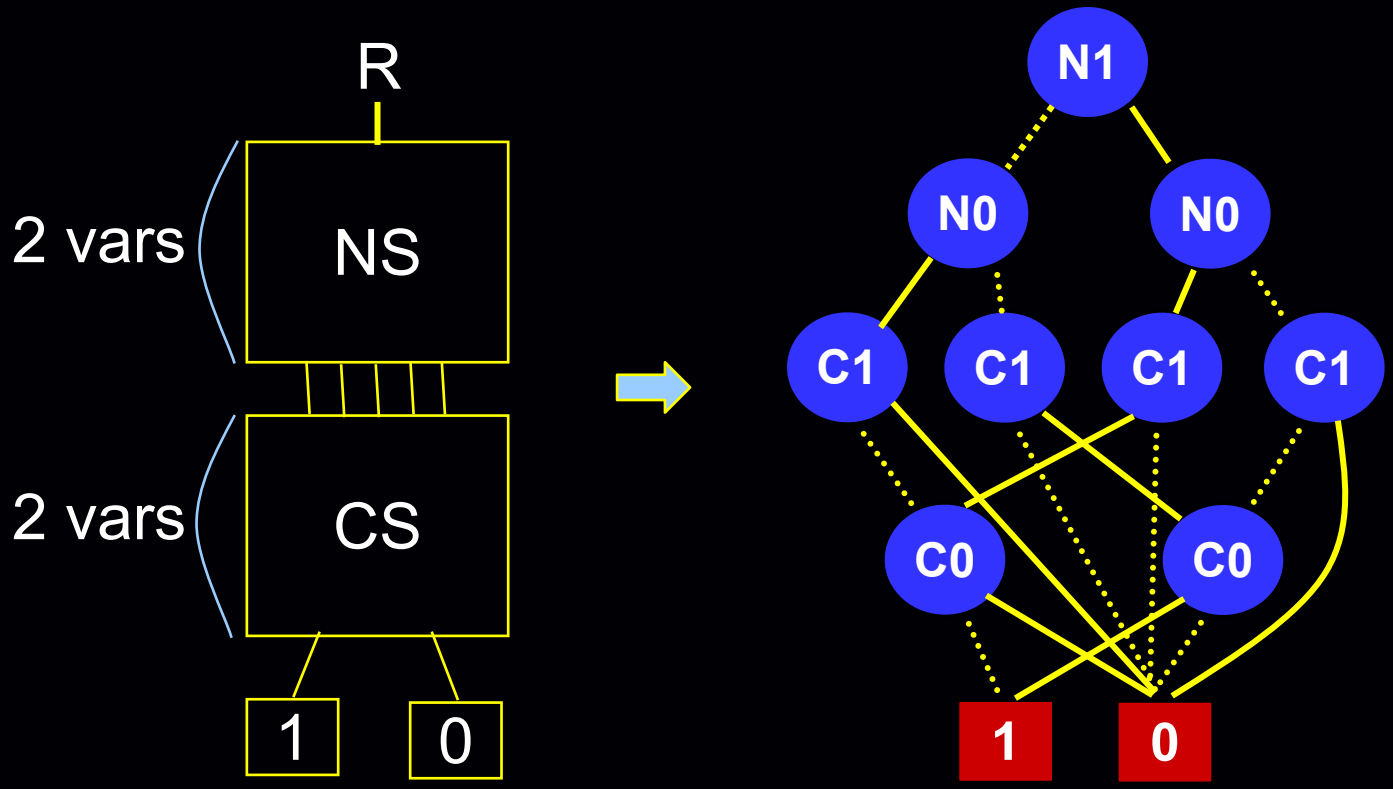
  ➔ How to build the BDD for TR(Y, X, I)?

  ➔ If we have R(X), how to compute the image on Y?? and represent it as BDD: R'(Y) ??



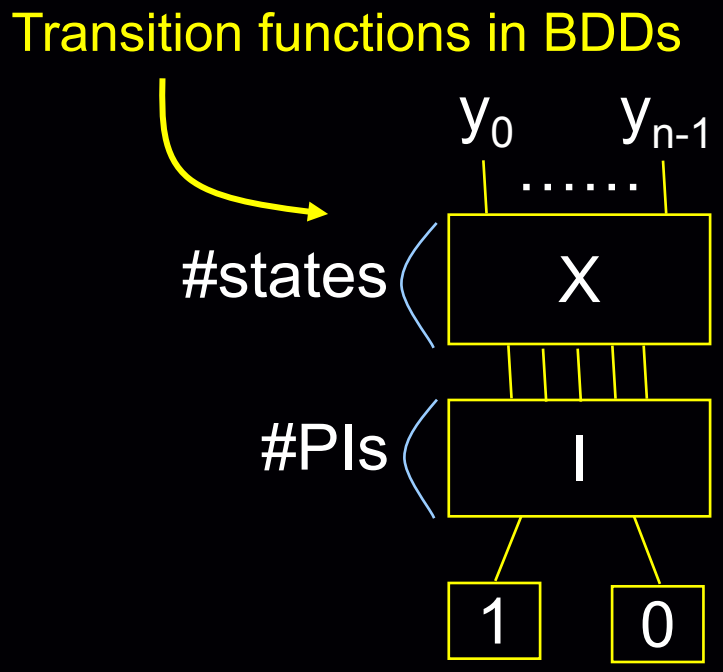
Note: BDD can also represent relations

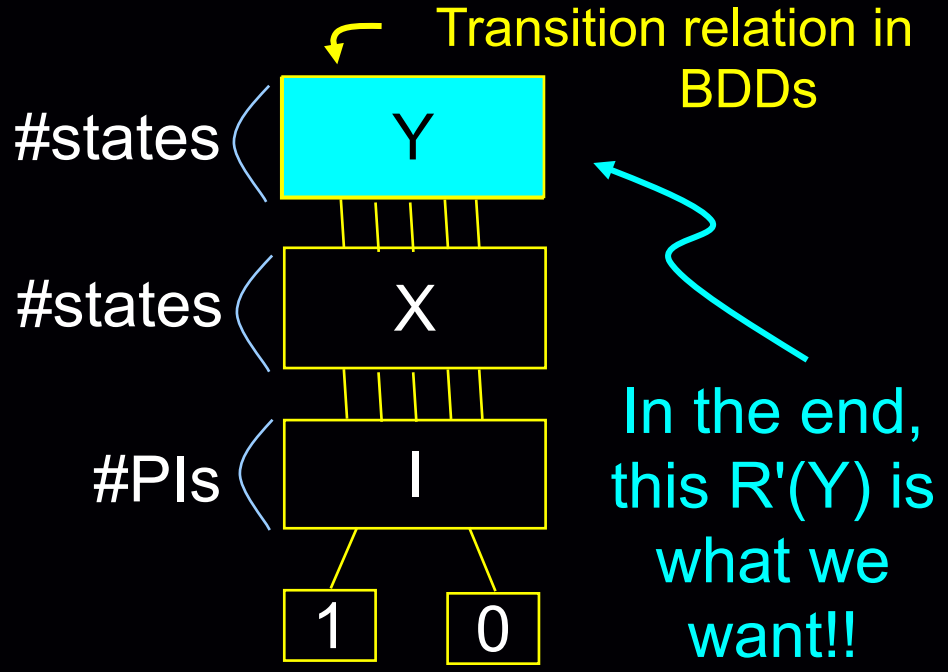
- e.g. 2-bit ring counter: NS = (CS + 1) % 4
  R: { (0→1), (1→2), (2→3), (3→0) }



From Transition Functions to Transition Relationship, to Set of Reachable States

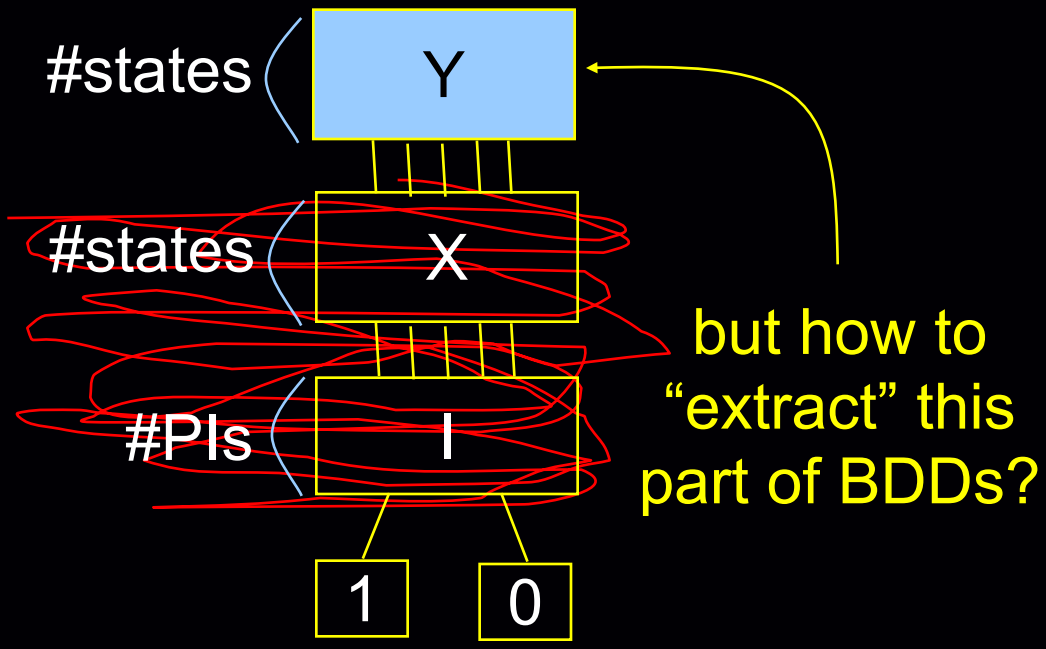
- Given Y = δ(X, I)...
- Compute TR(Y, X, I)





$$TR(Y,X,I) = \prod_i (\overline{y_i \oplus \delta_i(X,I)})$$

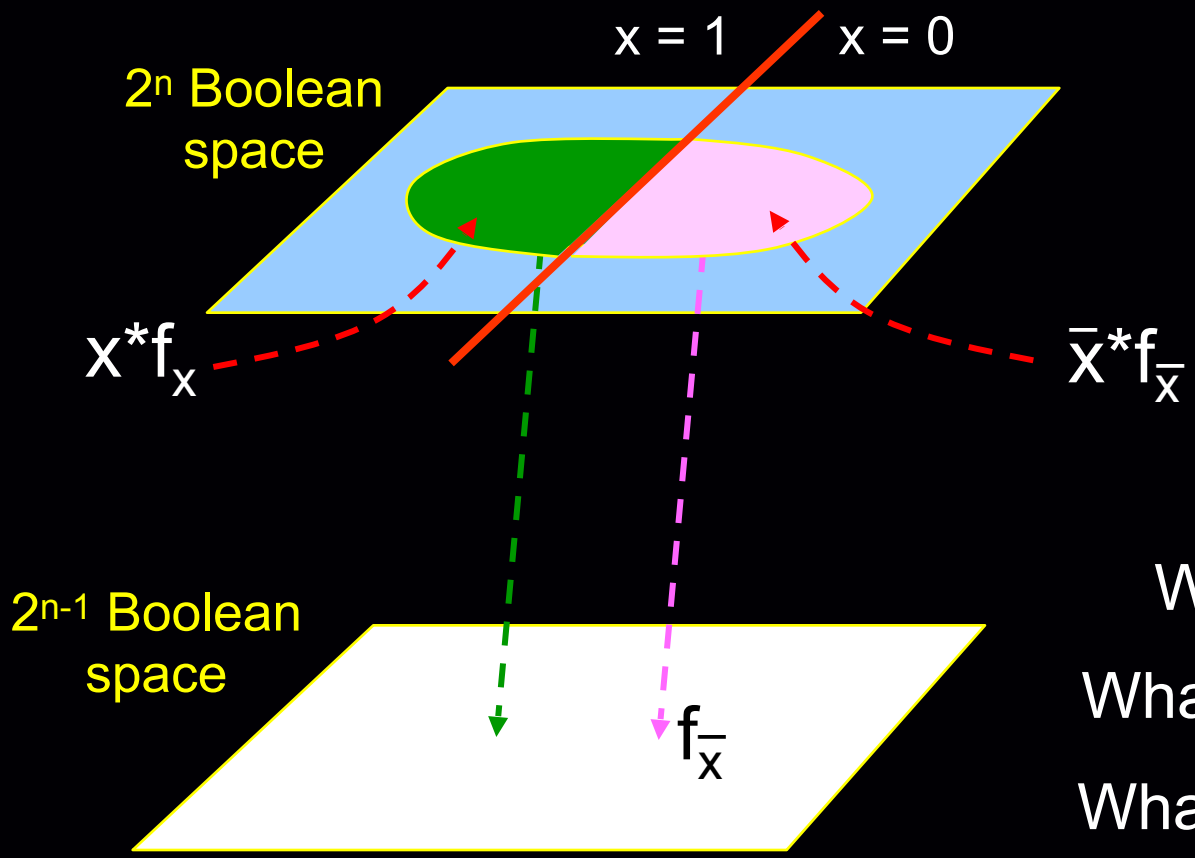
The Question is…



Erase BDD variables ➔ Existential Quantification

Remember: co-factors...

- Fallacy: $f_x \wedge f_{\bar{x}} = \varnothing$



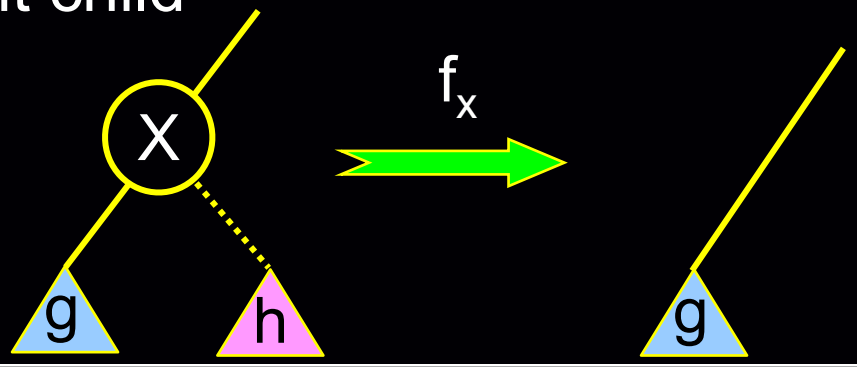
What is $x*f_x$?

What is $(f_x \wedge f_{\bar{x}})$?

What is $(f_x \oplus f_{\bar{x}})$?

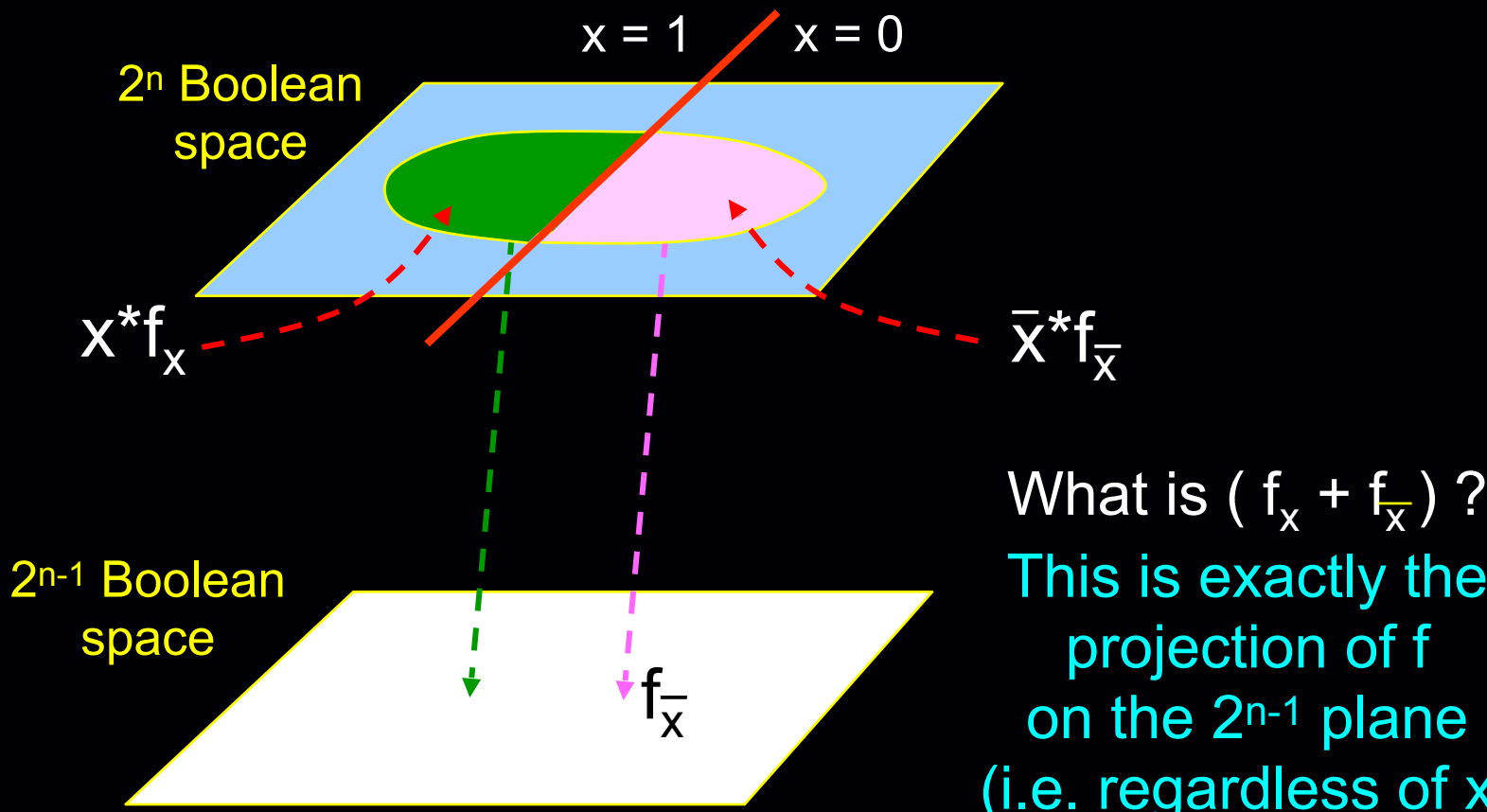
Computing Cofactors on BDD

- Given a function f
  ➔ find its positive/negative cofactor f_x / $f_{\overline{x}}$
  - e.g. Let f = a $\overline{c}$ + b c
    ➔ f_c = b
    ➔ $f_{\overline{c}}$ = a
    ➔ $f_{\overline{a}}$ = b c
- If x is top variable
  ➔ cofactor = left or right child
- Otherwise,
  for each node in level X



Existential Quantification

- $\exists x.f = f_x + f_{\bar{x}}$ (⬅ What does this mean?)

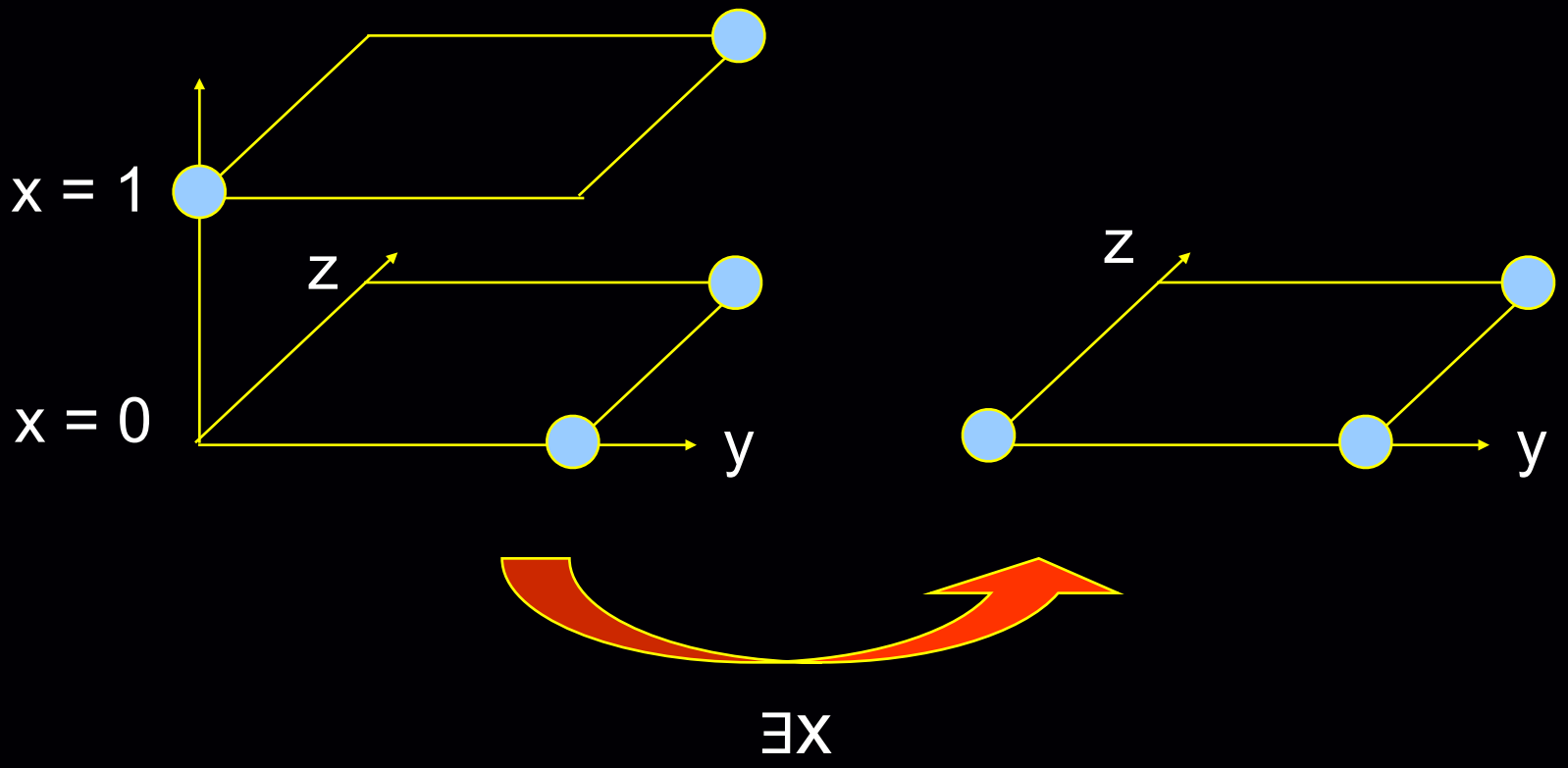


What is ($f_x + f_{\bar{x}}$) ?

This is exactly the projection of f on the 2^{n-1} plane (i.e. regardless of x)

Existential Quantification

- $\exists x.f = f_x + f_{\bar{x}}$ (🡸 What does this mean?)



Existential Quantification on BDD

- $\exists x.f = f_x + f_{\bar{x}}$ (🡸 How to perform it on BDD?)
- If x is top variable

  ➔ Perform an “OR” on its cofactors
- If x is bottom variable

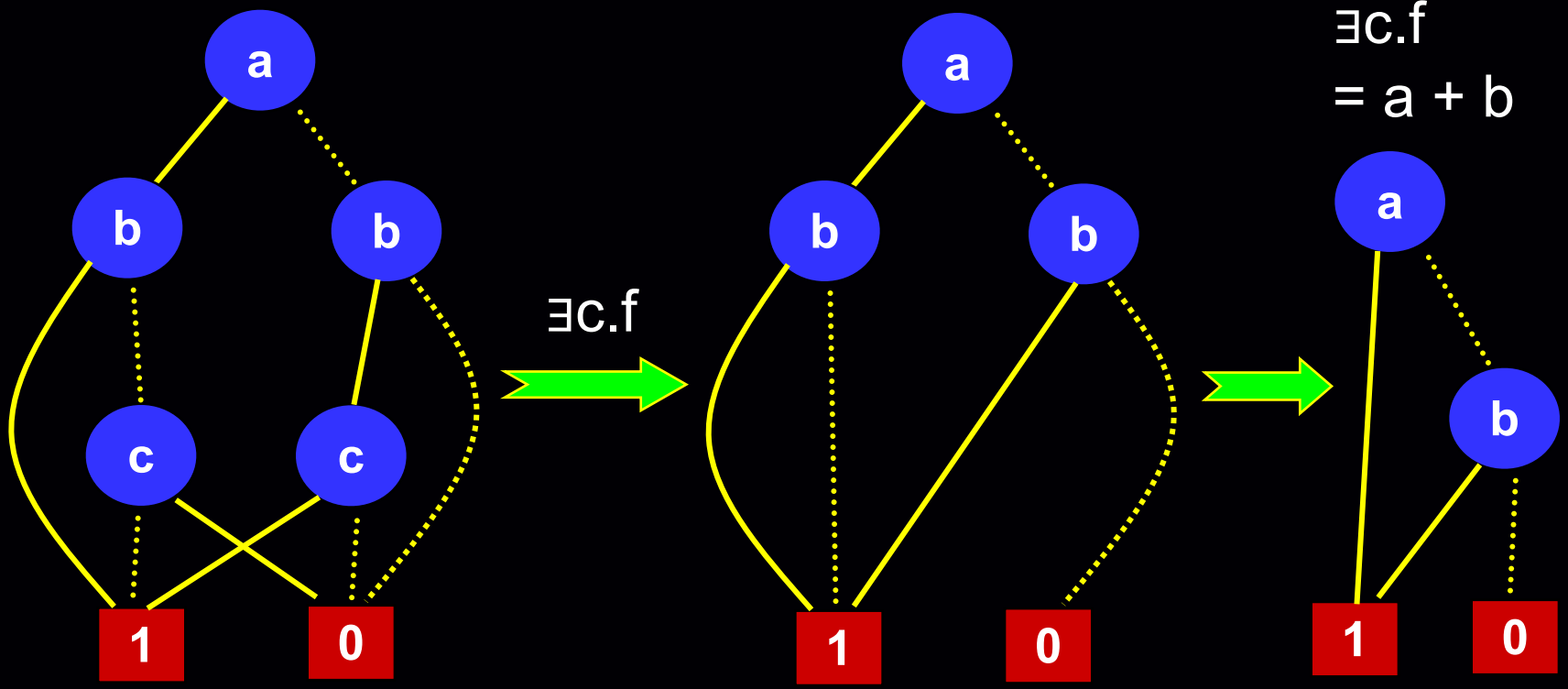
  ➔ Replace non-zero nodes with ‘1’ (why?)
- If x is middle variable

  ➔ ???

Which one is better??

Existential Quantification

- $\exists x.f = f_x + f_{\bar{x}}$
  - e.g. $f = a\,b + a\,\bar{b}\,\bar{c} + \bar{a}\,b\,c$



Cofactors, Boolean Difference, Existential Quantification

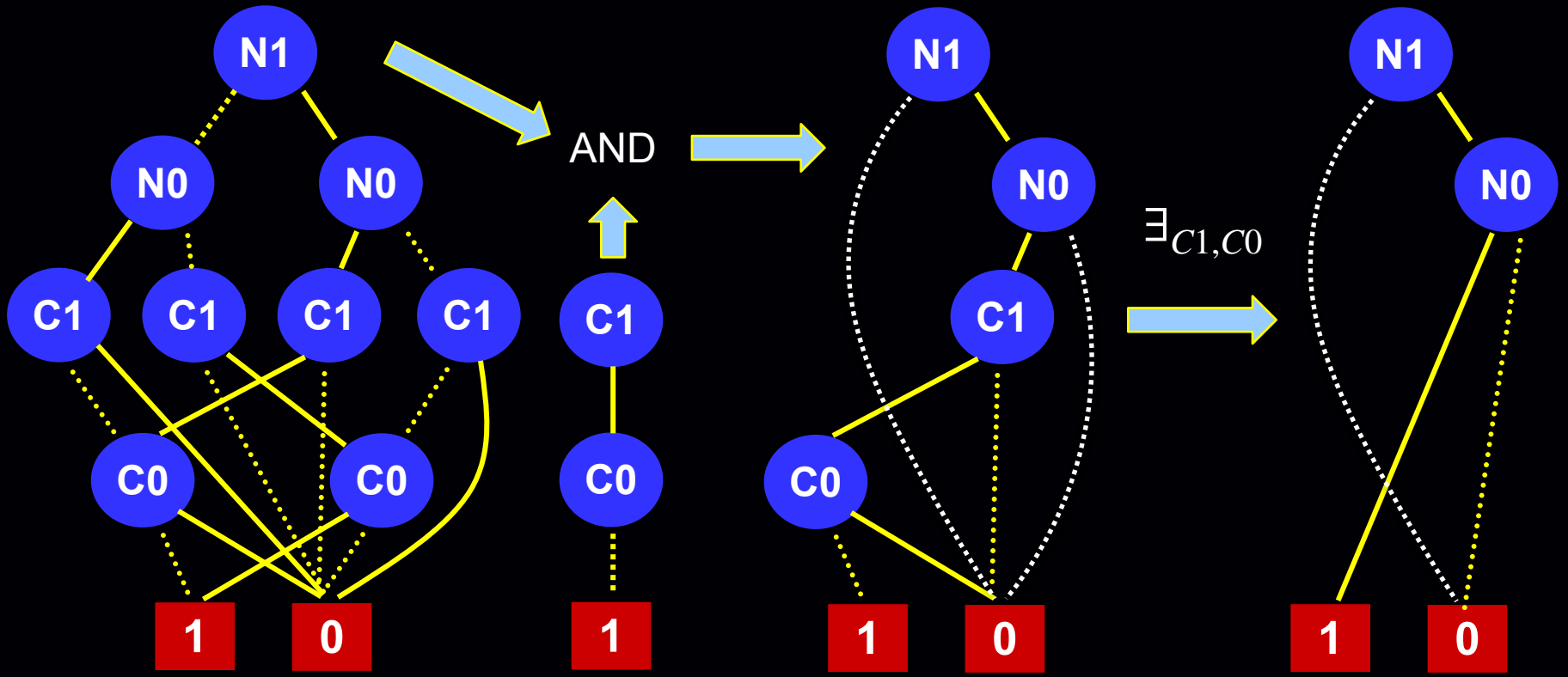
- Let $F = x \cdot A + \overline{x} \cdot B + C$

1. Cofactors
   - $F_x = A + C$
   - $F_{\overline{x}} = B + C$
2. Boolean difference
   - $F_d = (A \oplus B) \cdot \overline{C}$
3. Existential quantification
   - $\exists x.F = A + B + C$

- What if the formula is represented in PoS form?

Existential Quantification

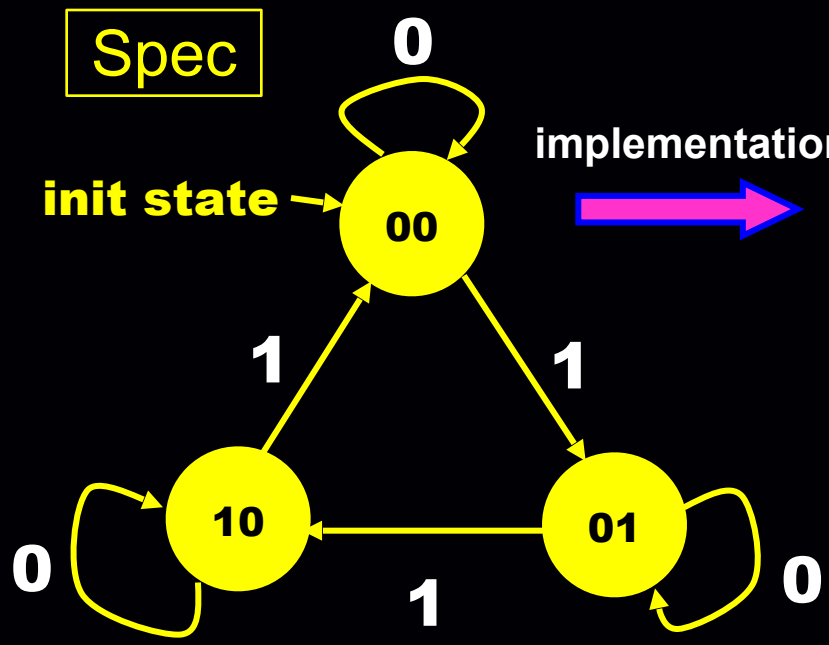
- For the 2-bit ring counter, how to compute the BDD for the next state of "CS = (1, 0)"?



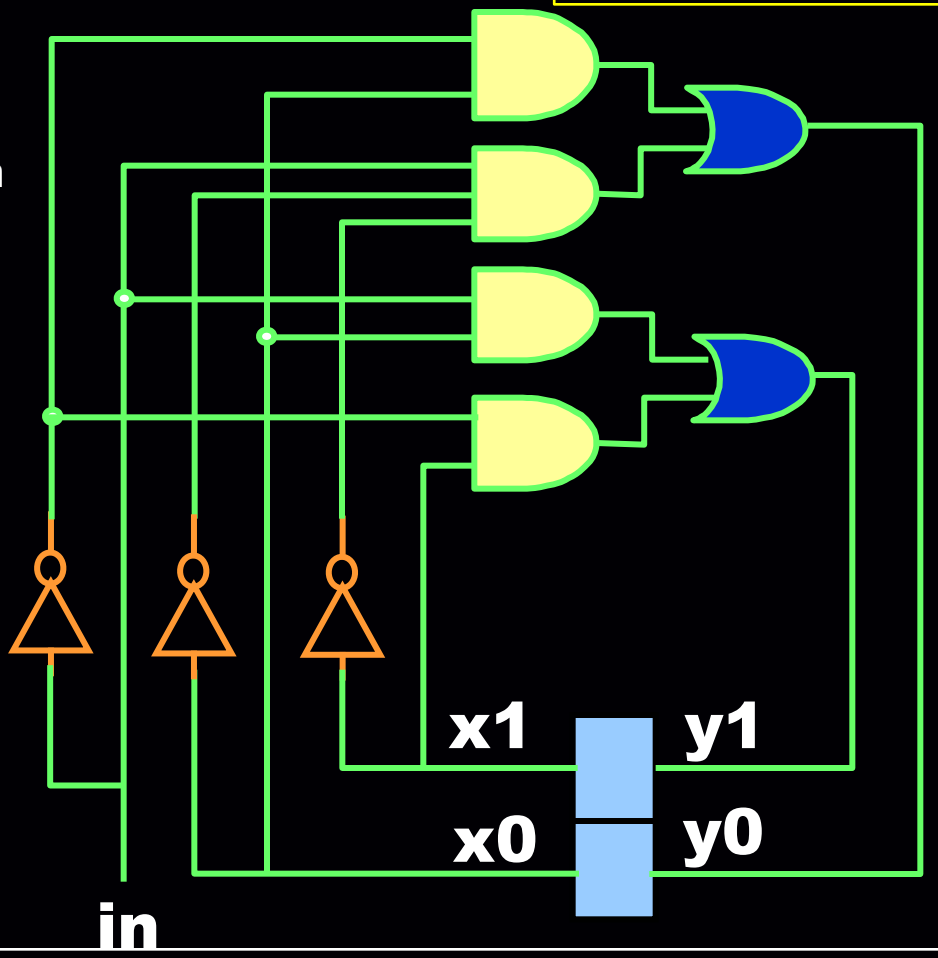
Example: Model Checking Using BDD

- Given a 2-bit counter as follows:

Is this implementation correct??



implementation



Assert: P

State <= 2

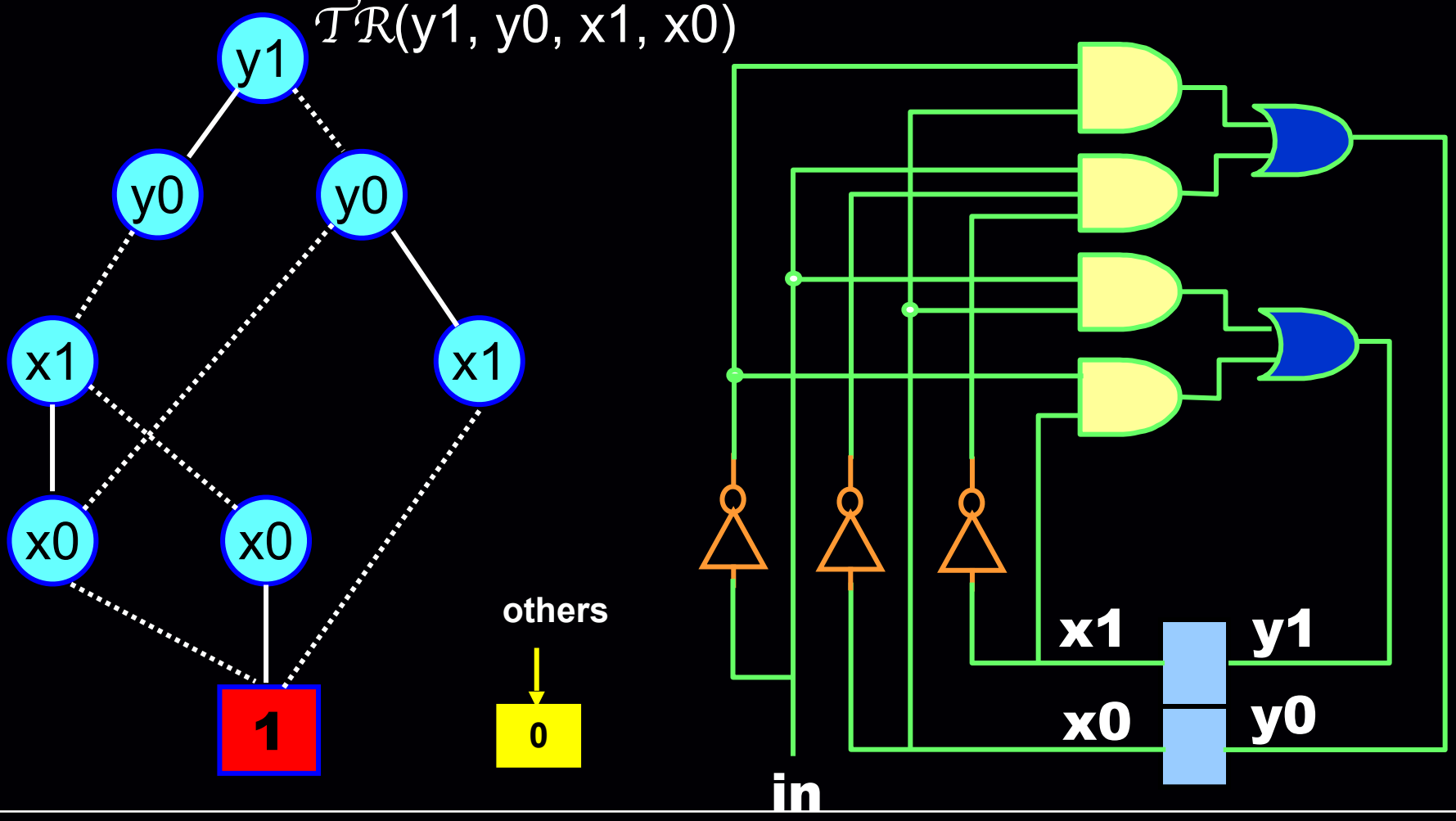
(i.e. {y1, y0} <= 2)

Model Checking by BDD

1. Determine the variable ordering
   - e.g. y1, y0, x1, x0
2. Compute state transition relationship ($\mathcal{TR}$) of current states and next states
   - $\mathcal{TR}$(y1, y0, x1, x0)
3. Starting from initial state R = $S_0(X)$ = { (x1, x0) = 00 }, compute the set of reachable states at time i
   - R += $S_i(X)$ until R_i = R;
   - Otherwise, R ← R_i
4. Check if !P (i.e. (y1, y0) = 11) intersects with the set of reachable states
   - Check (R && !P)
   - If yes (intersects), property is false
     else go to 3

Transition Relationship in BDD

- **We first build TR from the circuit netlist as follows --**



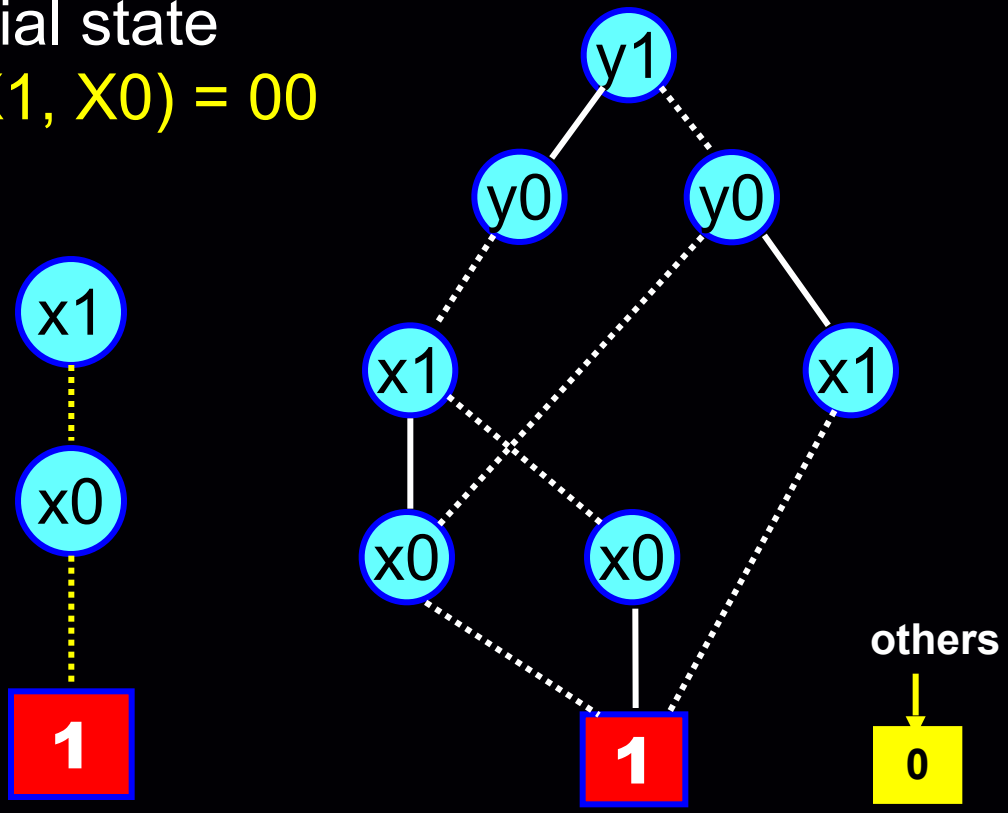
Compute Set of Reachable States (Time 1)

Transition Relationship

$\mathcal{TR}$(y1, y0, x1, x0)

initial state

R_0 : (X1, X0) = 00



What's the set of reachable state S1 for time 1?

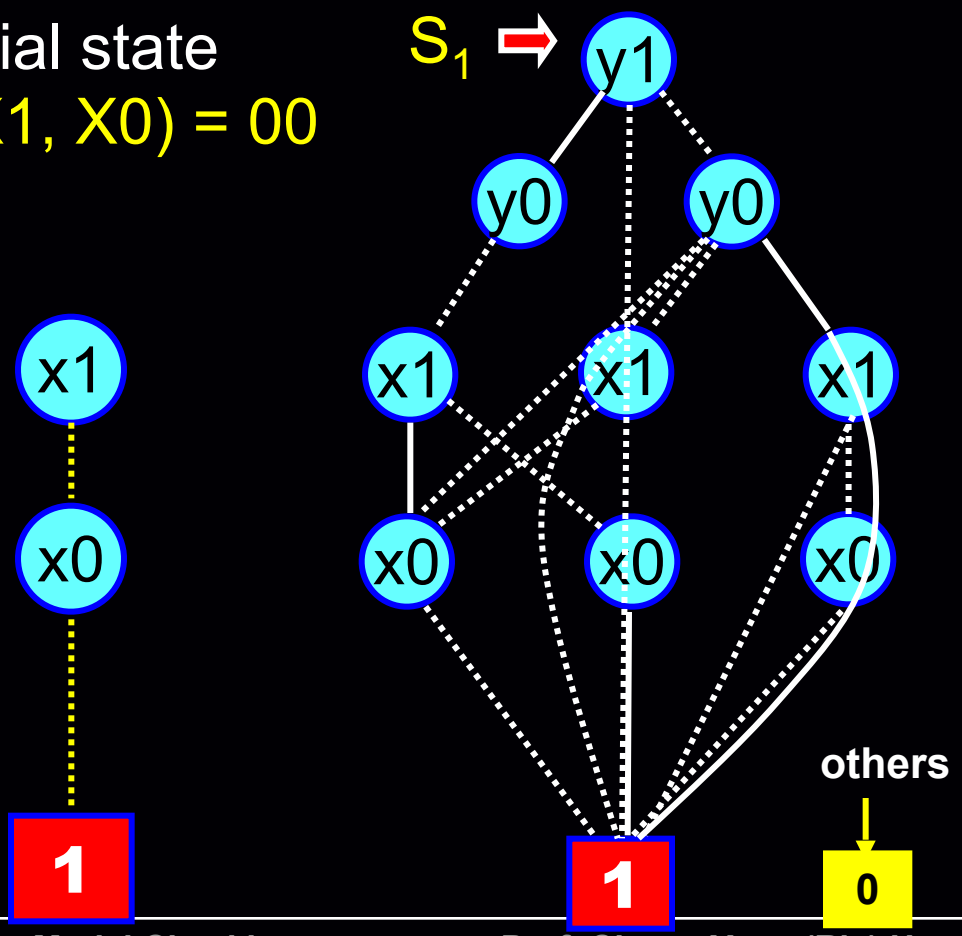
Compute Set of Reachable States (Time 1)

Transition Relationship

$\mathcal{TR}$(y1, y0, x1, x0)

initial state

R_0 : (X1, X0) = 00



To compute the set of reachable states in the next cycle ---

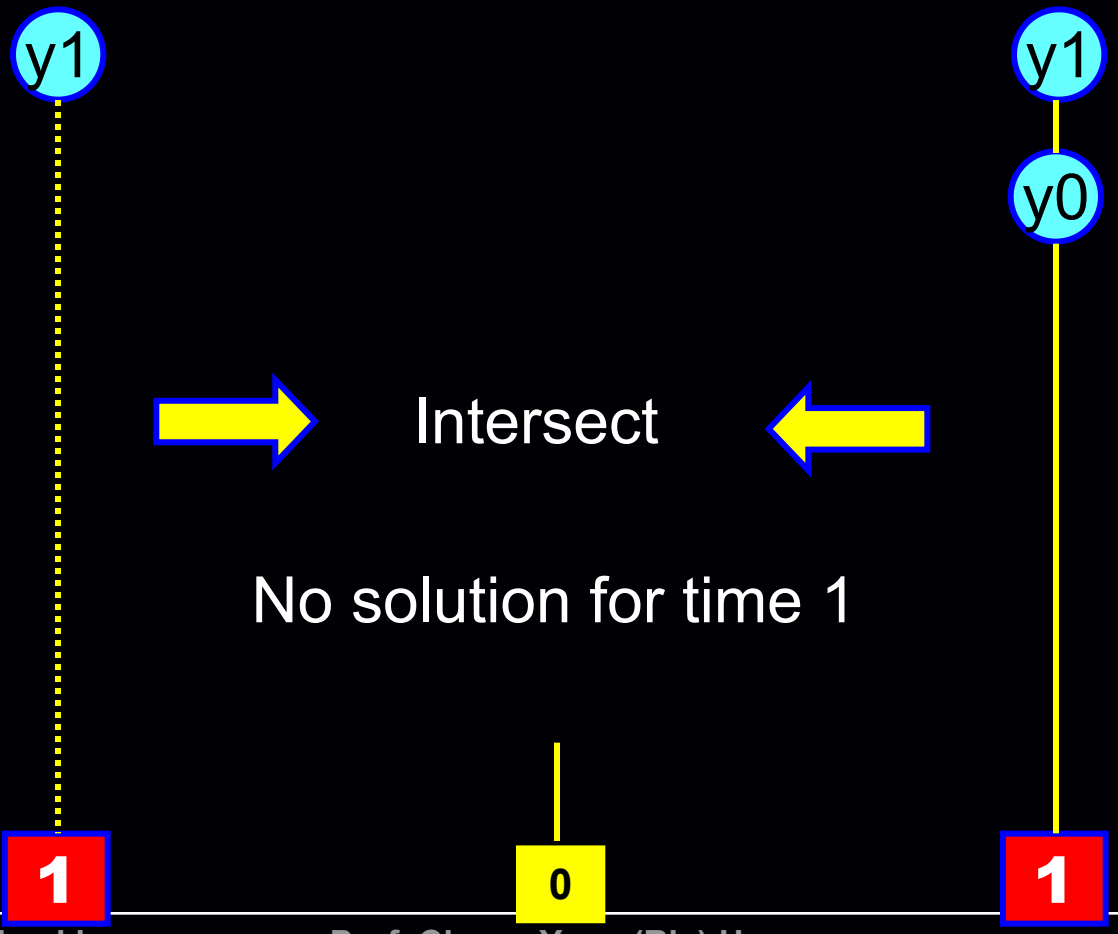
1. Build the TR BDD
2. Apply $\mathcal{TR}(X = R_0)$
3. Existential quantification

   $S_1 = \exists_{x1,x0} \mathcal{TR}(R0)$
4. Check intersection with !P

Check Intersection with !P

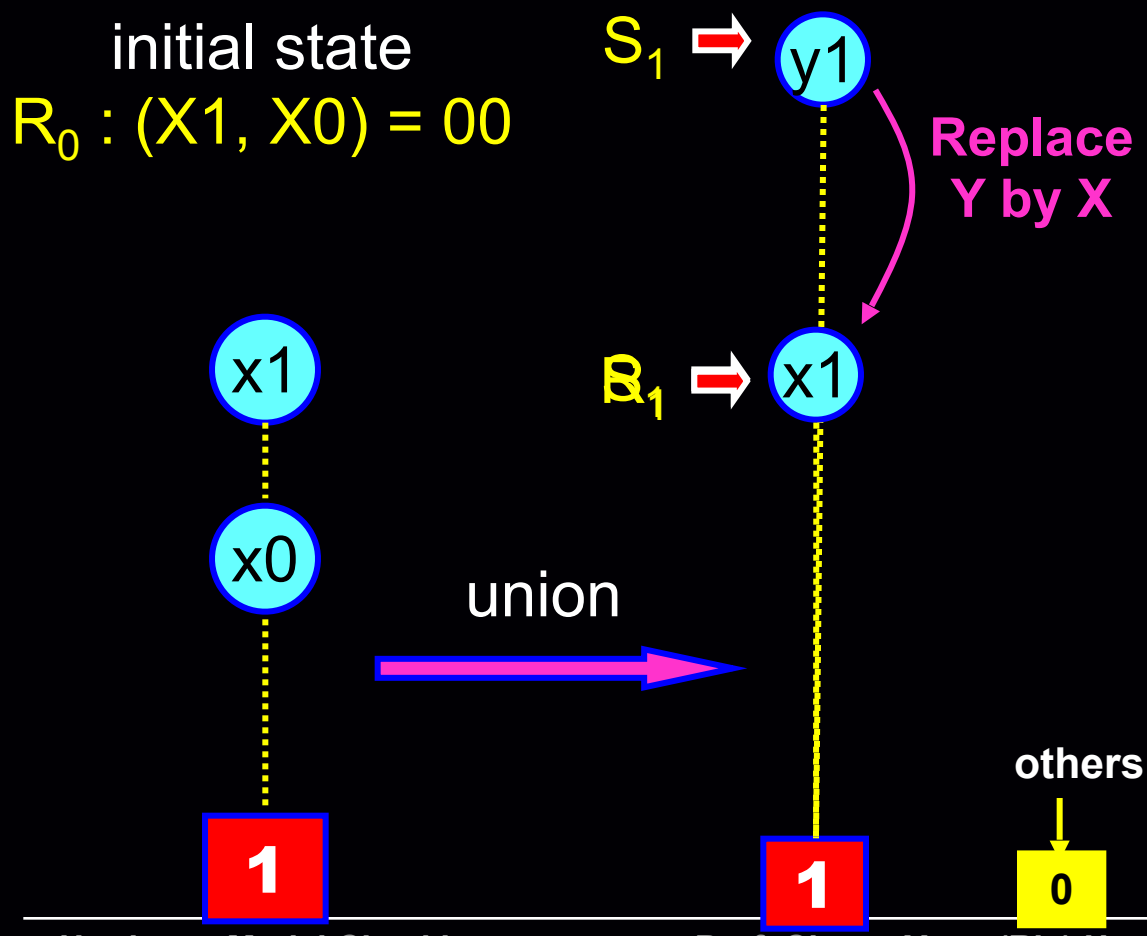
S_1 : (y1, y0) = 0 -
states reachable in time 1

P : (y1, y0) <= 2
➔!P : (y1, y0) = 11



Compute Set of Reachable States (Time 1)

Reachable states in time 1



To compute the set of reachable states in the next cycle ---

1. Build the TR BDD
2. Apply $\mathcal{TR}(R_0)$
3. Existential quantification
   $S_1 = \exists_{x1,x0} \mathcal{TR}(R0)$
4. Check intersection with !P
5. $R_1 = S_1(Y \rightarrow X) \cup R_0$

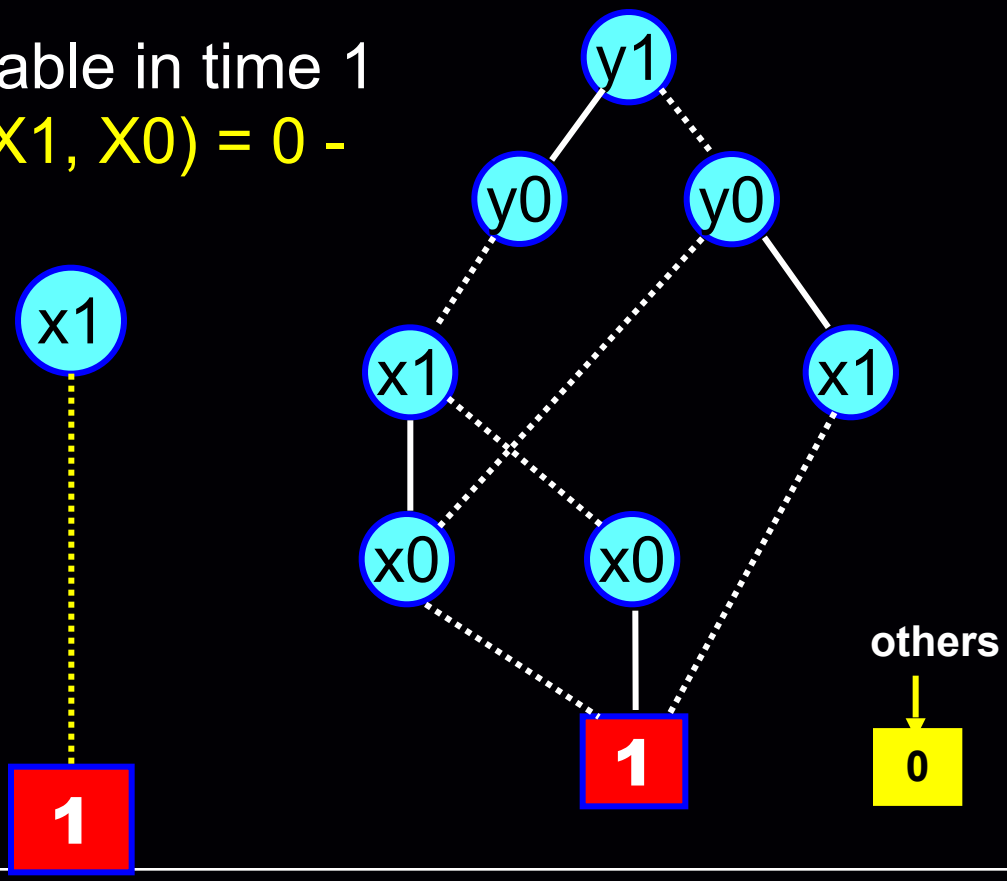
Compute Set of Reachable States (Time 2)

Transition Relationship

$\mathcal{TR}$(y1, y0, x1, x0)

Reachable in time 1

R_1 : (X1, X0) = 0 -

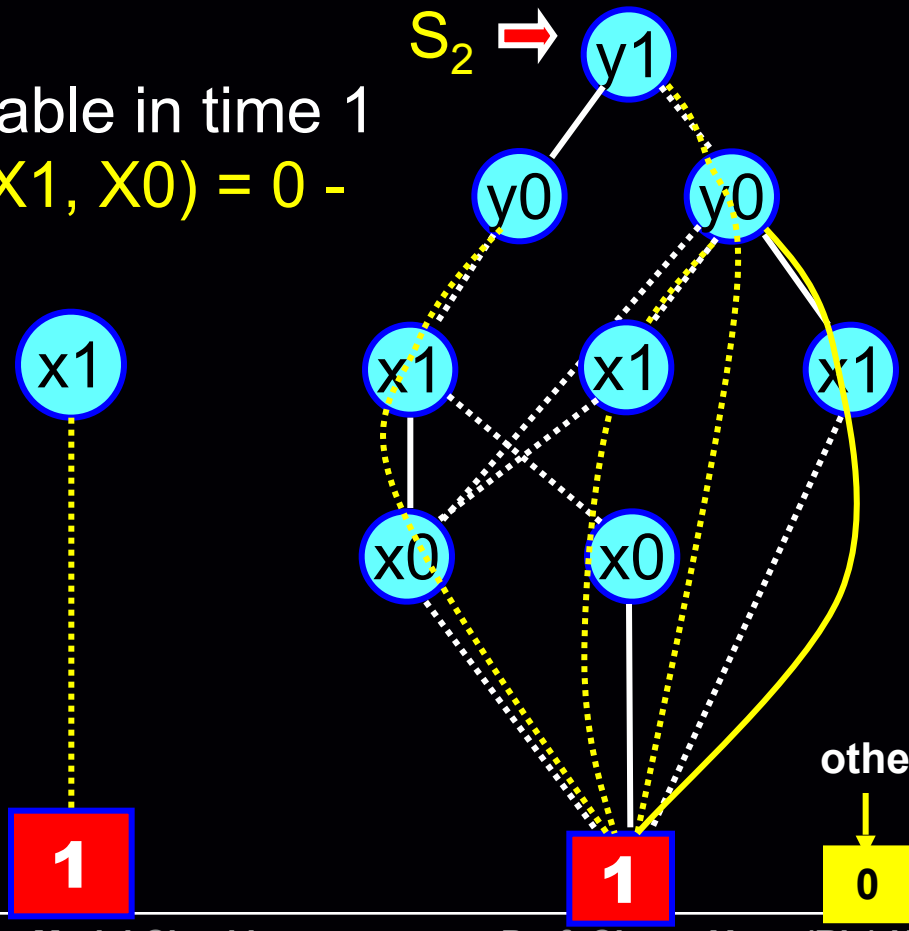


What's the set of reachable state S2 for time 2?

Compute Set of Reachable States (Time 2)

Transition Relationship

$\mathcal{TR}$(y1, y0, x1, x0)



Reachable in time 1

R_1 : (X1, X0) = 0 -

To compute the set of reachable states in the next cycle ---

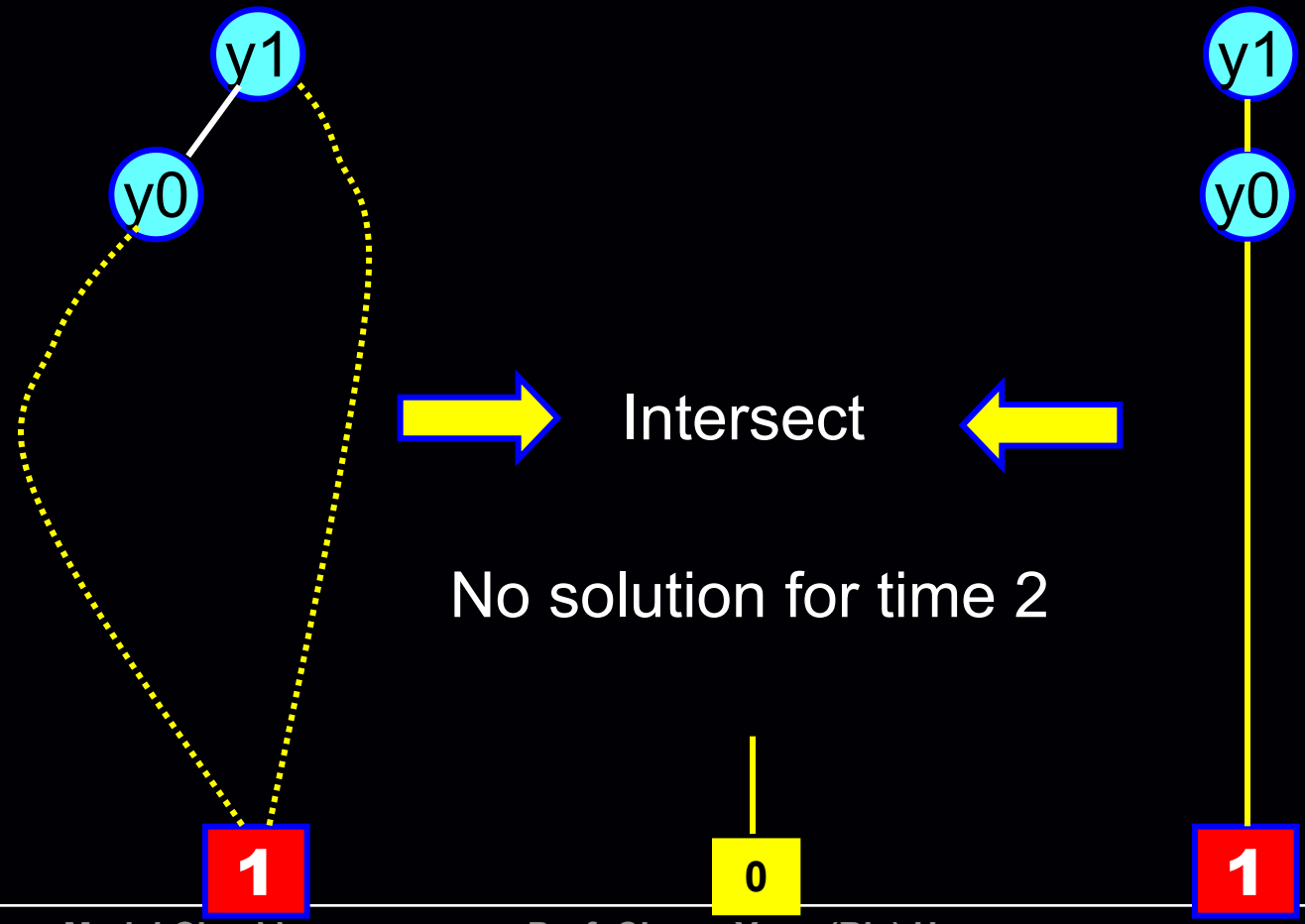
1. Apply TR(X = R_1)
2. Existential quantification

   $S_2 = \exists_{x1,x0} \mathcal{TR}(R_1)$
3. Check intersection with !P
4. $R_2 = S_2(Y \rightarrow X)\ U\ R_1$

Check Intersection

S_2 : (y1, y0) = { 10, 0 - }
states reachable in time 2

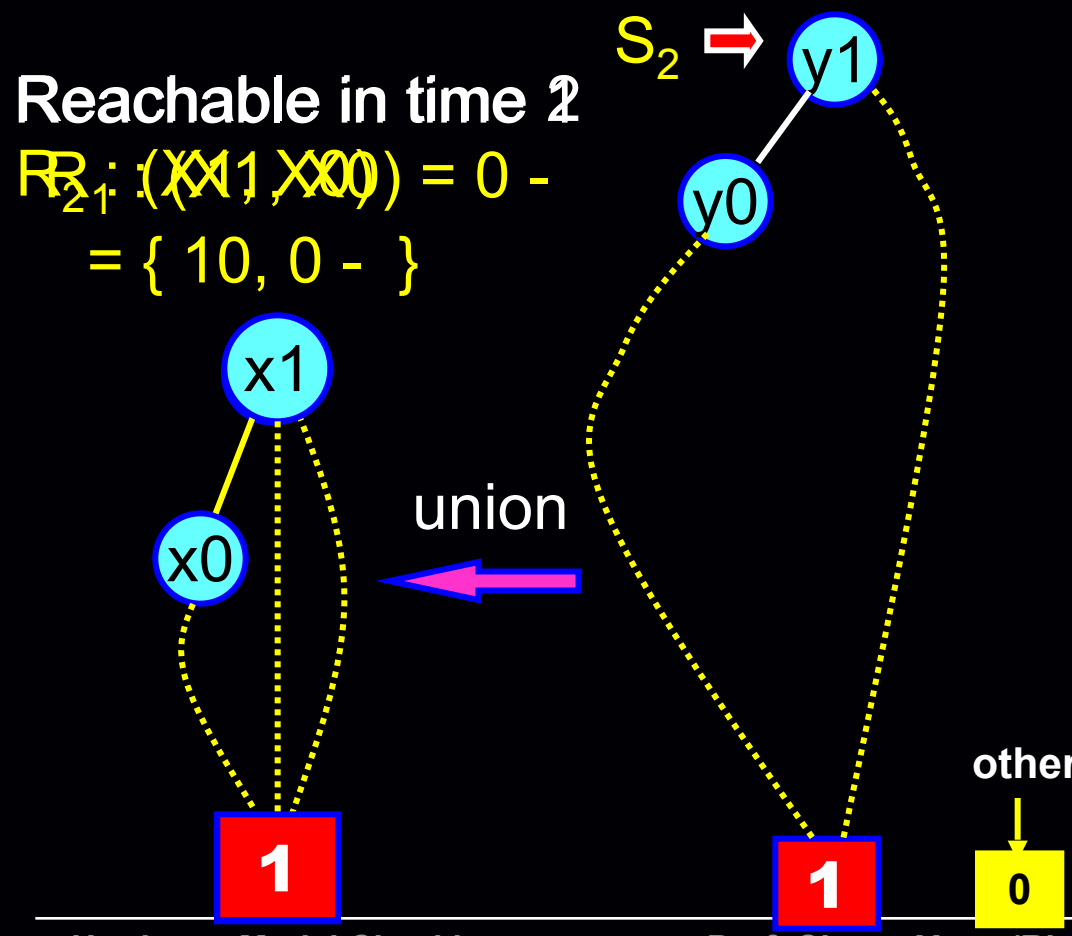
P : (y1, y0) <= 2
➔!P : (y1, y0) = 11



Compute Set of Reachable States (Time 2)

S_2 : (y1, y0) = { 10, 0 - }

states reachable in time 2

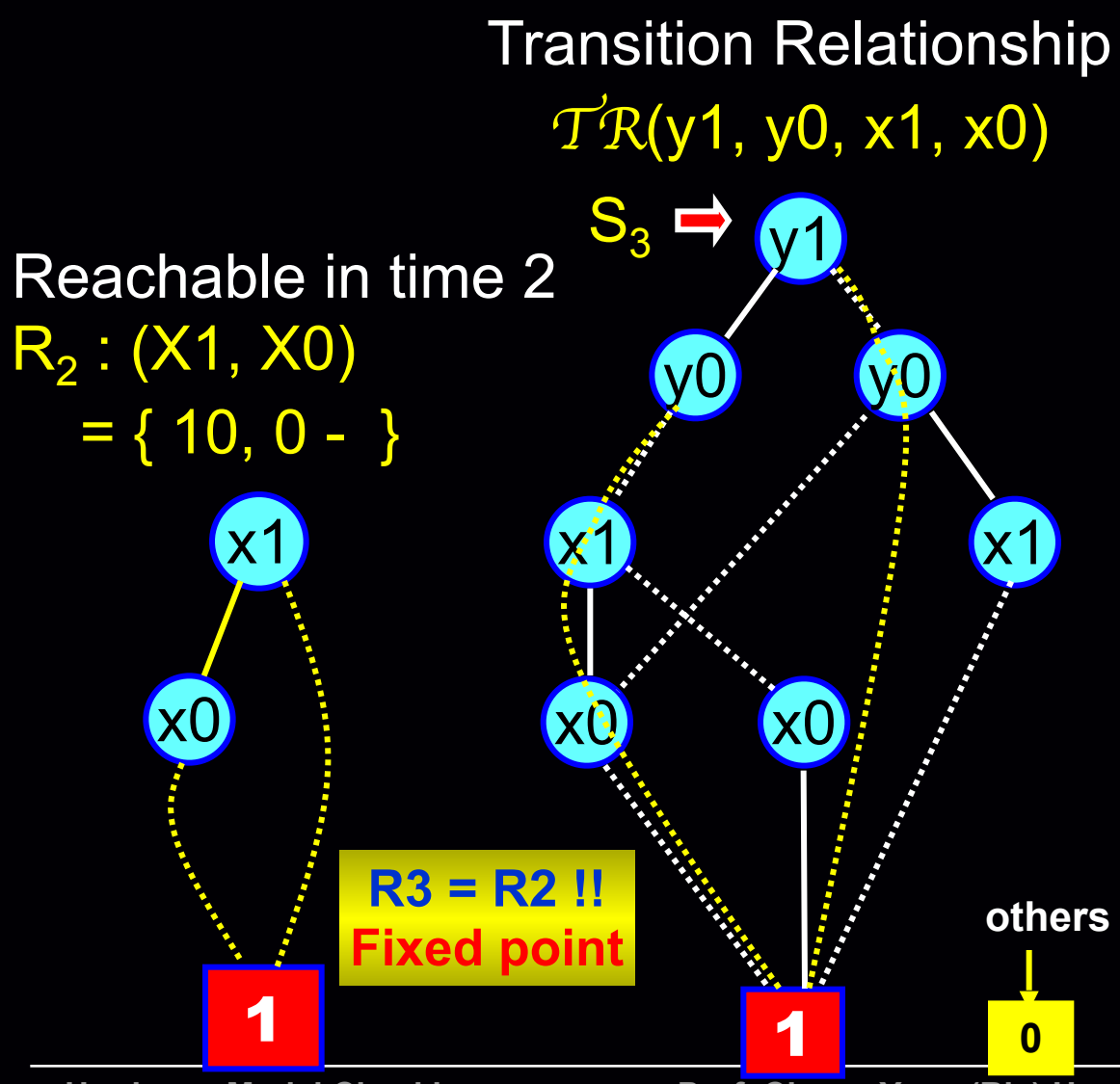


To compute the set of reachable states in the next cycle ---

1. Apply TR(R_1)
2. Existential quantification

   $S_2 = \exists_{x1,x0} \mathcal{TR}(R_1)$
3. Check intersection with !P
4. $R_2 = S_2(Y \rightarrow X)\ U\ R_1$

Compute Set of Reachable States (Time 3)



To compute the set of reachable states in the next cycle ---

1. Apply TR(R_2)
2. Existential quantification
   $S_3 = \exists_{x1,x0} \mathcal{TR}(R_2)$
3. Check intersection with !P
4. $R_3 = S_3(Y \rightarrow X) \cup R_2$

Fixed Point in State Reachability Analysis

- All the reachable states are in the set
  - Those not in the set are NOT reachable from the initial state
- If the intersection with !P is '0'
  - Property P is always true (for all input combinations in time infinity)

Property is formally proved!!

Limitation of BDD-based Model Checking

- Sounds good, if we can compute the set of reachable states, we can formally prove the properties...

- But the fact is that real design usually contains thousands of FFs, and the combinational cone may be as big as million gates.... Memory explosion problem!

Conclusion: BDD-Based Model Checking

- BDD is one important formal engine. It can compactly represent the circuit function/transition and compute the set of reachable states.

- However, the problem with BDD is that it tries to compute **all (or most) of the solutions at once**, and thus may suffer from memory space explosion problem...

Can we trade some “time” for the “space”?

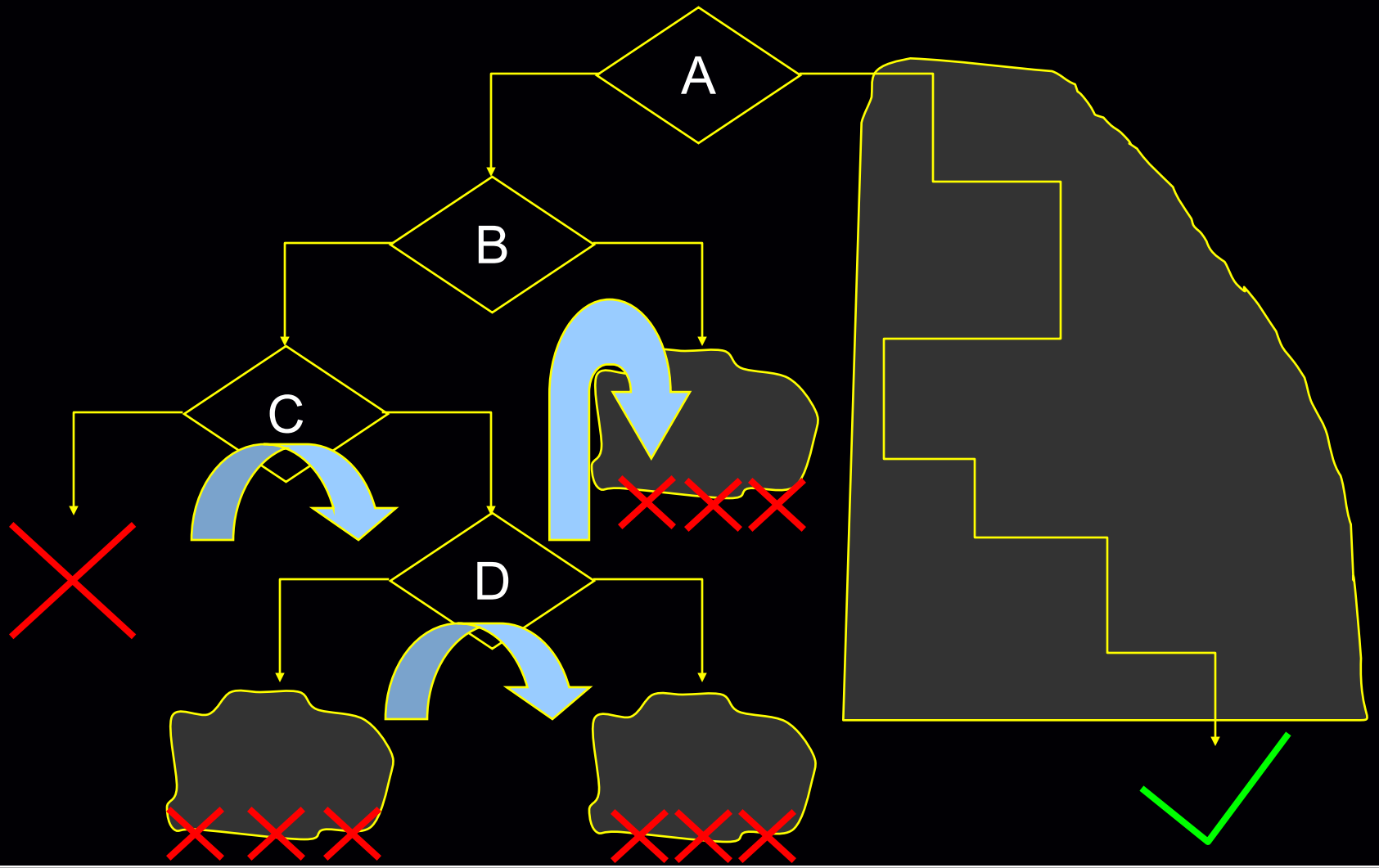
Think about the Human Search Process

(aka: search for !p)

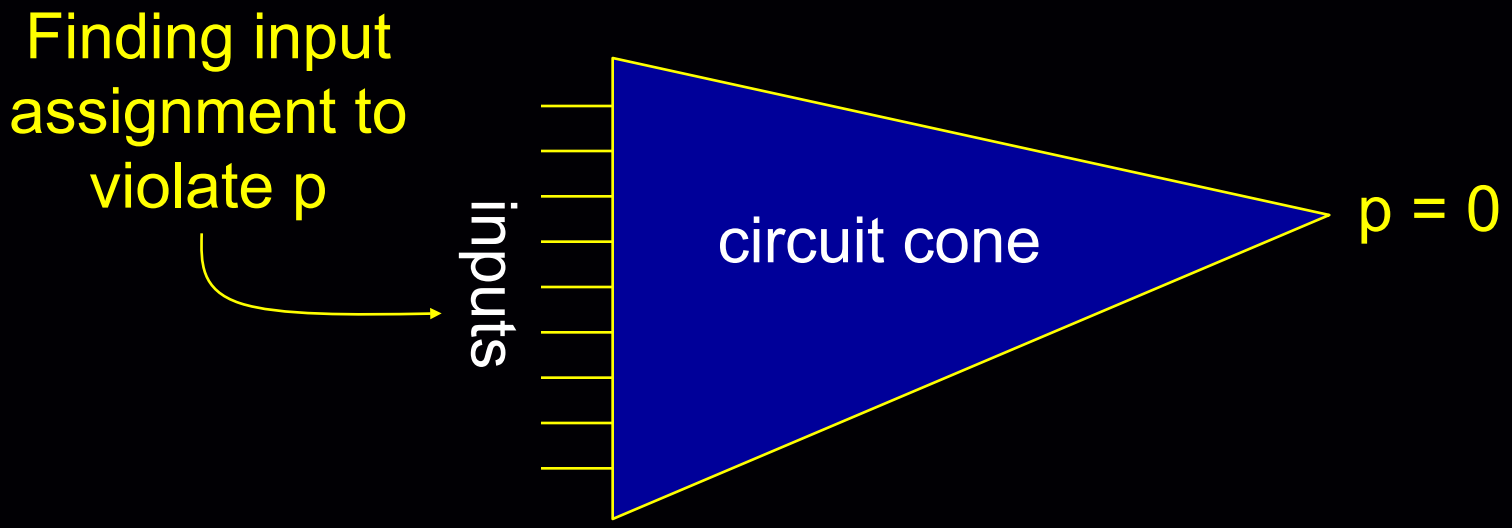
- When you search for something, you usually seek for one clue after another ---
  - Containing a keyword
  - Asking a question
  - By a direction, in a room, etc

- After some steps, if it is surely NOT there, you will reverse or revise some of the previous guesses/decisions and continue...

Something like --- A Decision Tree

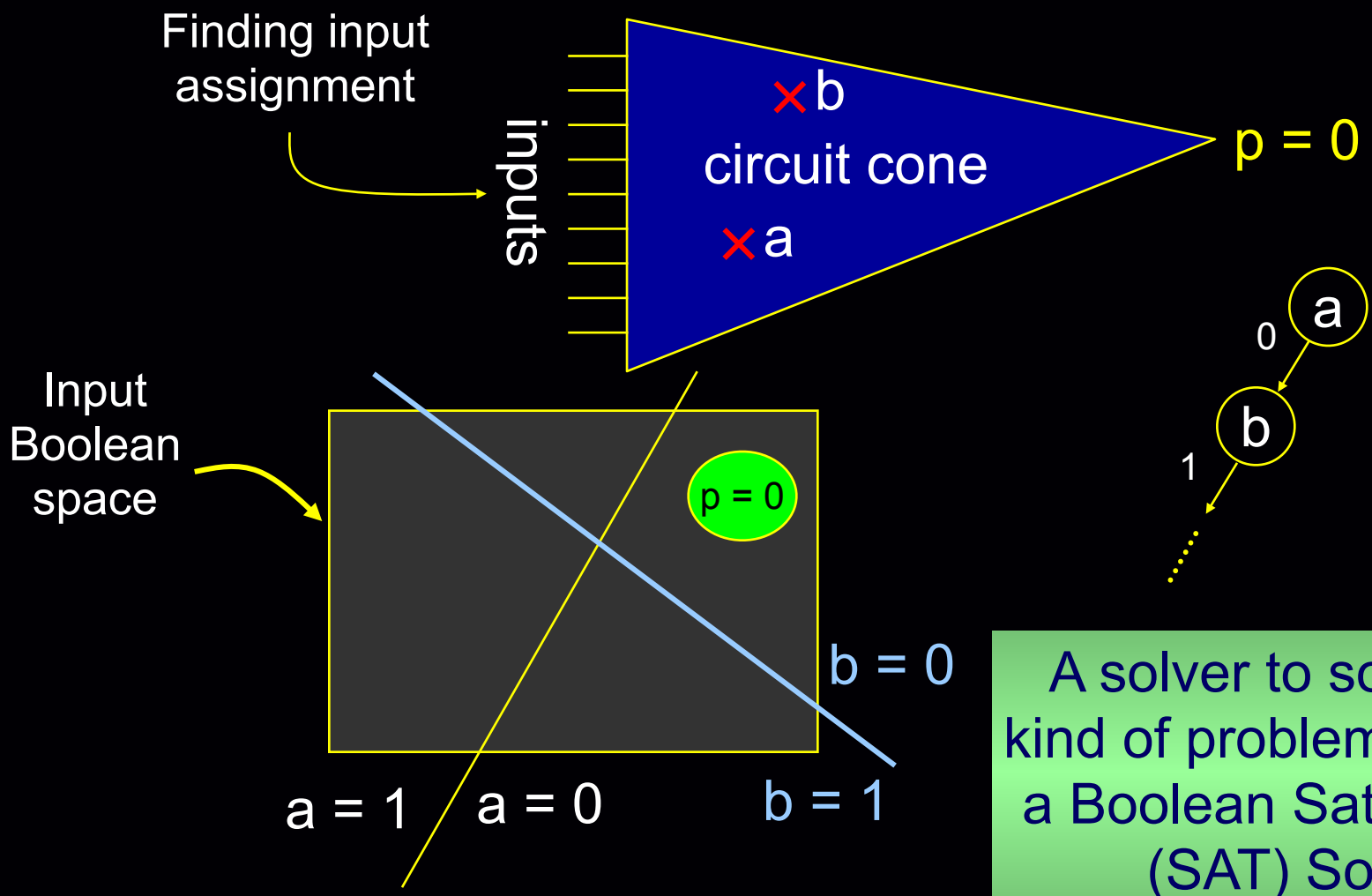


What about circuit properties?



- What are the “decisions”?
  - “Assigning input value one at a time” ?
    - ➔ Enumeration method: exponential complexity...
  - “Assigning values to internal signals” ?
    - ➔ Still exponential complexity...

Boolean Satisfiability Checking for Properties on Circuit



A solver to solve this kind of problem is called a Boolean Satisfiability (SAT) Solver

Introduction to Boolean Satisfiability (SAT)

A fundamental problem in computer science

- Given a Boolean network F: $B^n \rightarrow B$,
  where B = { 0, 1 }, and
  n is the number of inputs I = { x_1, x_2,... , x_n }

- Boolean Satisfiability
  → Finding an input assignment

  a: { $x_1 = a_1$, $x_2 = a_2$,... , $x_n = a_n \mid a_i \in B$ }

  such that F = 1.

- Exponential complexity...?

Try this out --
SAT Game

Complexity of SAT solver

- Boolean Satisfiability (SAT) was the first proven NP-complete problem by Dr. S. Cook in 1971
  - Given n variables, the number of decisions can be as many as 2^n...
  - If there is a non-deterministic machine, we can construct a polynomial-time algorithm that can guarantee to prove/disprove the SAT problem

  [Pitfall?] Unless there is a non-deterministic machine, we cannot construct a polynomial-time SAT algorithm (i.e. P != NP)

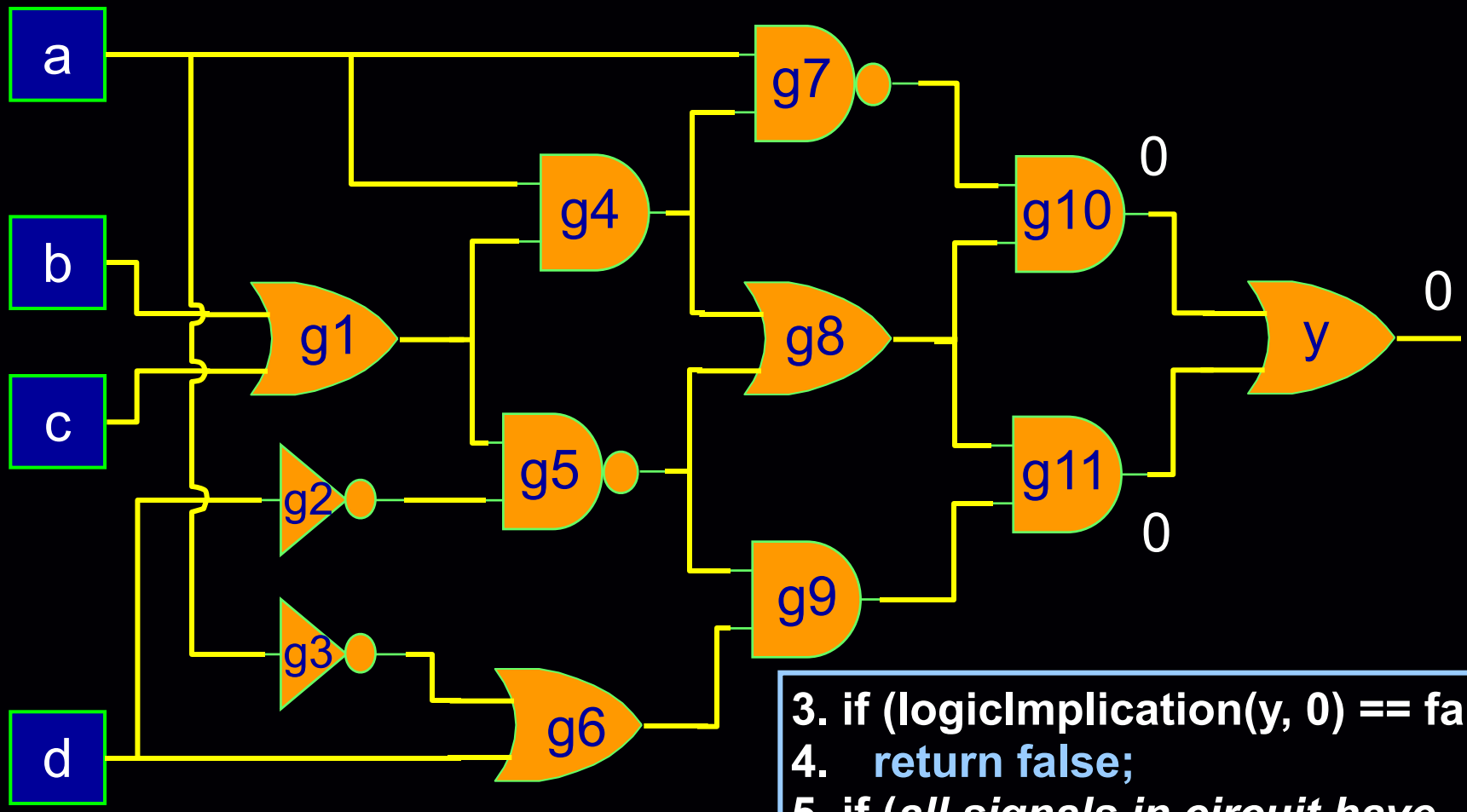
➔ How can SAT be useable for million-gate designs?

A Very Basic SAT Algorithm

```
1. bool sat(Gate g, Value v)
2. {
3.    if (logicImplication(g, v) == false) // conflict
4.       return false;
5.    if (all signals in circuit have been implied)
6.       return true;
7.    pick an unassigned signal s
8.    if (sat(s, V0) == true) // found a solution
9.       return true;
10.   backtrack(s);
11.   if (sat(s, ~V0) == true) // found a solution
12.      return true;
13.   backtrack(s);
14.   return false; // no solution
15.}
```

Proving always(y == 1)

sat(y, 0)



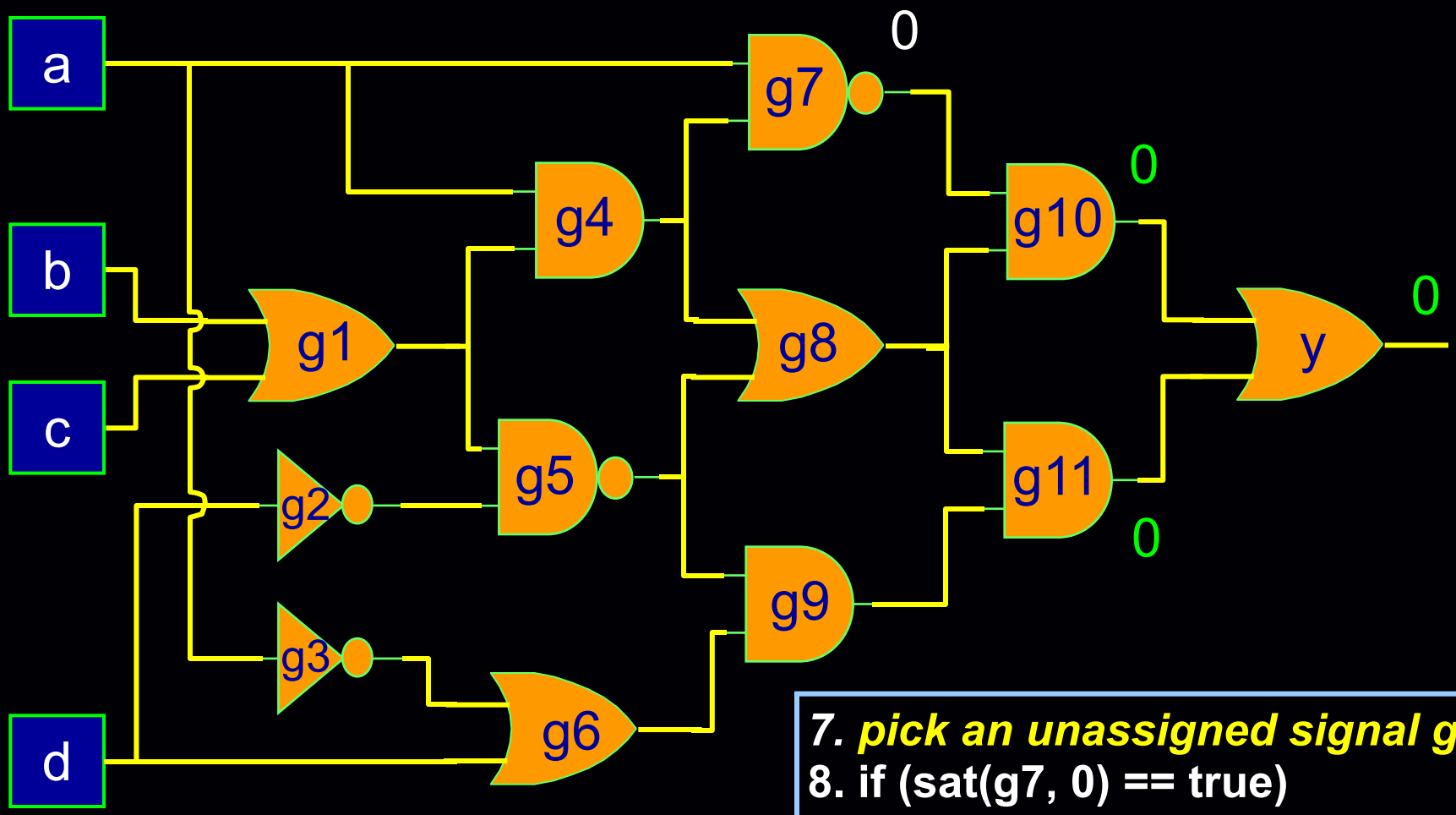

```
3. if (logicImplication(y, 0) == false)
4.    return false;
5. if (all signals in circuit have
6.    been implied)   return true;
```

A Very Basic SAT Algorithm

```
1. bool sat(Gate g, Value v)
2. {
3.    if (logicImplication(g, v) == false) // conflict
4.       return false;
5.    if (all signals in circuit have been implied)
6.       return true;
7.    pick an unassigned signal s
8.    if (sat(s, V0) == true) // found a solution
9.       return true;
10.   backtrack(s);
11.   if (sat(s, ~V0) == true) // found a solution
12.      return true;
13.   backtrack(s);
14.   return false;
15.}
```

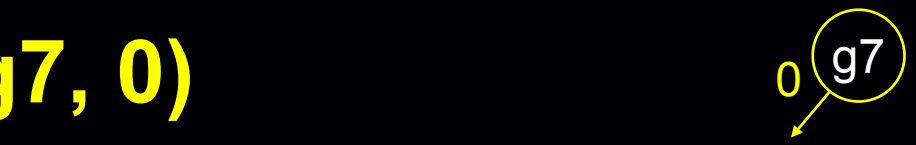
Solve: sat (y, 0)

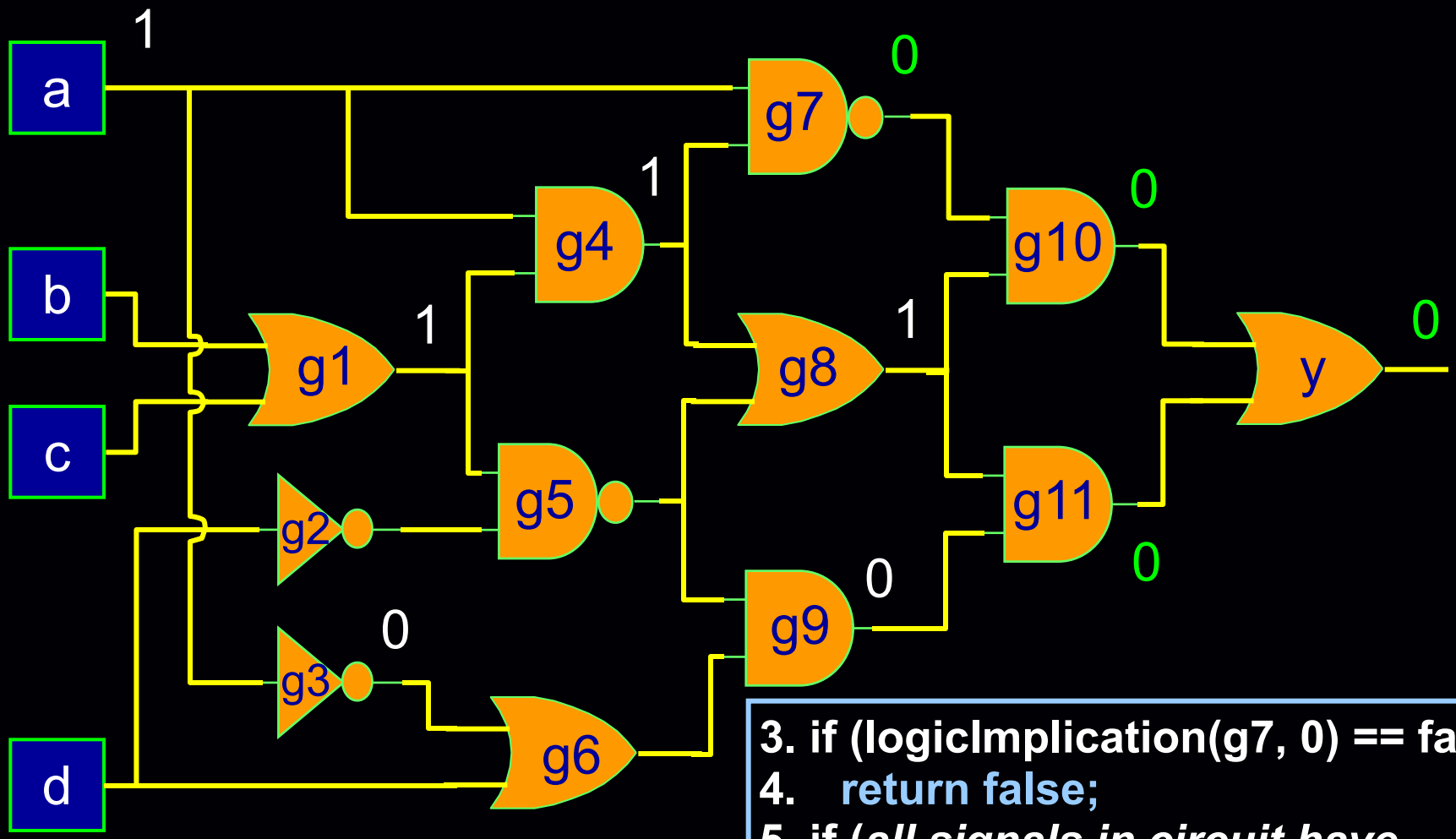
sat (g7, 0)



7. *pick an unassigned signal g7*
8. if (sat(g7, 0) == true)
9. return true;

Recursion: sat (g7, 0)

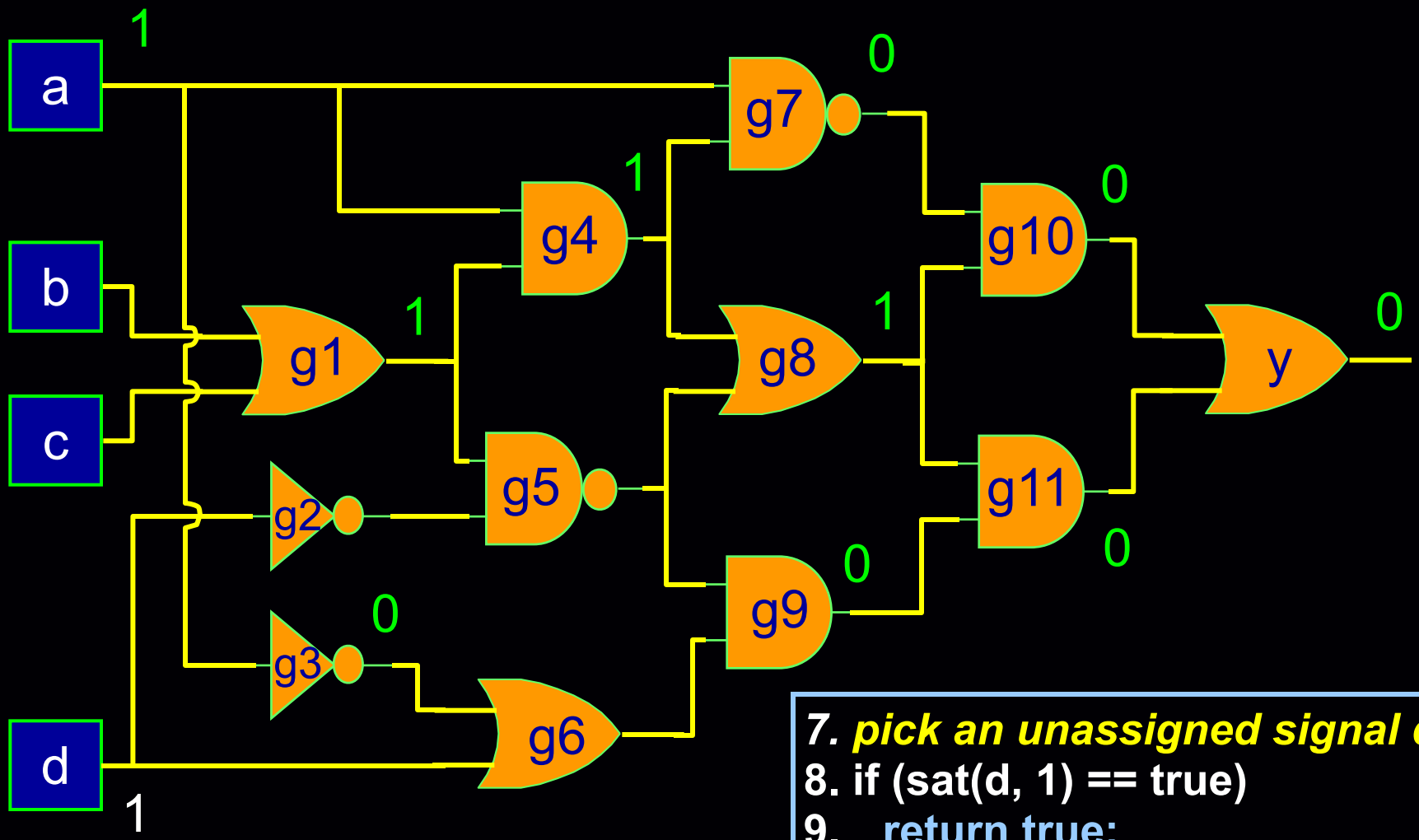





```
3. if (logicImplication(g7, 0) == false)
4.    return false;
5. if (all signals in circuit have
6.    been implied)   return true;
```

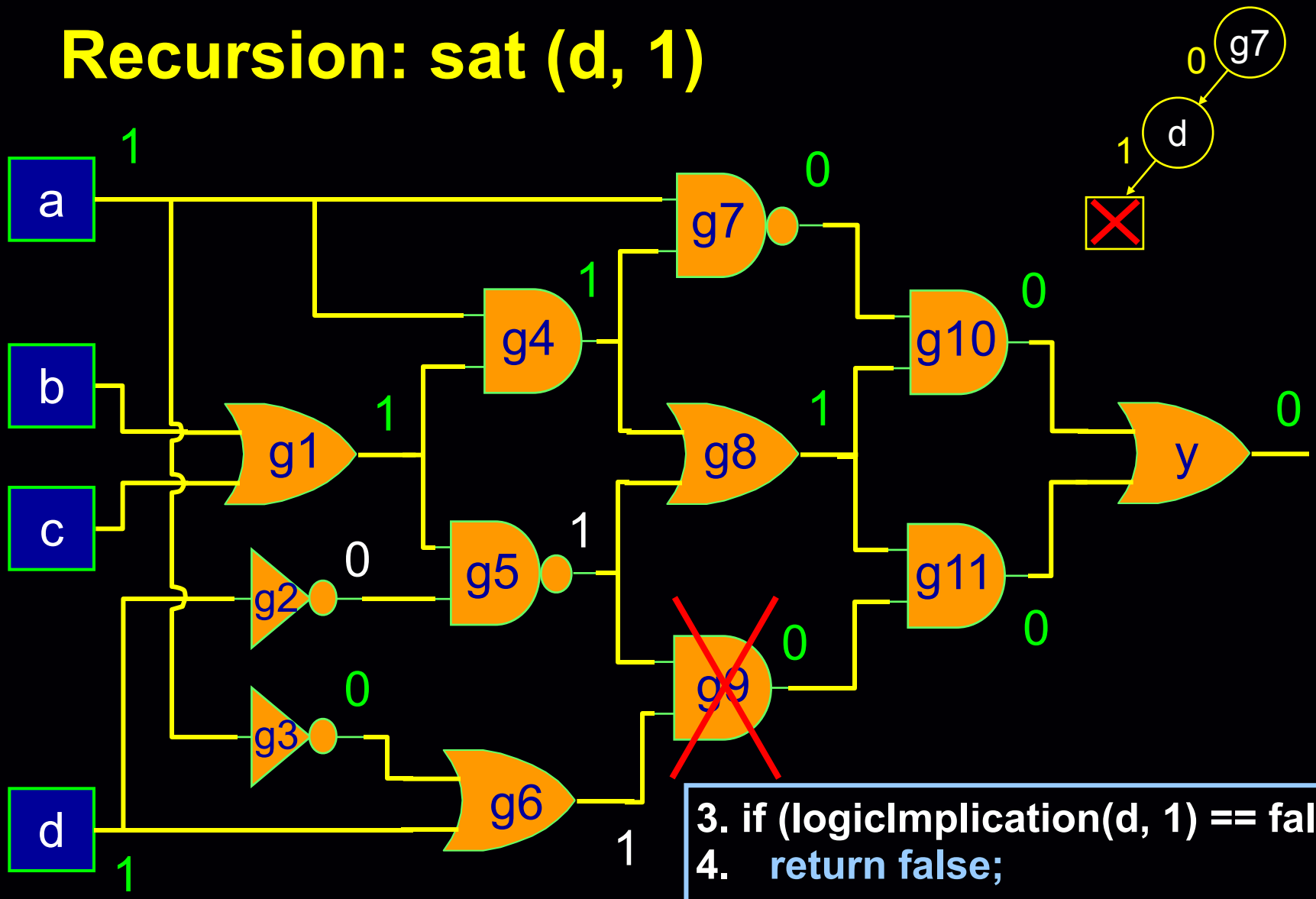
Recursion: sat (g7, 0)

sat (d, 1)



7. *pick an unassigned signal d*
8. if (sat(d, 1) == true)
9. return true;

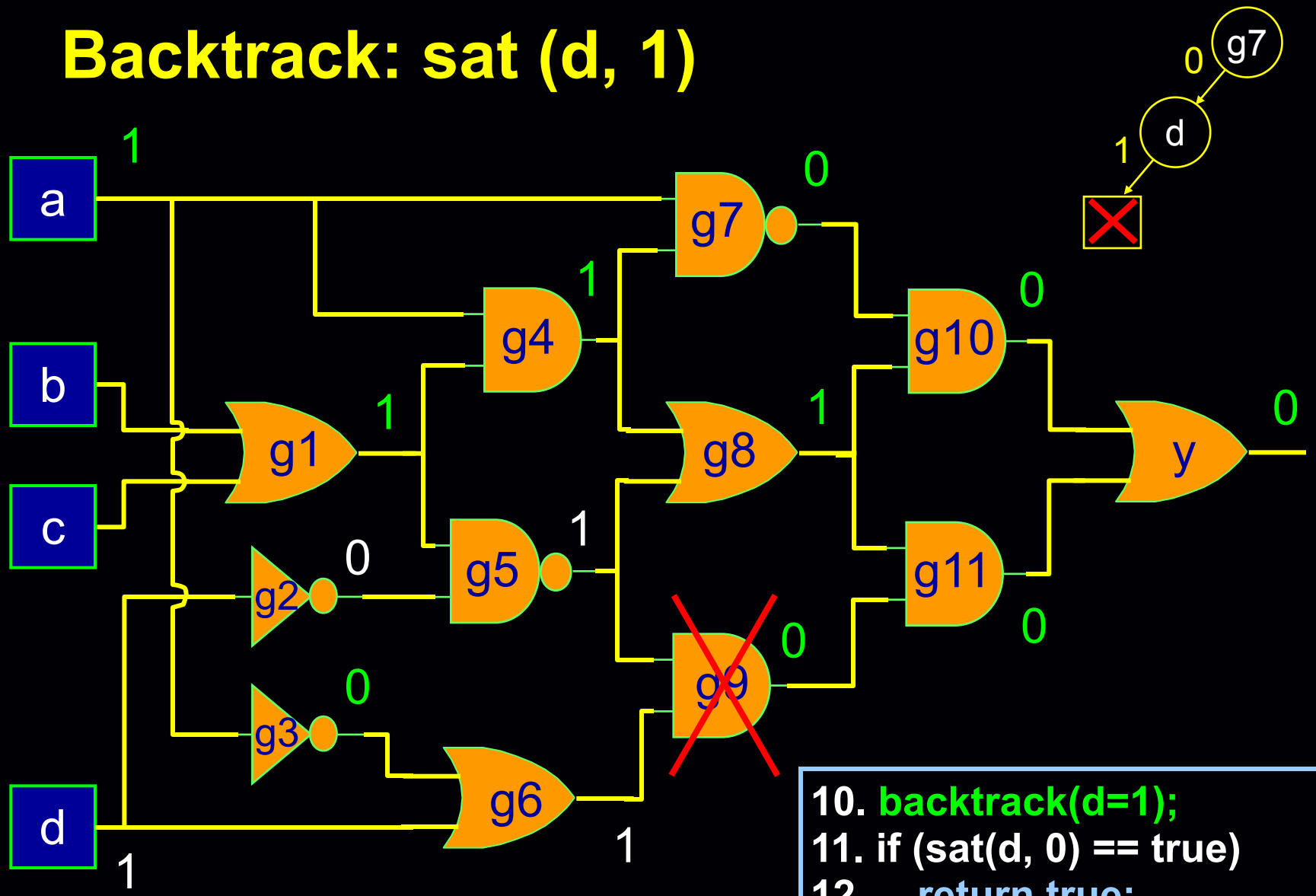
Recursion: sat (d, 1)



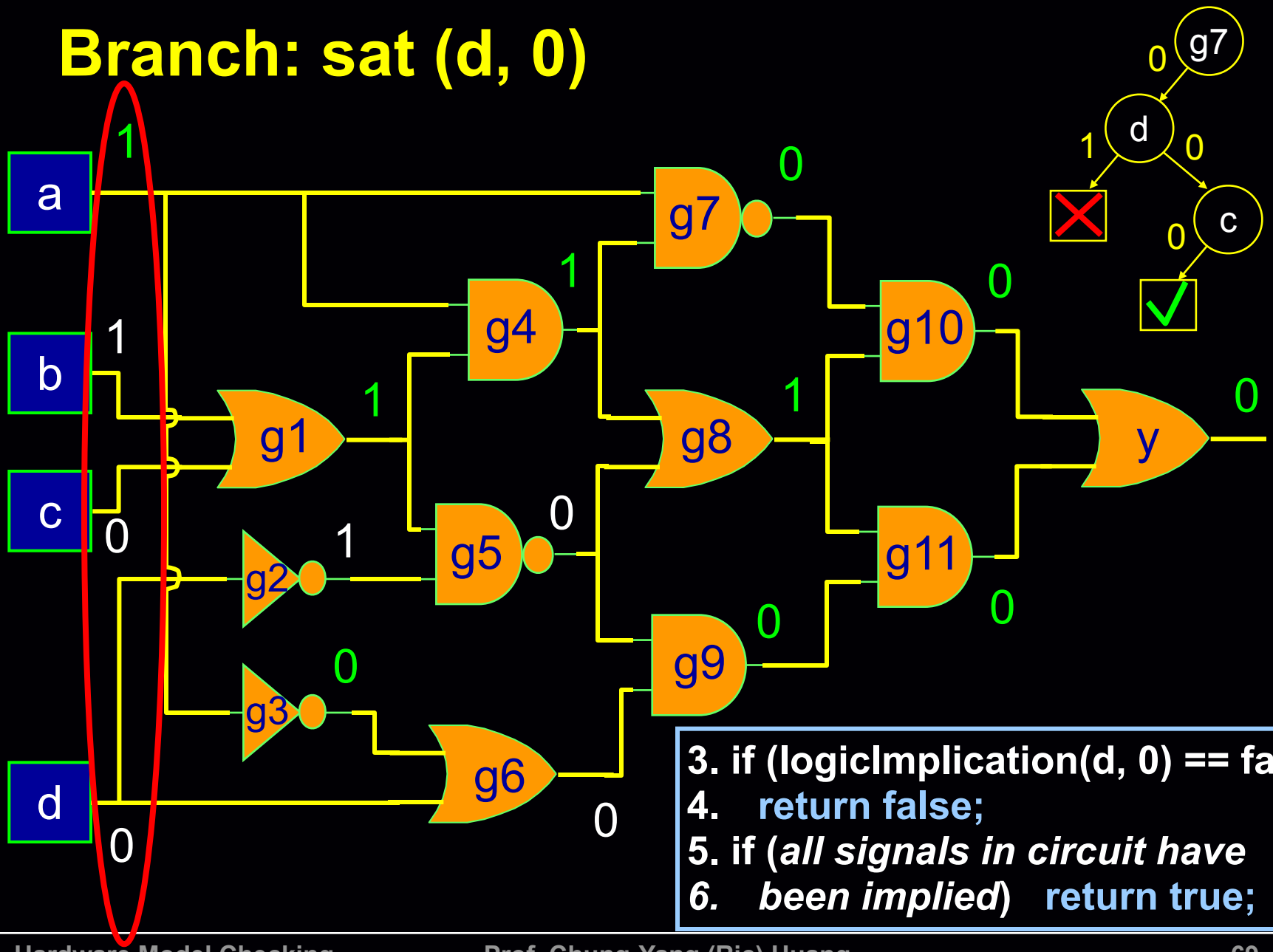
A Very Basic SAT Algorithm

```
1. bool sat(Gate g, Value v)
2. {
3.    if (logicImplication(g, v) == false) // conflict
4.       return false;
5.    if (all signals in circuit have been implied)
6.       return true;
7.    pick an unassigned signal s
8.    if (sat(s, V0) == true) // found a solution
9.       return true;
10.   backtrack(s);
11.   if (sat(s, ~V0) == true) // found a solution
12.      return true;
13.   backtrack(s);
14.   return false;
15.}
```

Backtrack: sat (d, 1)



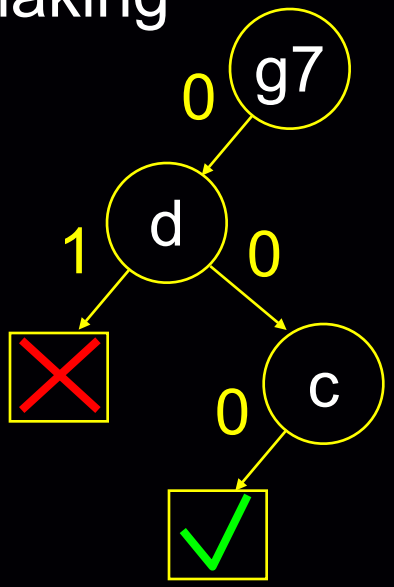
Branch: sat (d, 0)




```
3. if (logicImplication(d, 0) == false)
4.    return false;
5. if (all signals in circuit have
6.    been implied)   return true;
```

What are the key steps in a SAT algorithm?

- Decision making



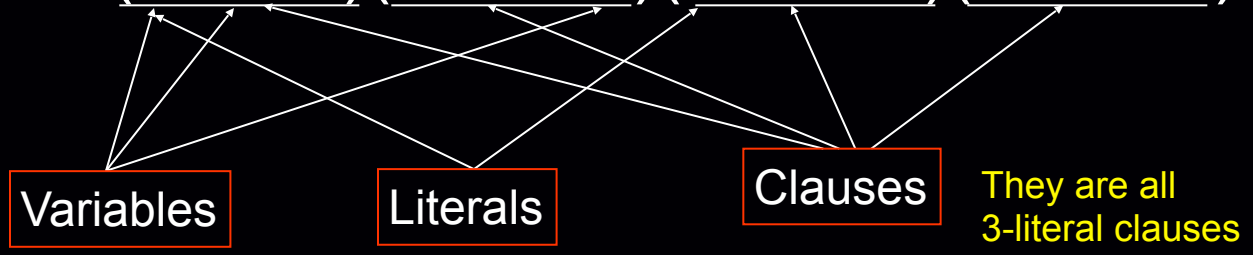
- Logic implication (Boolean Constraint Propagation, BCP)
- What else?

In the previous “very basic SAT algorithm”, we make decisions and apply logic implications on the circuit.

However, in practice, most SAT solvers are implemented on a “Conjunctive Normal Form (CNF)” data structure.

Types of Boolean Satisfiability Solvers

1. Conjunctive Normal Form (CNF) Based
   - Boolean function is represented as a CNF (i.e. Product of Sum, POS format)
   - e.g. $(a+b+c)(a'+b'+c)(a'+b+c')(a+b'+c')$



   - To be satisfied, all the clauses should be '1'

2. Circuit-Based
   - Boolean function is represented as a circuit netlist
   - SAT algorithm is directly operated on the netlist

CNF vs. Circuit SAT

- Although CNF and circuit SAT solvers look quite different, their algorithms can be very similar

- CNF SAT
  - Simpler data structure; easier to implement
- Circuit SAT
  - Structural information; extensible to word-level

➔ In the following slides, we will focus on the easier-to-implement solver, CNF SAT, first

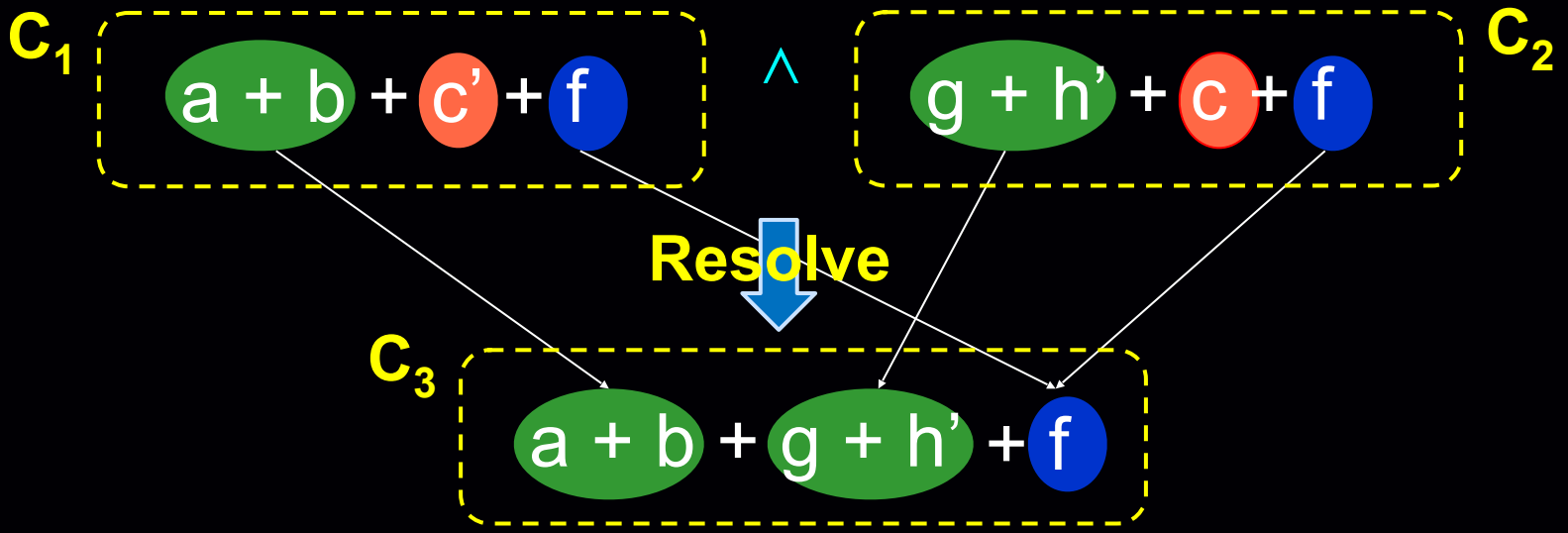
CNF-Based SAT Algorithm

1. Davis, Putnam, 1960
   - Explicit resolution based
   - May explode in memory
2. Davis, Logemann, Loveland, (DLL) 1962
   - Search based.
   - Most successful, basis for almost all modern SAT solvers
   - Learning and non-chronological backtracking, 1996
3. Stålmarcks algorithm, 1980s
   - Proprietary algorithm. Patented.
   - Commercial versions available
4. Stochastic Methods, 1992
   - Unable to prove unsatisfiability, but may find solutions for a satisfying problem quickly.
   - Local search and hill climbing

Souce: Prof. Sharad Malik, “The Quest for Efficient Boolean Satisfiability Solvers”

Resolution

- Resolution of a pair of clauses with exactly ONE incompatible variable
  - Two clauses are said to have distance 1



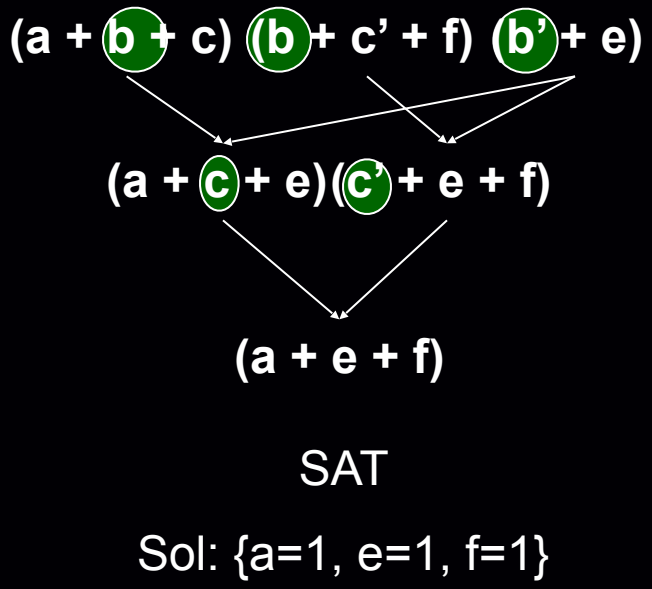
  - $C_1 \wedge C_2 \rightarrow C_3$ or $C_3 \rightarrow C_1 \wedge C_2$?
  - Existential quantification?

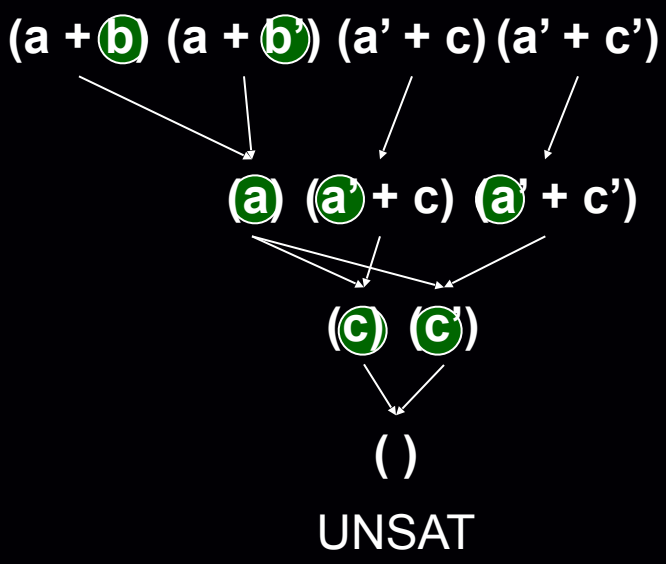
Souce: Prof. Sharad Malik, "The Quest for Efficient Boolean Satisfiability Solvers"

Davis Putnam Algorithm

M .Davis, H. Putnam, “A computing procedure for quantification theory", *J. of ACM*, Vol. 7, pp. 201-214, 1960 (360 citations in citeseer)

- Existential abstraction using resolution
- Iteratively select a variable for resolution till no more variables are left.





Potential memory explosion problem!

Souce: Prof. Sharad Malik, “The Quest for Efficient Boolean Satisfiability Solvers”

Boolean Satisfiability (SAT) Algorithm

1. Davis, Putnam, 1960
   - Explicit resolution based
   - May explode in memory
2. Davis, (Putnam), Logemann, Loveland, (D(P)LL) 1962
   - Search based.
   - Most successful, basis for almost all modern SAT solvers
   - Learning and non-chronological backtracking, 1996
3. Stålmarcks algorithm, 1980s
   - Proprietary algorithm. Patented.
   - Commercial versions available
4. Stochastic Methods, 1992
   - Unable to prove unsatisfiability, but may find solutions for a satisfying problem quickly.
   - Local search and hill climbing

Basic DLL Procedure - DFS

(a' + b + c)

(a + c + d)

(a + c + d')

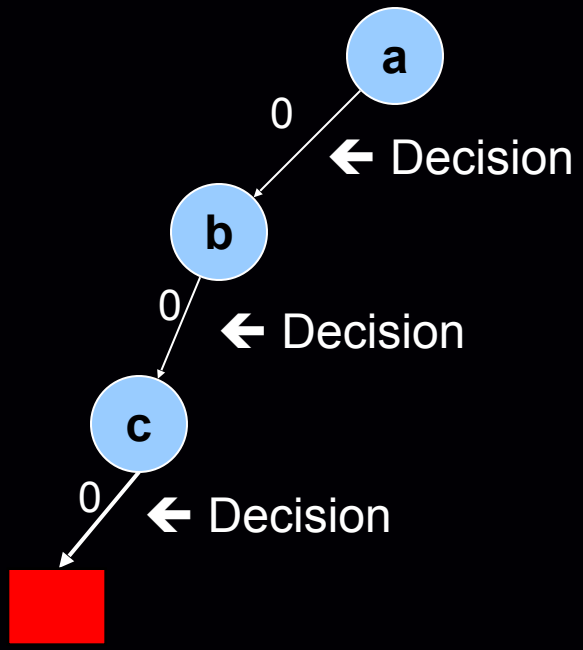
(a + c' + d)

(a + c' + d')

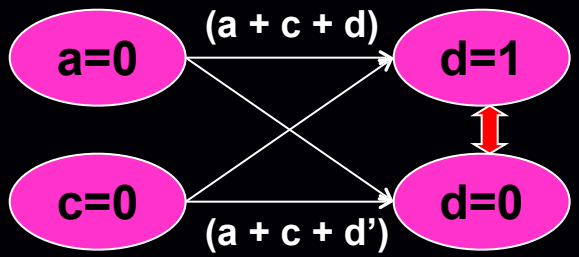
(b' + c' + d)

(a' + b + c')

(a' + b' + c)



Implication Graph



Conflict!

Modified from: Prof. Sharad Malik, "The Quest for Efficient Boolean Satisfiability Solvers"

Basic DLL Procedure - DFS

(a’ + b + c)

(a + c + d)

(a + c + d’)

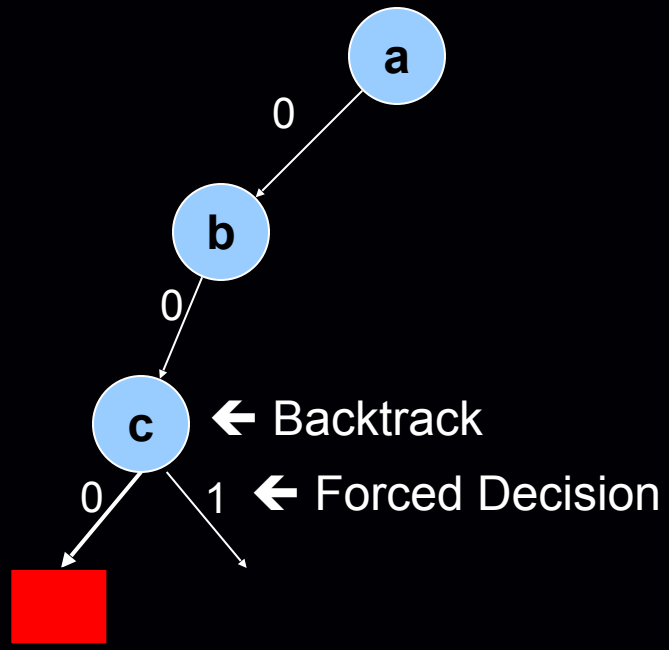
(a + c’ + d)

(a + c’ + d’)

(b’ + c’ + d)

(a’ + b + c’)

(a’ + b’ + c)



Modified from: Prof. Sharad Malik, “The Quest for Efficient Boolean Satisfiability Solvers”

Basic DLL Procedure - DFS

(a’ + b + c)

(a + c + d)

(a + c + d’)

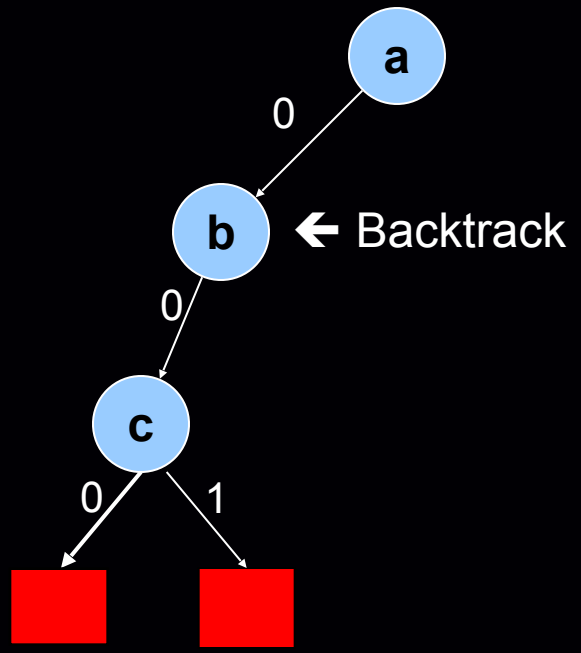
(a + c’ + d)

(a + c’ + d’)

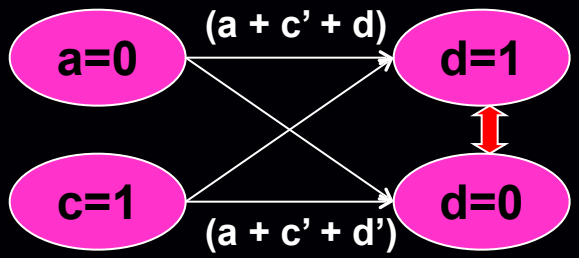
(b’ + c’ + d)

(a’ + b + c’)

(a’ + b’ + c)



Implication Graph



Conflict!

Modified from: Prof. Sharad Malik, “The Quest for Efficient Boolean Satisfiability Solvers”

Basic DLL Procedure - DFS

(a' + b + c)

(a + c + d)

(a + c + d')

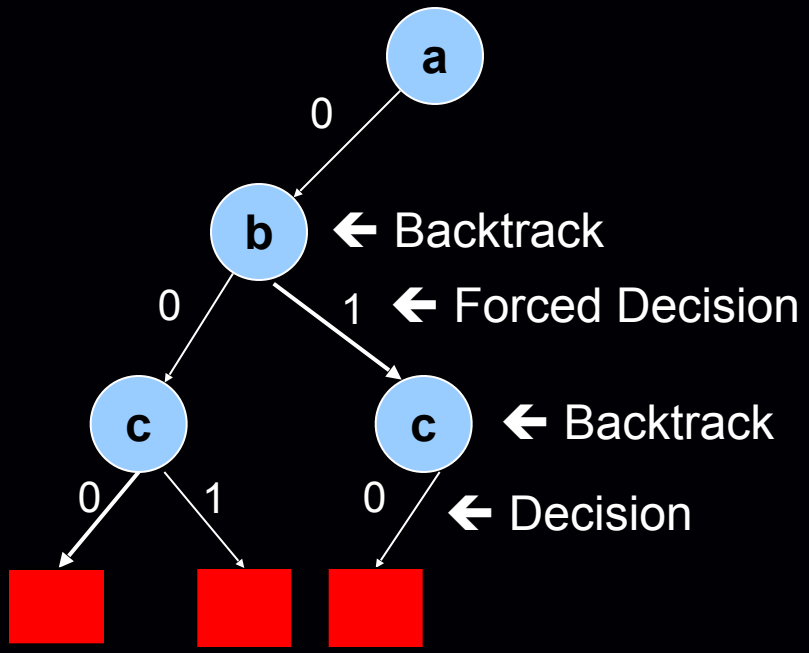
(a + c' + d)

(a + c' + d')

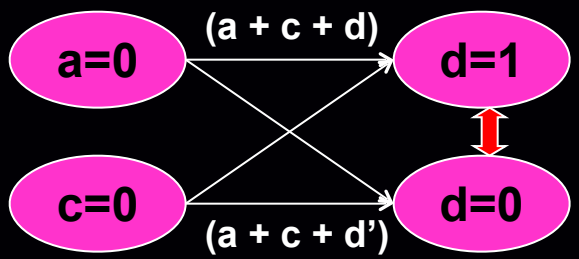
(b' + c' + d)

(a' + b + c')

(a' + b' + c)



Implication Graph



Conflict!

Modified from: Prof. Sharad Malik, "The Quest for Efficient Boolean Satisfiability Solvers"

Basic DLL Procedure - DFS

(a’ + b + c)

(a + c + d)

(a + c + d’)

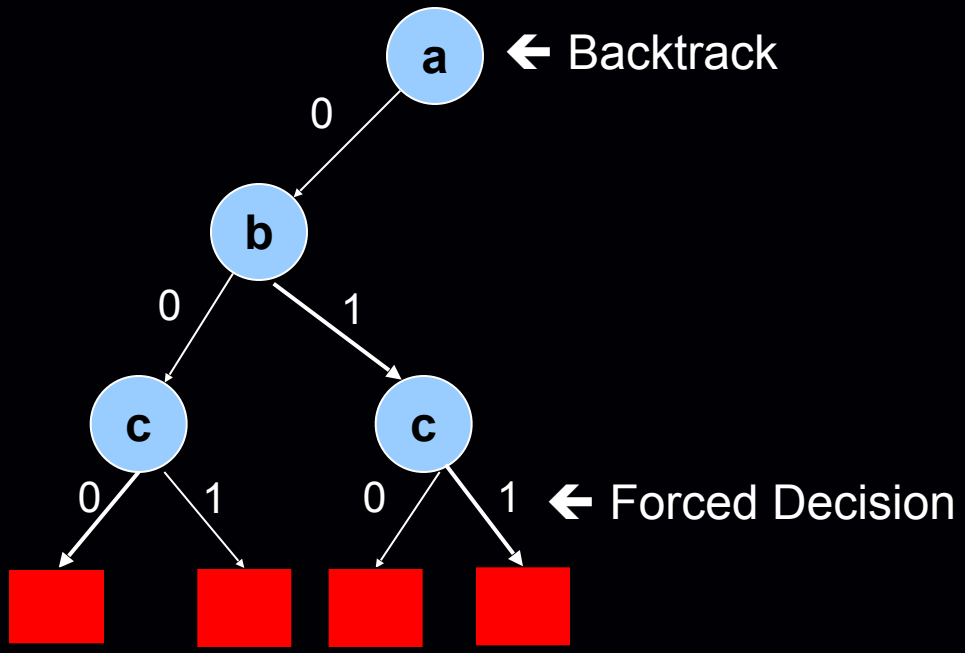
(a + c’ + d)

(a + c’ + d’)

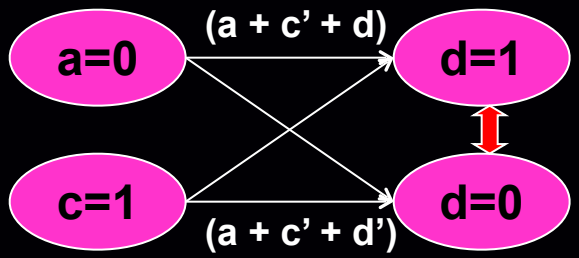
(b’ + c’ + d)

(a’ + b + c’)

(a’ + b’ + c)



Implication Graph



Conflict!

Modified from: Prof. Sharad Malik, “The Quest for Efficient Boolean Satisfiability Solvers”

Basic DLL Procedure - DFS

(a’ + b + c)

(a + c + d)

(a + c + d’)

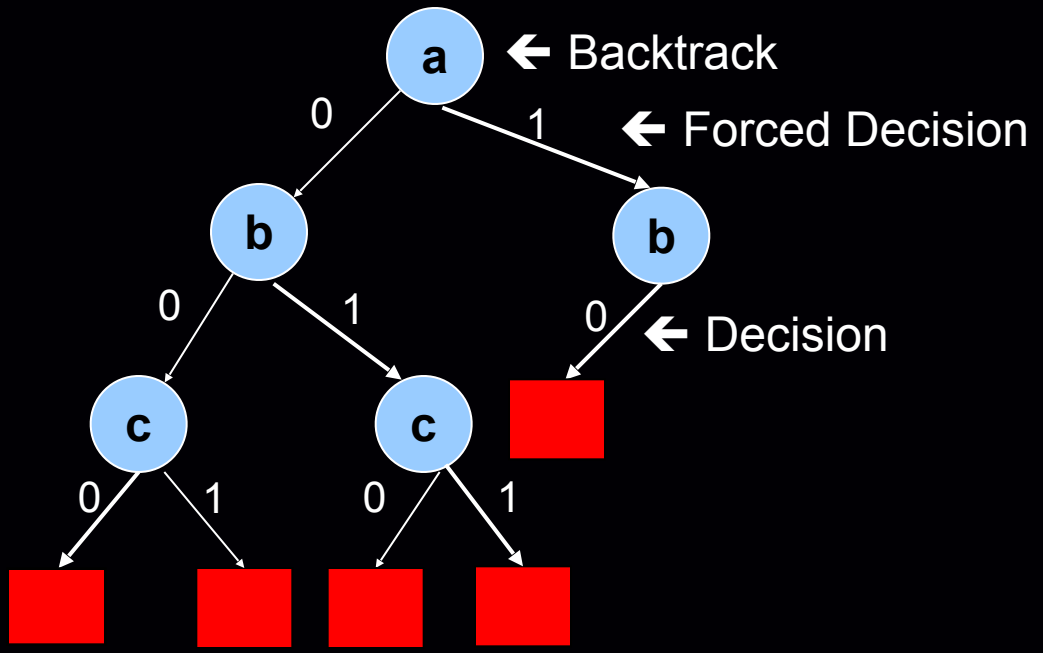
(a + c’ + d)

(a + c’ + d’)

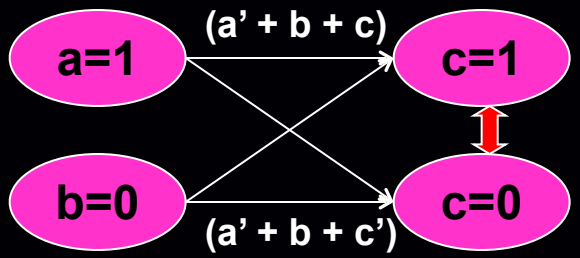
(b’ + c’ + d)

(a’ + b + c’)

(a’ + b’ + c)



Implication Graph



Conflict!

Modified from: Prof. Sharad Malik, “The Quest for Efficient Boolean Satisfiability Solvers”

Basic DLL Procedure - DFS

(a’ + b + c)

(a + c + d)

(a + c + d’)

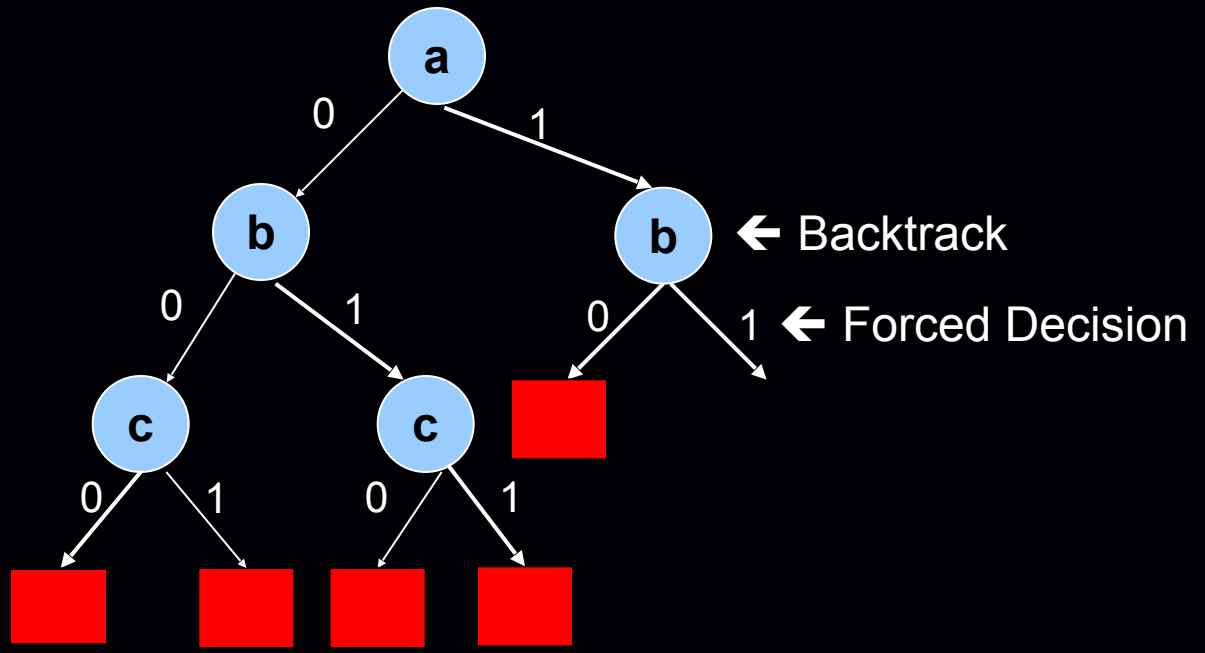
(a + c’ + d)

(a + c’ + d’)

(b’ + c’ + d)

(a’ + b + c’)

(a’ + b’ + c)



Modified from: Prof. Sharad Malik, “The Quest for Efficient Boolean Satisfiability Solvers”

Basic DLL Procedure - DFS

(a’ + b + c)

(a + c + d)

(a + c + d’)

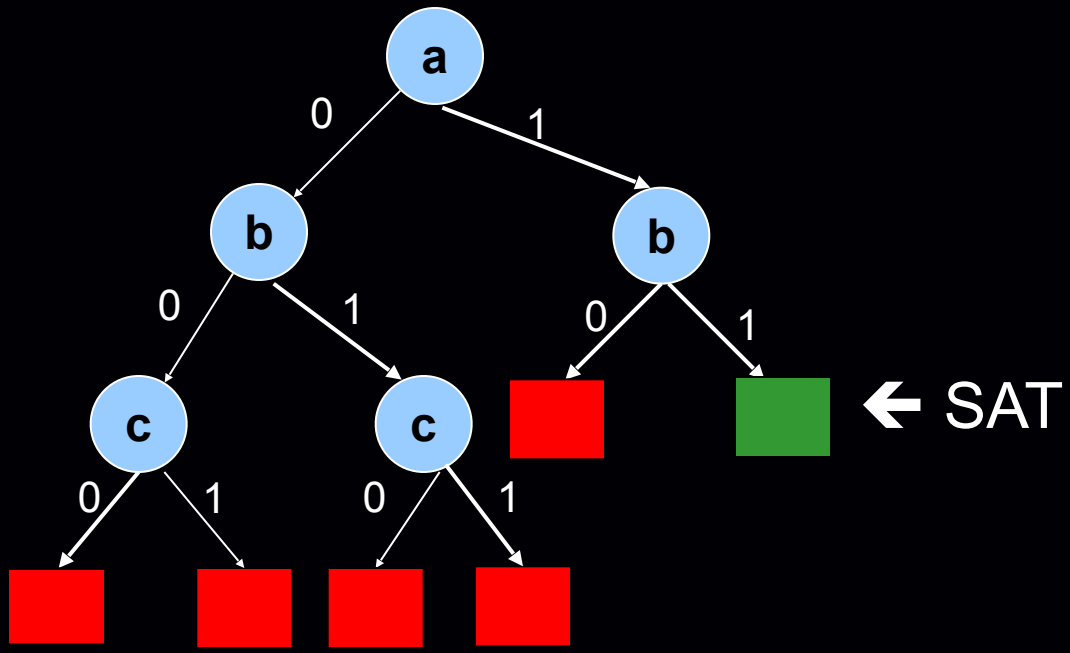
(a + c’ + d)

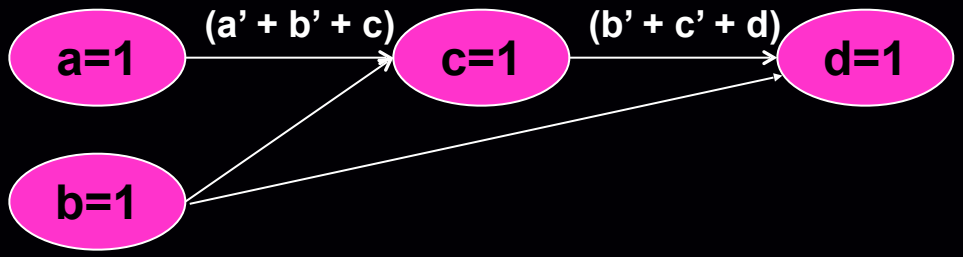
(a + c’ + d’)

(b’ + c’ + d)

(a’ + b + c’)

(a’ + b’ + c)





Modified from: Prof. Sharad Malik, “The Quest for Efficient Boolean Satisfiability Solvers”

Potentially exponential complexity!!

Did you see any unnecessary work?

SAT Improvements

1. Conflict-driven learning
   - Once we encounter a conflict

   ➔ Figure out the cause(s) of this conflict and prevent to see this conflict again!!

Conflict-Driven Learning

(a’ + b + c)

(a + c + d)

(a + c + d’)

(a + c’ + d)

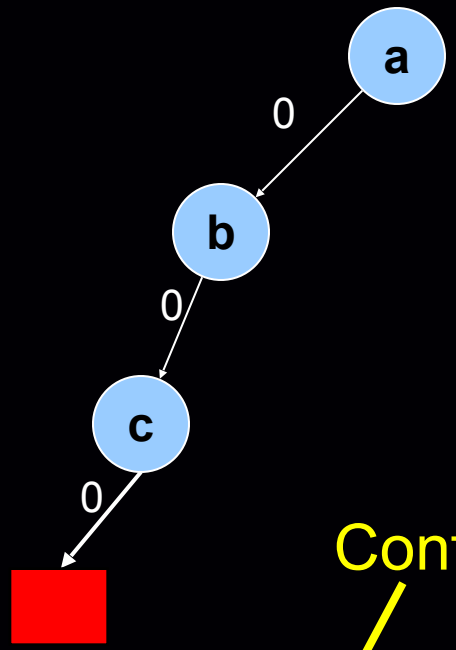
(a + c’ + d’)

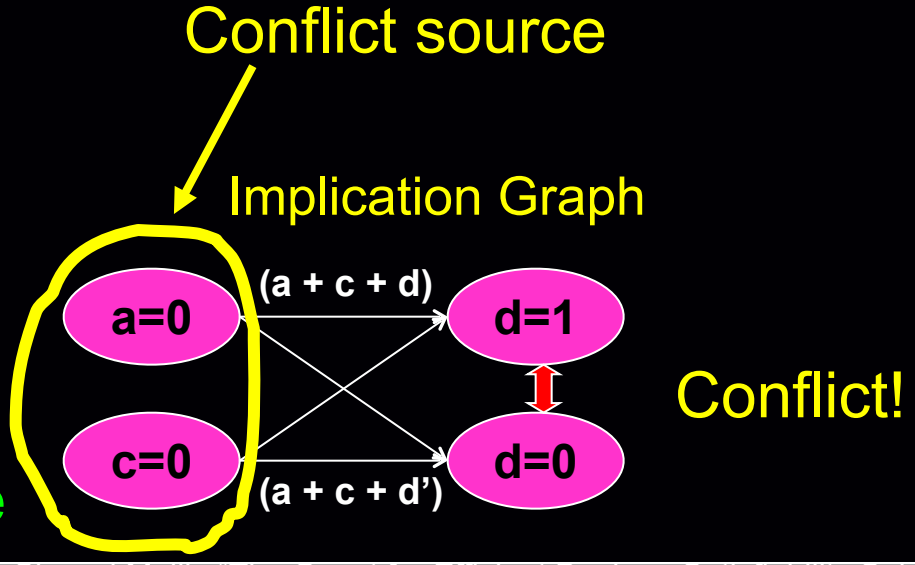
(b’ + c’ + d)

(a’ + b + c’)

(a’ + b’ + c)

(a + c) Learned clause





Modified from: Prof. Sharad Malik, “The Quest for Efficient Boolean Satisfiability Solvers”

SAT Improvements

2. Non-chronological backtracking
   - Since we get a learned clause from the conflict analysis…

   ➔ Instead of backtracking 1 decision at a time, backtrack to the "next-to-the-last" variable in the learned clause

Non-Chronological Backtracking

(a’ + b + c)

(a + c + d)

(a + c + d’)

(a + c’ + d)

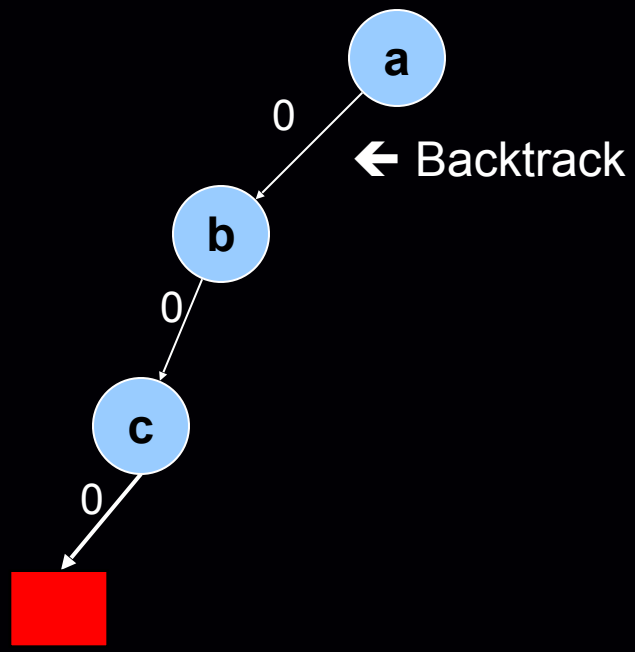
(a + c’ + d’)

(b’ + c’ + d)

(a’ + b + c’)

(a’ + b’ + c)

(a + c) Learned clause



- ‘a’ is the next-to-the-last variable in the learned clause
- Backtrack c = 0 && b = 0

Modified from: Prof. Sharad Malik, “The Quest for Efficient Boolean Satisfiability Solvers”

Deduced Implication from Learned Clause

(a’ + b + c)

(a + c + d)

(a + c + d’)

(a + c’ + d)

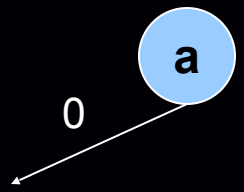
(a + c’ + d’)

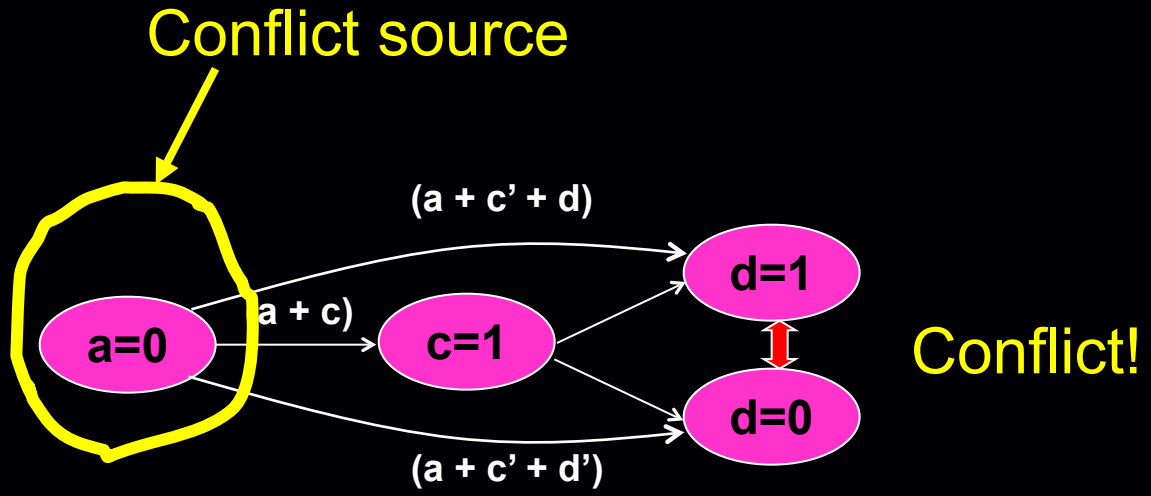
(b’ + c’ + d)

(a’ + b + c’)

(a’ + b’ + c)

(a + c)





Modified from: Prof. Sharad Malik, “The Quest for Efficient Boolean Satisfiability Solvers”

Deduced Implication from Learned Clause

(a' + b + c)

(a + c + d)

(a + c + d')

(a + c' + d)

(a + c' + d')

(b' + c' + d)

(a' + b + c')

(a' + b' + c)

(a + c)

(a) Learned clause

- Since there is only one variable in the learned clause

  → No one is the next-to-the-last variable

- Backtrack all decisions

  → (a = 1) is asserted

Deduced Implication from Learned Clause

(a’ + b + c)

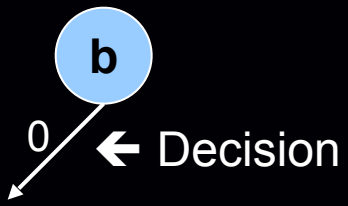
(a + c + d)

(a + c + d’)

(a + c’ + d)

(a + c’ + d’)

(b’ + c’ + d)



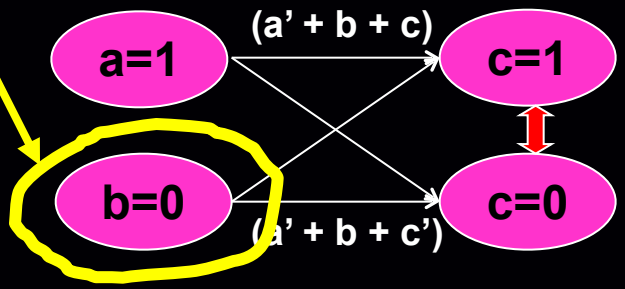
(a’ + b + c’)

(a’ + b’ + c)

(a + c)

(a)

Conflict source



Conflict!

Deduced Implication from Learned Clause

(a’ + b + c)

(a + c + d)

(a + c + d’)

(a + c’ + d)

(a + c’ + d’)

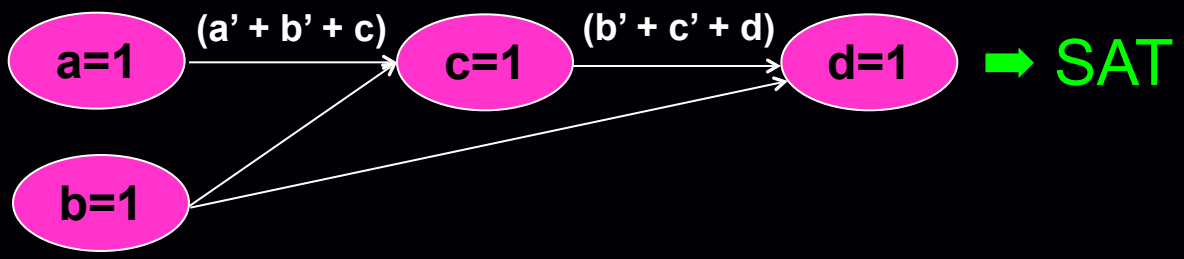
(b’ + c’ + d)

(a’ + b + c’)

(a’ + b’ + c)

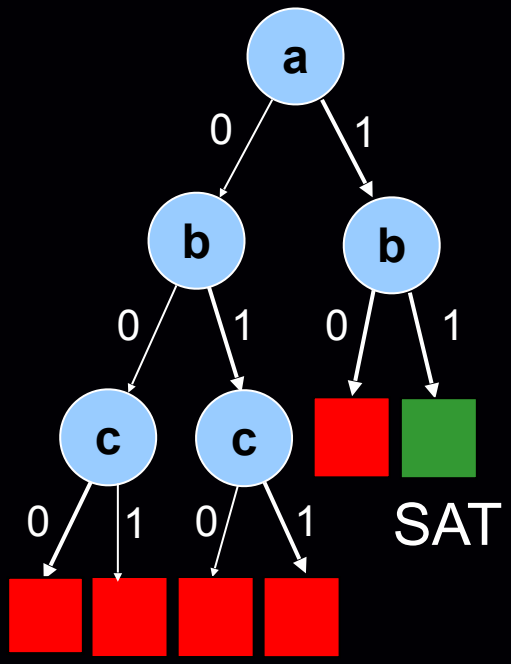
(a + c)

(a) (b) Learned clause

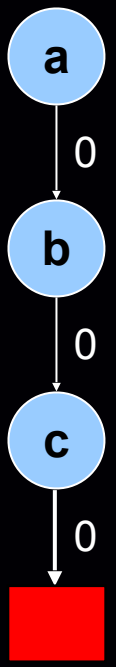


Modified from: Prof. Sharad Malik, “The Quest for Efficient Boolean Satisfiability Solvers”

Without vs. With Conflict-Driven Learning

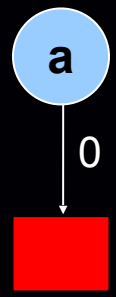


Without conflict-driven learning



Conflict !!

(a + c) is learned

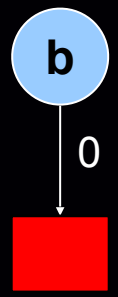


Backtrack to 'a'

(c = 1) is implied

Conflict !!

(a) is learned



Backtrack ALL

(a = 1) is asserted

Decision (b = 0)

Conflict !!

(b) is learned

Backtrack ALL

(b = 1) is asserted

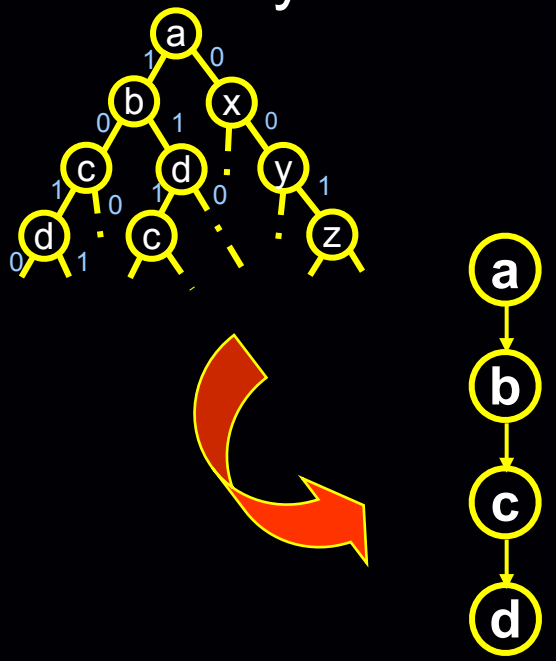
(c = 1) is implied

(d = 1) is implied

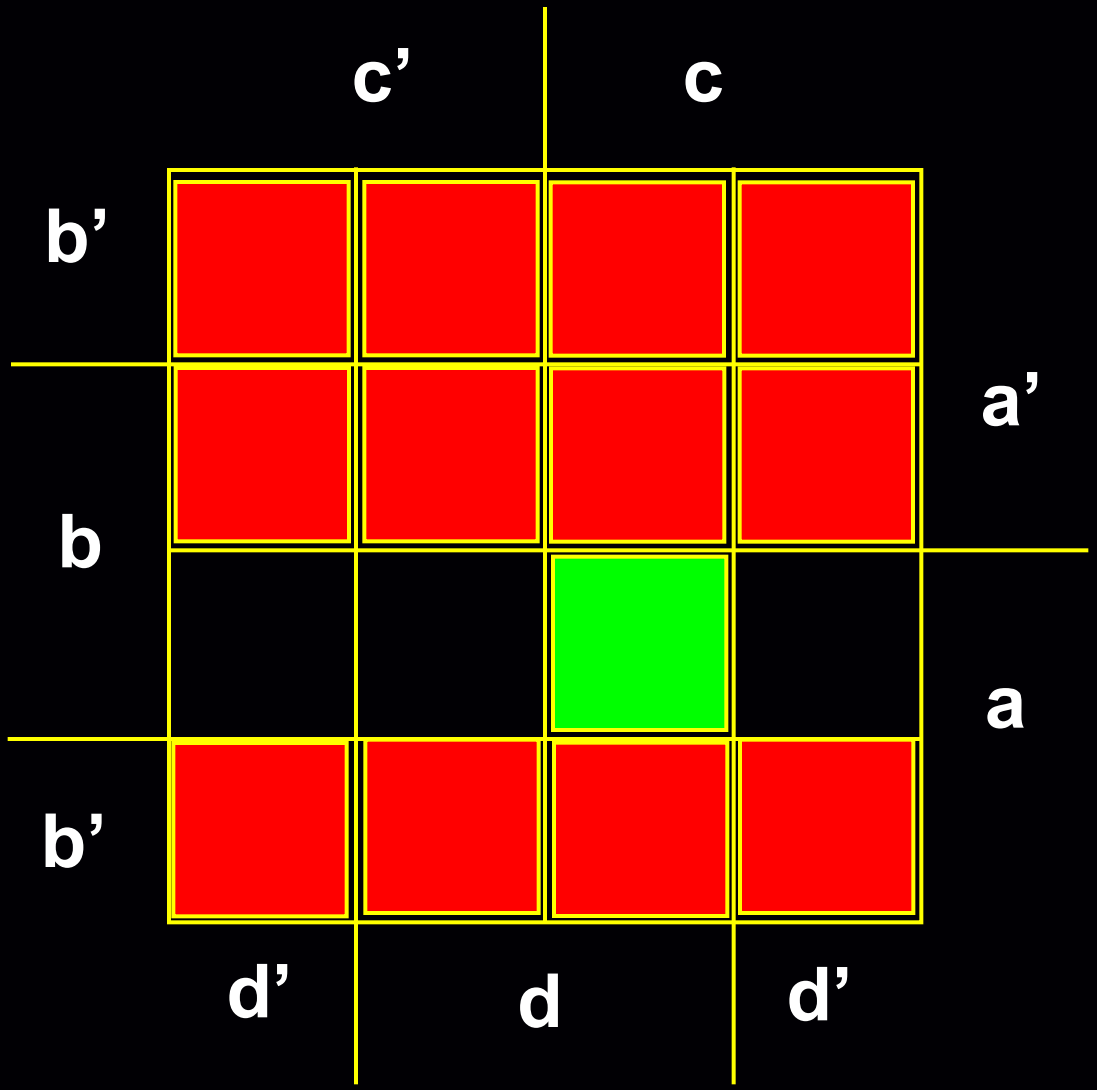
With conflict-driven learning

Conflict-Driven Non-Chronological Backtracking --- Completeness

- Although non-chronological, with conflict-driven learning, SAT search can be guaranteed to be complete
- SAT search is not a binary decision tree anymore…
  - Becomes a decision stack
  - Conflict
    → Learned clause (gate)
    → Indicate where to backtrack
    → Learned implication



Conflict-Driven Learning
= Constraint Refinement Process



Decision: a = 0
Decision: b = 0
Decision: c = 0
conflict!!
Learned: (a + c)
Backtrack: c = 0, b = 0
Implied: c = 1
conflict!!
Learned: (a)
Implied: a = 1
Decision: b = 0
conflict!!
Learned: (b)
Implied: b = 1
Implied: c = 1, d = 1
SAT!!

Conflict-Driven Non-Chronological Backtracking --- Completeness

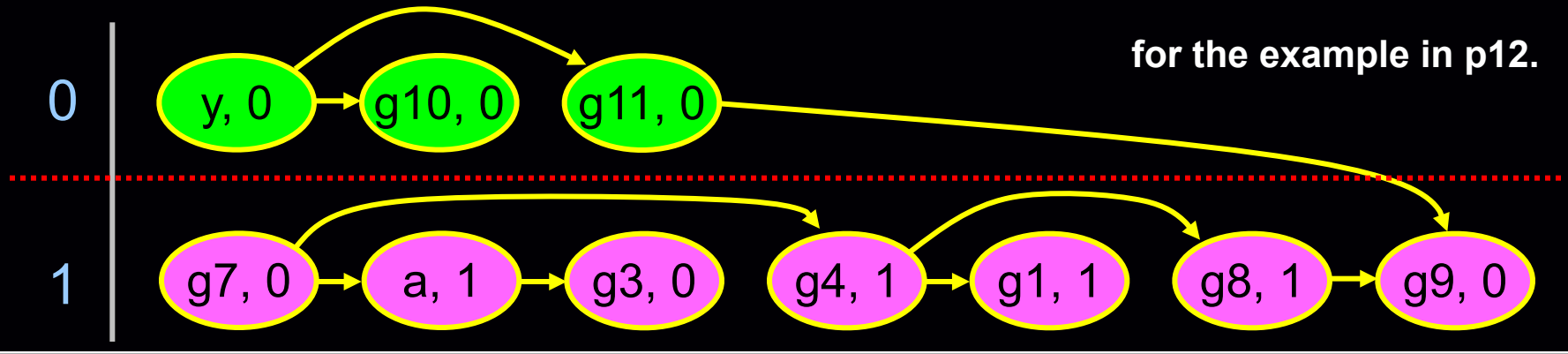
- Branch-and-bound algorithm for Constraint Satisfiability Problem (CSP) becomes a "constraint refinement process"

➔ Search region is gradually narrowed down

➔ At the end, either becomes empty, or finds the solution !!

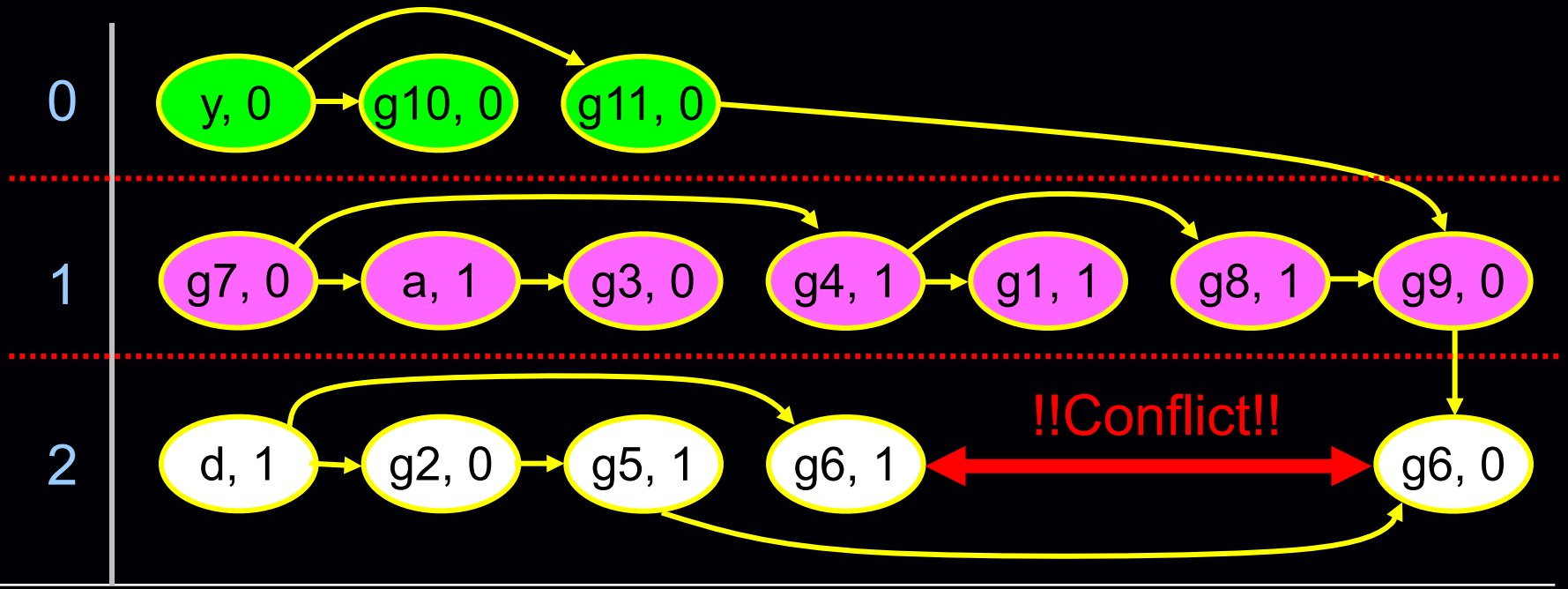
A Closer Look at the Implication Graph (a conceptual implementation)

- Implications are grouped into different decision levels
  - Level 0: target imp; constants
  - Level 1+: decisions
- Node (gate, value): implications
- Incoming edge(s) of a node: implication sources (reasons)
  - The nodes with no incoming edges are called “root implication nodes”
  - There should only be ONE root implication node for each decision level >= 1 (which is the decision in that level)

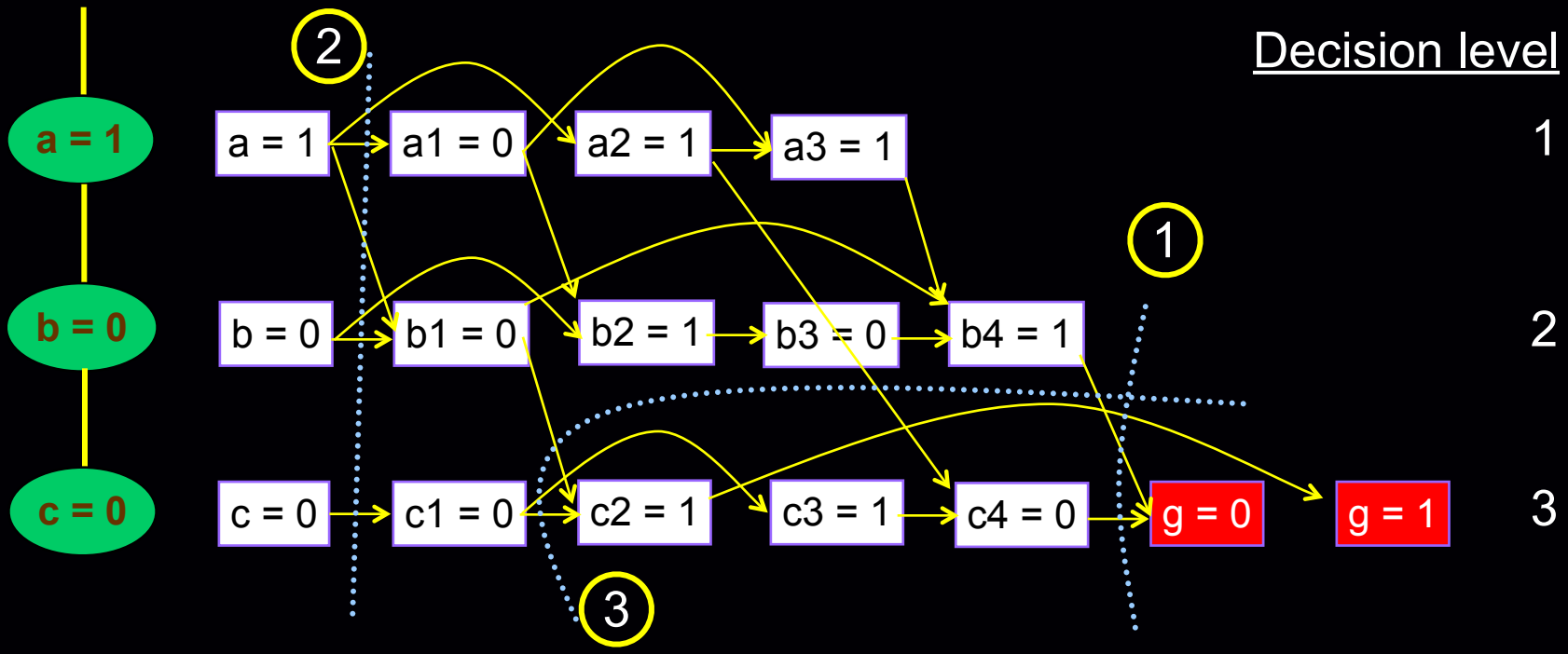


Conflict Analysis

- When we encounter a decision conflict, we want to figure out the causes so that ---
  1. Try to avoid the same conflict
  2. Backtrack as many decisions as possible

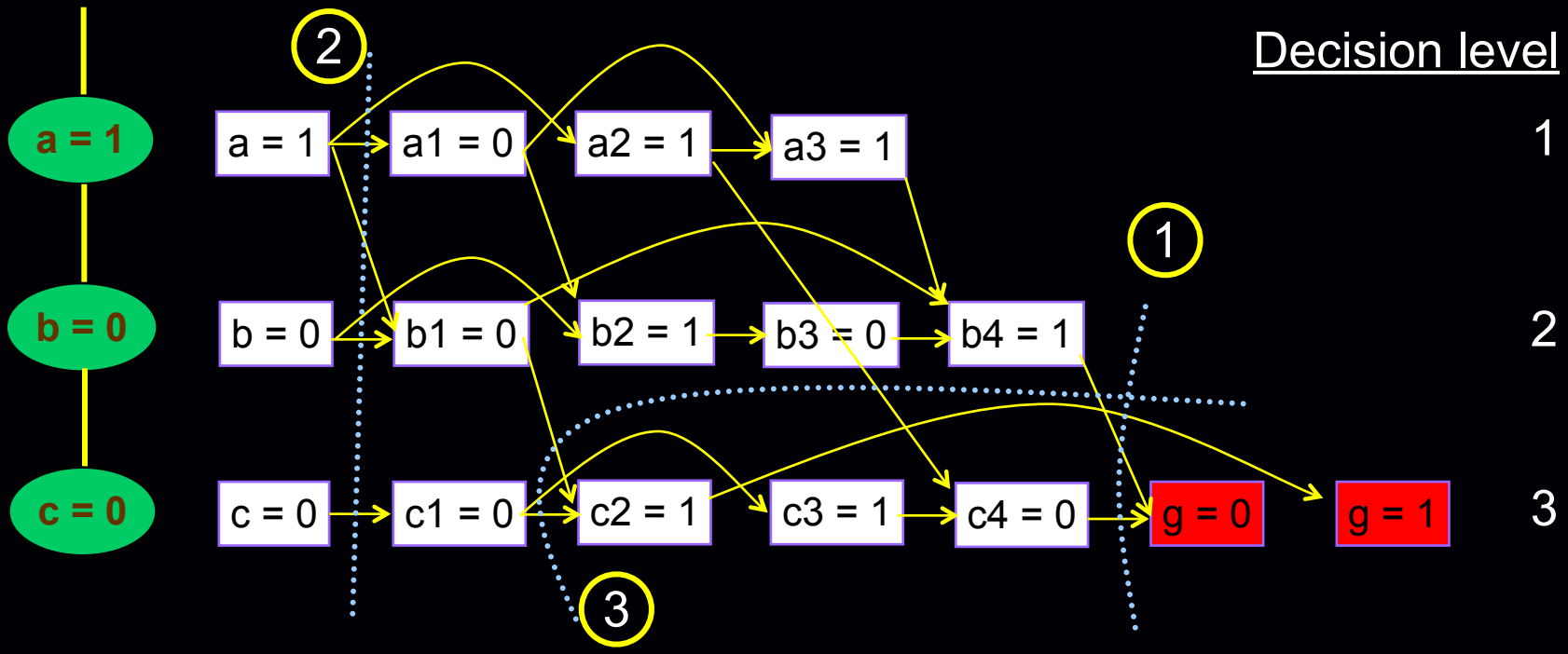


Conflict Analysis



1. Try to avoid the same conflict
   - Starting from the conflict implications (g = 0) & (g = 1), backward trace their implication sources
   - (An informal explanation) Any cut in the implication graph defines a set of conflict sources
   - Add a constraint for the conflict sources to prevent the conflict from happening again

Conflict-Driven Learning



- Add a constraint to prevent the same conflict
  1. b4 && c2 && c4' = 0; ➔ (b4' + c2' + c4)
  2. a && b' && c' = 0; ➔ (a' + b + c)
  3. b4 && a2 && b1' && c1' = 0; ➔ (b4' + a2' + b1 + c1)

Which constraint is the best to add?

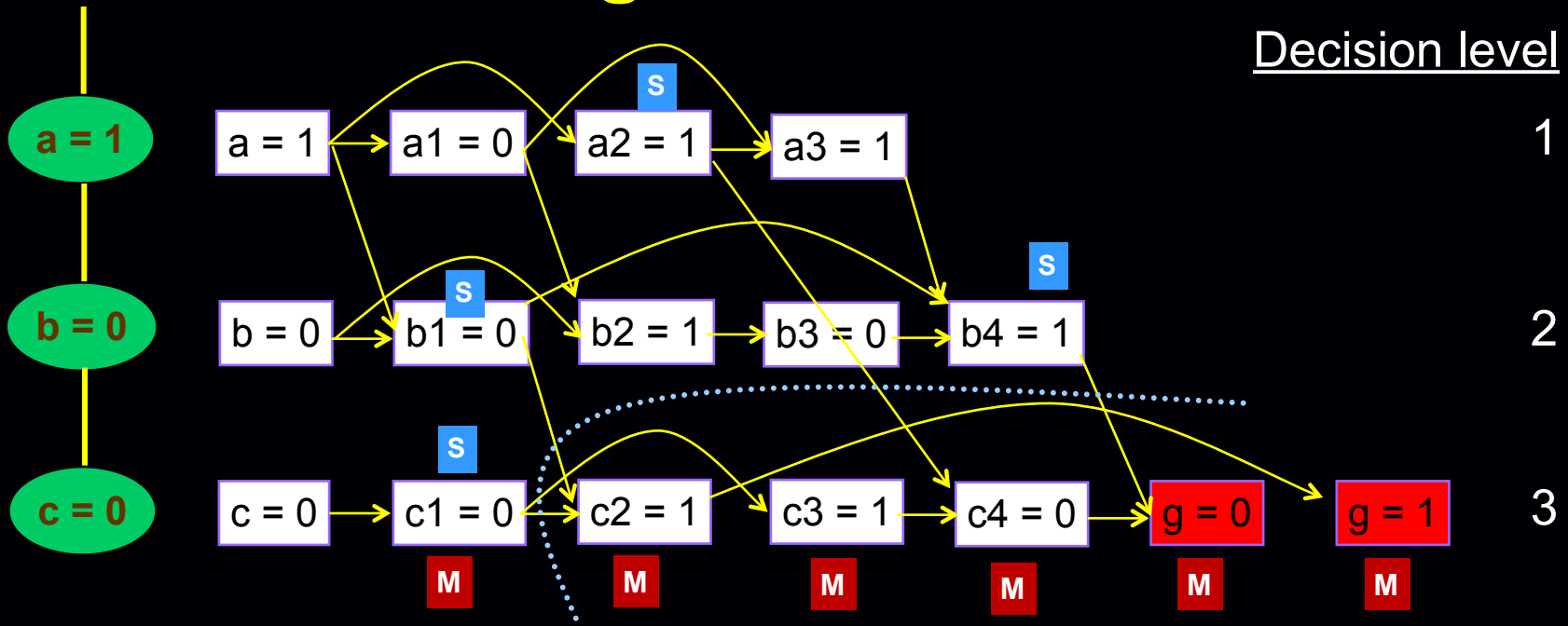
- [Zhang, *et al*, ICCAD 2001] Experiment shows that "first-UIP" (1st-UIP) is the best
  - UIP: Unique Implication Point
    - In a cut that there is only one node (i.e. UIP) in the last (where conflict happens) decision level (why UIP cut?)
    - Starting from the conflict gate, the first encountered UIP is namely first UIP
    - The cut with only decision nodes is the last-UIP
      - In the previous example, (2) is the last UIP, and (3) is the first UIP

Pseudo-code to find the first UIP

```
conflictAnalysis(imp0Src, imp1Src) {
  int nMarked = 0;
  for_each_imp(imp, imp0Src)
    checkImp(imp, nMarked,conflictSrc);
  for_each_imp(imp, imp1Src)
    checkImp(imp, nMarked, conflictSrc);
  for_each_imp_rev(imp, lastDLevel) {
    if (!imp.isMarked()) continue;
    if (--numMarked == 0) {// UIP found!!
      conflictSrc.push_back(imp);
      break;  // ready to return
    }
    imp.unsetMark();
    for_each_imp_src(imp_src, imp) {
      checkImp(imp_src, nMarked,
                        conflictSrc);
    }
  }
  for_each_imp(imp, conflictSrc)
    imp.unsetMark();
  return conflictSrc;
}

checkImp(imp, nMarked, conflictSrc) {
  if (imp.isMarked()) return;
  imp.setMark();
  if (!imp.isLastDecisionLevel())
    conflictSrc.push_back(imp);
  else ++numMarked;
}
```

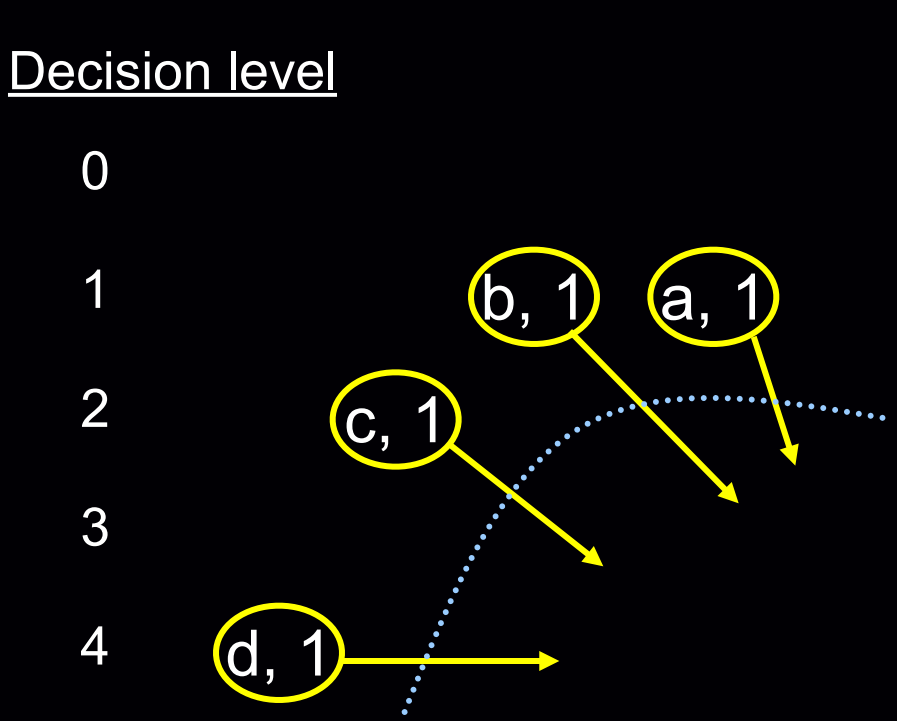
Linear-Time Algorithm to Find First UIP



- Start from (g = 0), (g = 1) // #Marks = 2
- Unmark (g = 1), mark (c2 = 1) // #Marks = 2
- Unmark (g = 0), mark (c4 = 0), add (b4 = 1) // #Marks = 2
- Unmark (c4 = 0), mark (c3 = 1), add (a2 = 1) // #Marks = 2
- Unmark (c3 = 1), mark (c1 = 0) // #Marks = 2
- Unmark (c2 = 1), add (b1 = 0) // #Marks = 1
- Find first UIP: (c1=0), conflict sources: { (c1=0), (b1=0), (a2=1), (b4=1) }

UIP for Non-chronological Backtracking

- Since in UIP cut there is only one node with the last decision level…
- As we add a constraint for the UIP cut ---



Constraint

(a && b && c && d) = 0

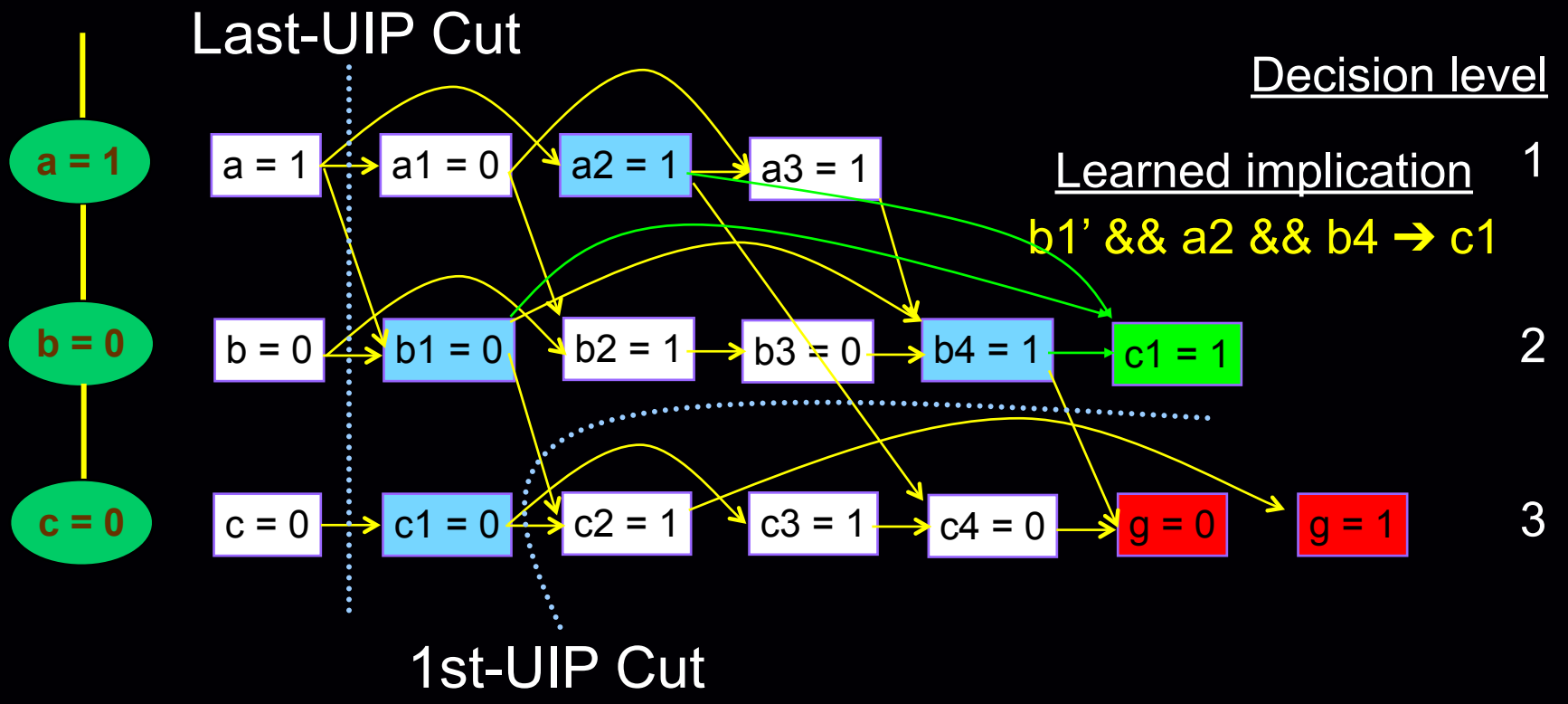
If we backtrack to the second-to-the-last decision level (i.e. c)

1. { a, b, c } still have the original implications

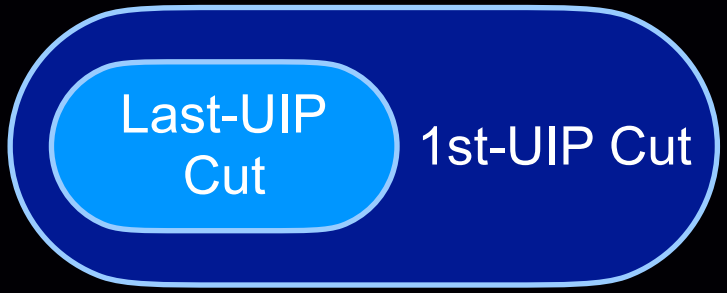
   (a && b && c) ➔ d'

2. d can be implied with the opposite value at the max level above

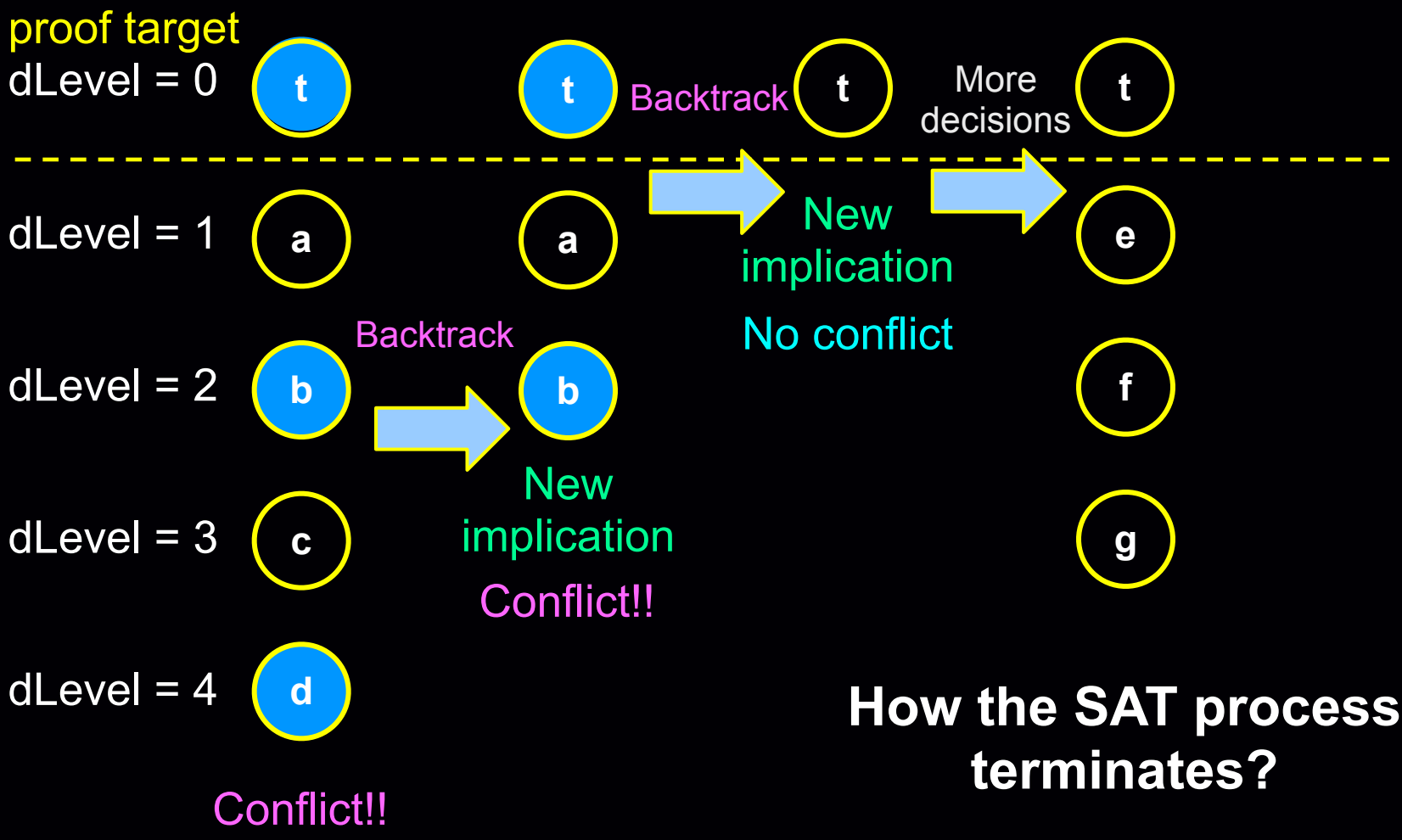
Why is first UIP the best cut?



Last-UIP Cut => 1st-UIP Cut



Conflict-Driven Non-Chronological Backtracking --- Algorithm

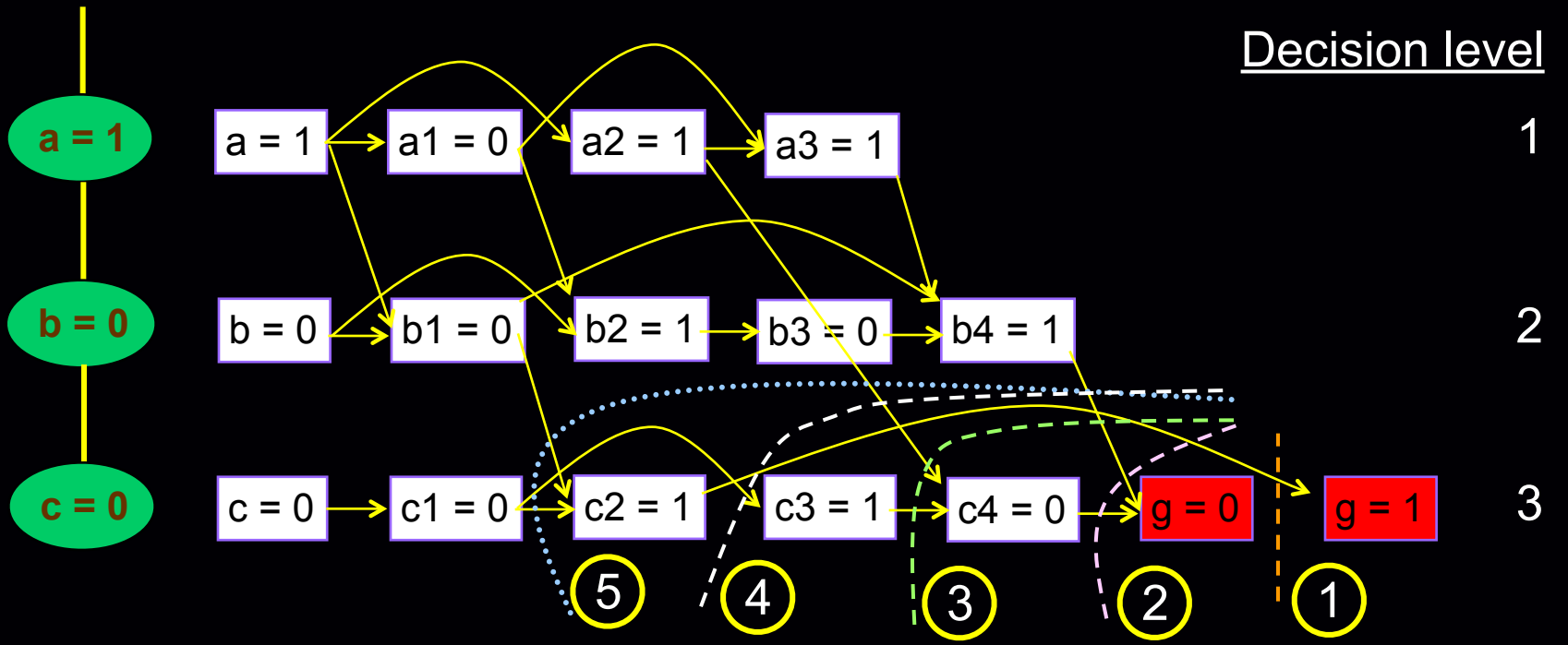


How the SAT process terminates?

Conflict-Driven Non-Chronological Backtracking --- Algorithm

1. When conflict occurs, check if the conflict level == 0 (implication level for the SAT target)
   a) If yes, return *unsatisfiability* (Why?)
   b) Else, continue to 2
2. Find the 1st-UIP cut as the conflict causes
3. Backtrack to the max decision level of the nodes other than UIP in the cut
4. The UIP gate will be implied with the opposite value
5. Perform the new implication
6. If conflict, go to 1, else continue for the next decision

Implication graph, resolution, and learning



(1): (c2' + g)

(2): (b4' + c4 + g') → (b4' + c2' + c4)

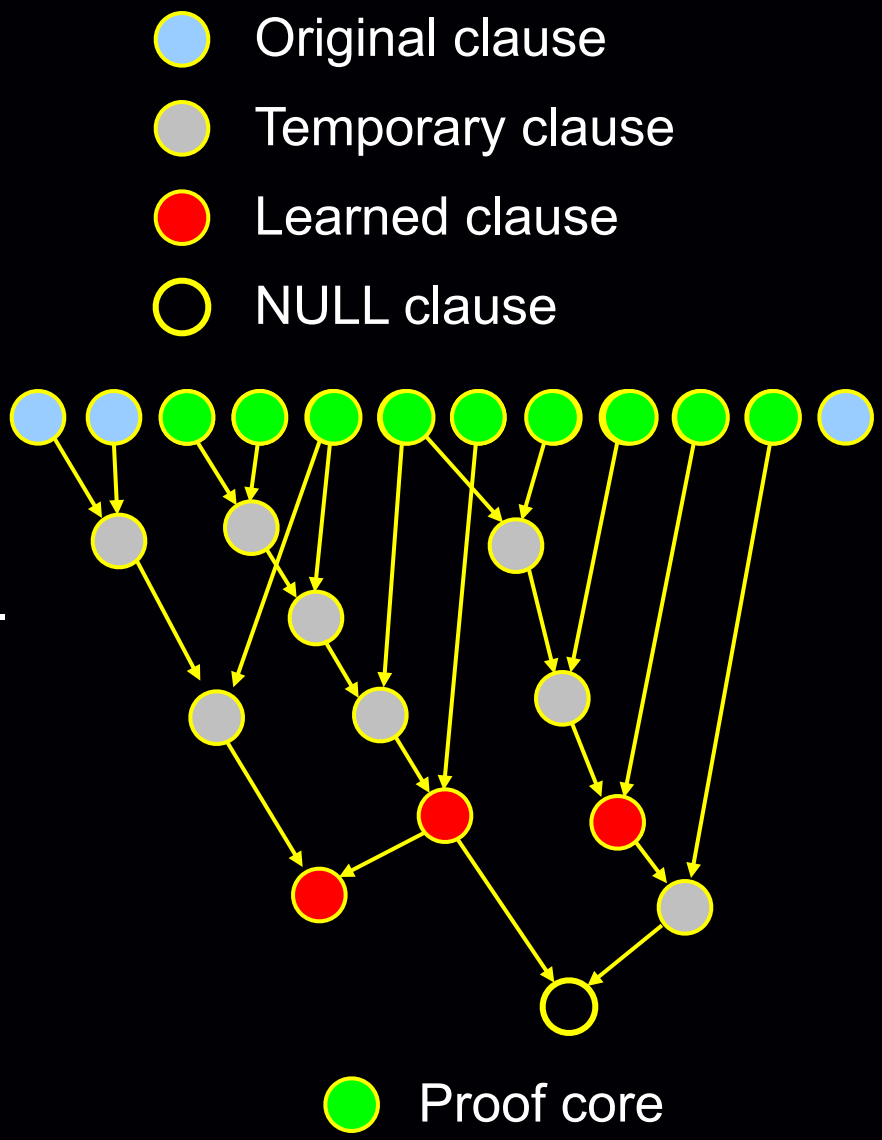
(3): (a2' + c3' + c4') → (a2' + b4' + c2' + c3')

(4): (c1 + c3) → (a2' + b4' + c1 + c2')

(5): (b1 + c1 + c2) → (a2' + b1 + b4' + c1)

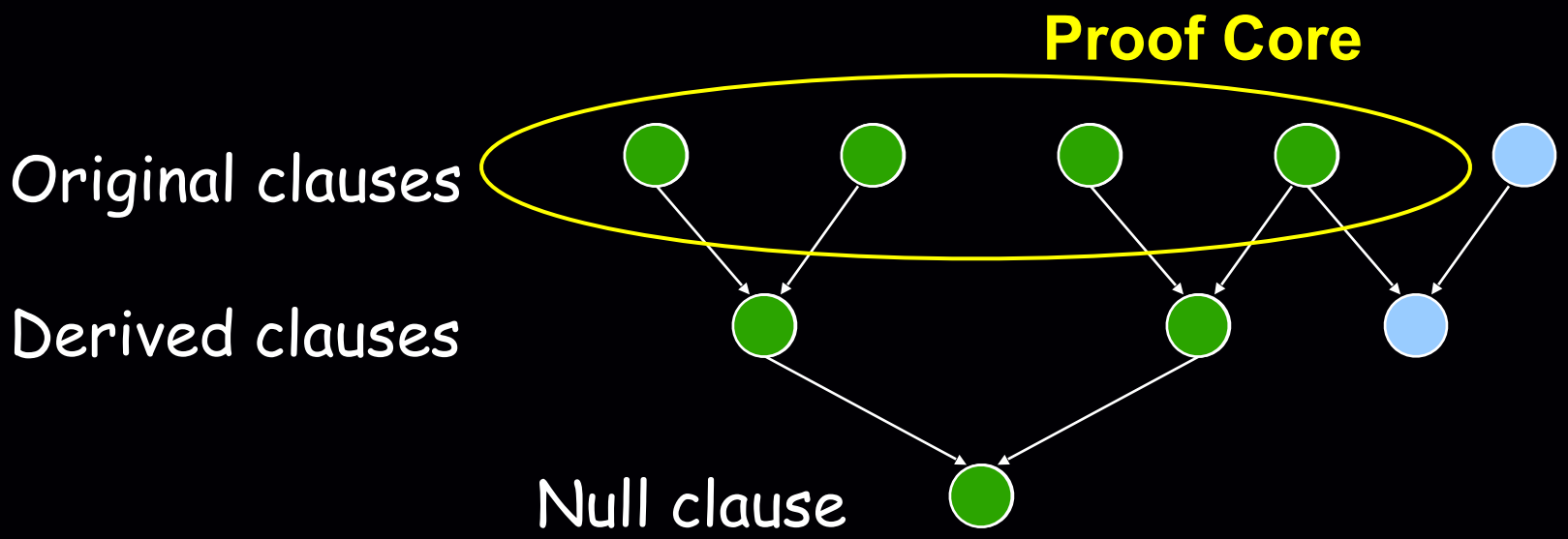
Resolution Graph

- A conflict is encountered
  - A learned clause is generated
- More conflicts are resolved...
- A conflict is encountered in decision level 0
  - Problem is proven UNSAT



Refutation / Proof Core of a SAT Problem

- Remember: Resolution-based SAT?
  - A problem is proven UNSAT if the resolution steps end up in a NULL clause
  - Tends to learn EVERYTHING!
- Refutation = a proof for the null clause
  - Also called "proof core" or "UNSAT core"
  - Record a DAG containing all resolution steps performed during conflict clause generation.
  - When null clause is generated, we can extract a proof of the null clause as a resolution DAG.



The validity of learned information and incremental SAT

- Note that, learned clause is a resolution of clauses that are involved in the implication process.
  - As long as these clauses are still in the proof database, the learned information is always valid.
- Incremental SAT
  - (For example) Proving two properties in a circuit --- the learned information obtained in proving one property can be reused in proving another.
  - (Challenge) What if some of the clauses or variables are deleted?

What affect the SAT efficiency?

1. Decision order

We will cover these...

2. Logic implication (Boolean Constraint Propagation, BCP)

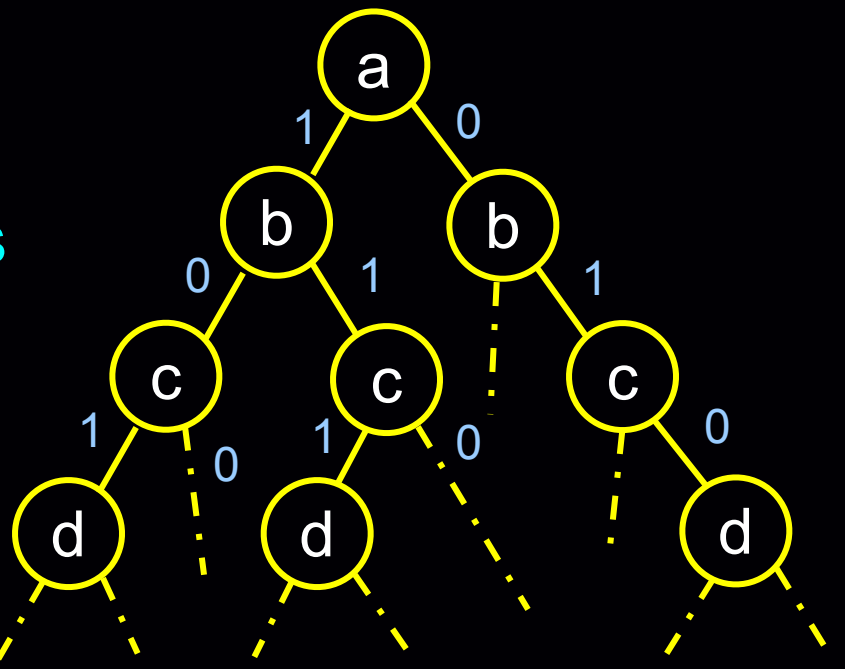
3. Various learning techniques

4. Database simplification

Impact of Decision Ordering

- Decision ordering: the order of gates that the corresponding decisions are made
  1. Order of gates
  2. Decision values

  ➔ Good and bad decisions can lead to exponential difference (e.g. 2^{10} vs. 2^{50})



- (Think) Does the decision value matter? (i.e. should we decide on ‘1’ or ‘0’ first?)

Static Decision Ordering

- Decision order and values are pre-computed in the beginning and remain unchanged

1. Topological
   - Depth-first
   - Breadth-first
   - Guided by gate types
2. Probability-based
   - Controllability / Observability
   - Signal probability
   - (Weighted) Random
3. Influence-based
   - Literal count
   - #fanins / #fanouts
   - Influence of implications

Dynamic Decision Ordering

- Decision order and values are dynamically determined based on current implication values, justification frontier, etc.
  - Use similar criteria as static method
  - But can mix different rules dynamically
- Pros
  - May lead to better decisions
  - Avoid useless decisions
- Cons
  - Overhead in computing dynamic ordering may be high
  - Effectiveness sometimes is hard to predict

➔ However, experiences show that the best is:

1. Has a good initial decision ordering
2. Adaptively adjust the decision order after a certain amount of backtracks

zChaff's Variable State Independent Decaying Sum (VSIDS) Decision Heuristic

(1) Each variable in each polarity has a counter, initialized to 0.

(2) When a clause is added to the database, the counter associated with each literal in the clause is incremented.

(3) The (unassigned) variable and polarity with the highest counter is chosen at each decision.

(4) Ties are broken randomly by default, although this is configurable

(5) Periodically, all the counters are divided by a constant.

Zhang, et al, DAC 2001

Berkmin – Decision Making Heuristics

E. Goldberg, and Y. Novikov, "BerkMin: A Fast and Robust Sat-Solver", *Proc. DATE* 2002, pp. 142-149.

- Identify the most recently learned clause which is unsatisfied
- Pick most active variable in this clause to branch on
- Variable activities
  - updated during conflict analysis
  - decay periodically
- If all learnt conflict clauses are satisfied, choose variable using a global heuristic
- Increased emphasis on "locality" of decisions

More decision heuristics...

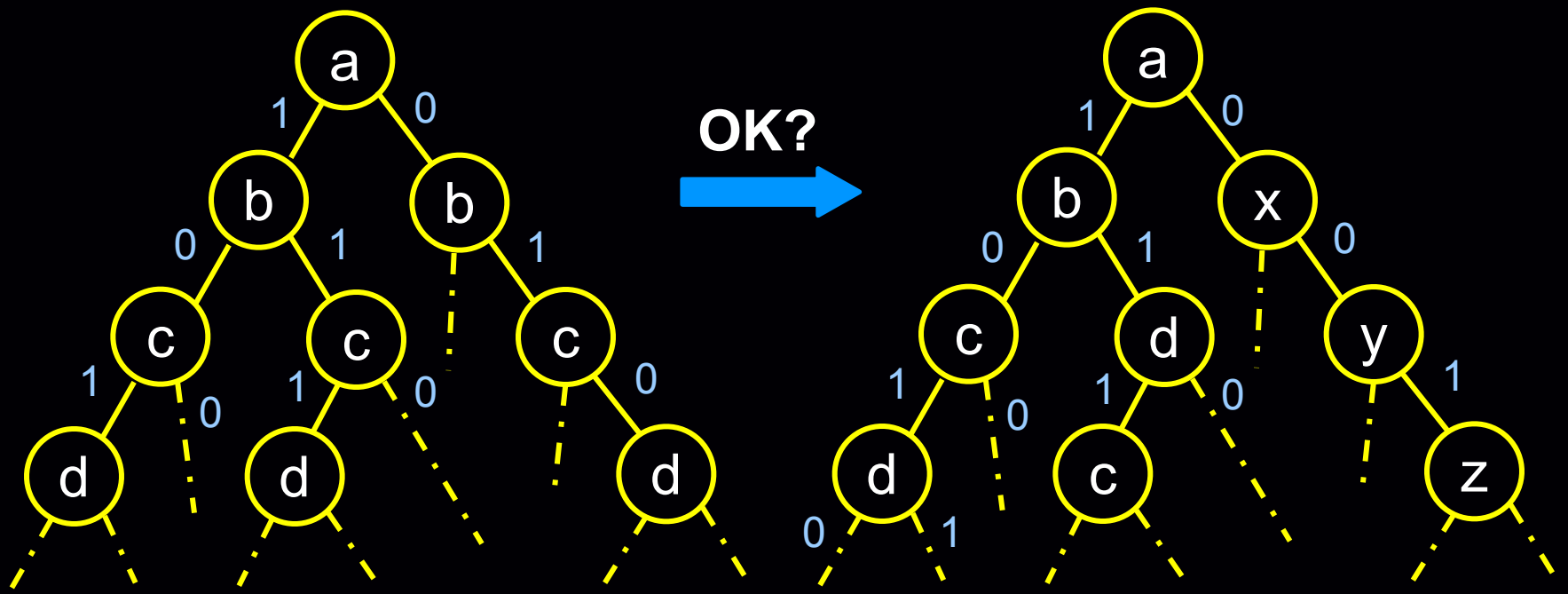
- Variable Move-To-Front (VMTF)
- Clause Based Heuristic (CBH)
- Resolution Based Scoring (RBS)
- ...

- In general, there is no single decision heuristic that works for every case.

  ➔ How to adaptively move to a good decision heuristic may be the winner...

A closer look at binary decision tree

Should the decision orderings on all branches be the same?



Remember when we talked about conflict-driven learning,

we mentioned that

by adding learned clauses

we can do non-chronological backtracking, while still achieve complete proof,

and the decision tree becomes a decision stack!

How??

The Constraint Refinement Process

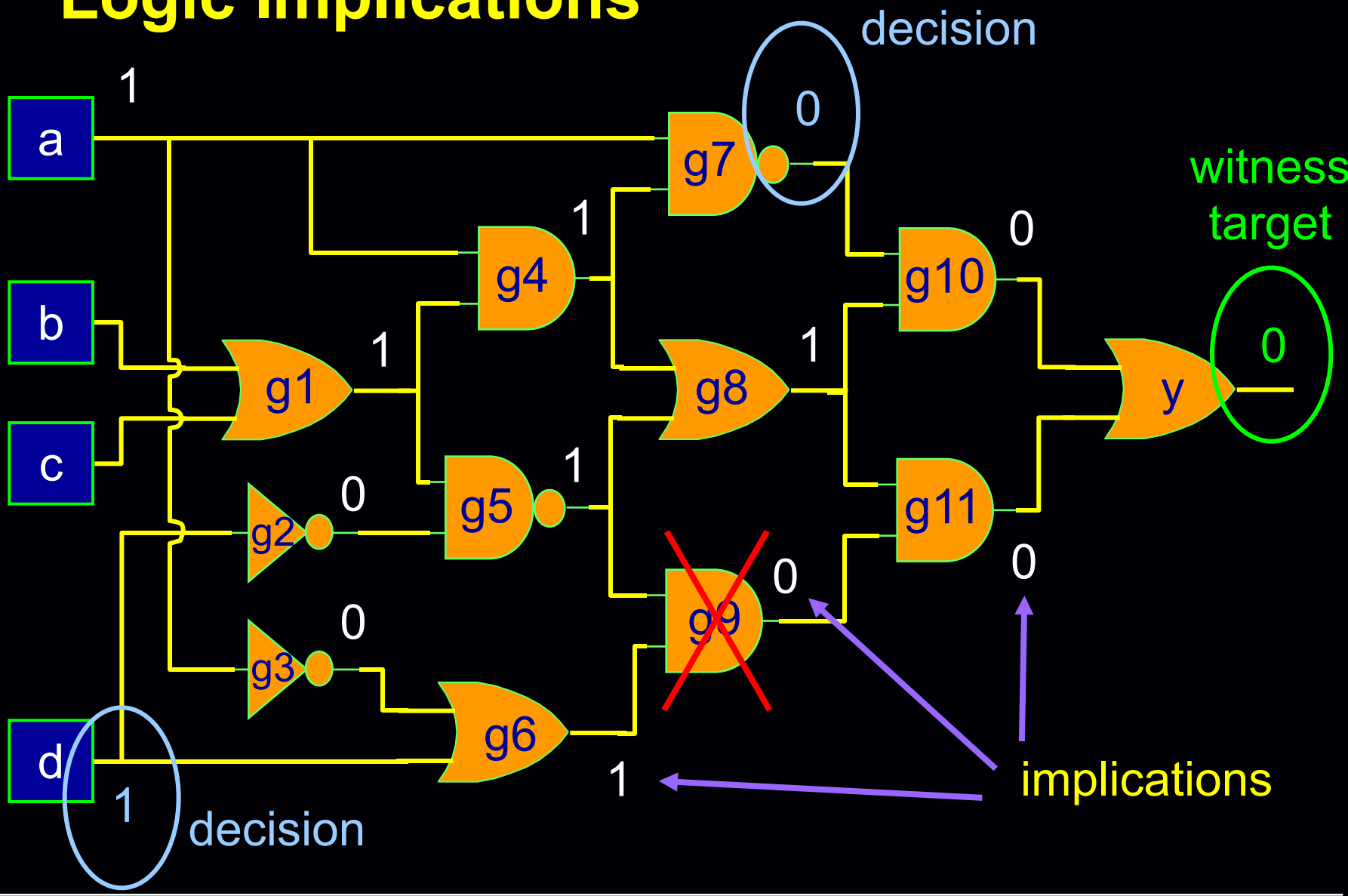
- Search region is gradually narrowed down by the learned constraints
- Learned information is universally true
  - Independent of the target implication, only related to the circuit function
  - The proof efforts between different properties can be shared

    ➔ Incremental SAT
- Decision process can “restart” any time any where!!
  - Can use different decision ordering to explore different area in the decision tree
    - Previous efforts will not be wasted

What affect the SAT efficiency?

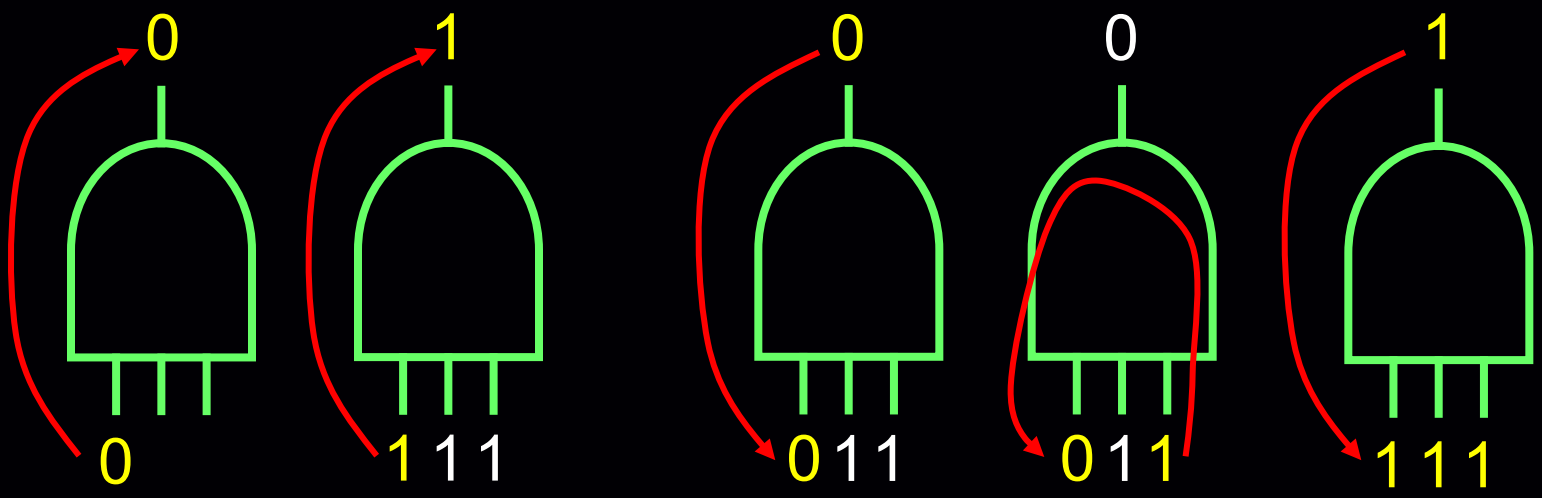
1. Decision order
2. Logic implication (Boolean Constraint Propagation, BCP)
3. Various learning techniques
4. Database simplification

Logic Implications



Logic Implications

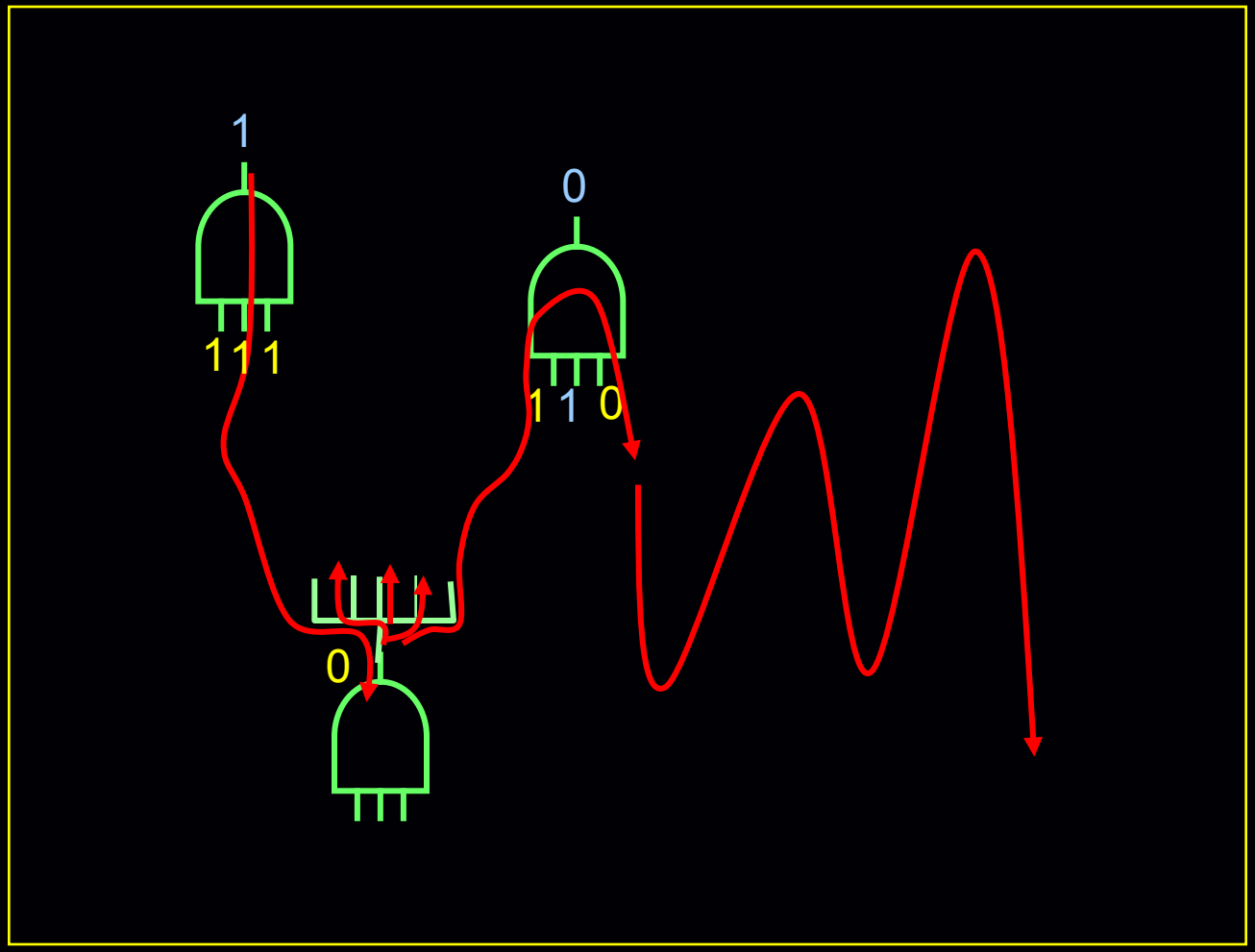
- Also called “Boolean Constraint Propagation” (BCP)
- Take over 90% of the SAT runtime
- Imply values to other gates in both forward and backward directions



Forward implications

Backward implications

“Omni-directional” in a netlist

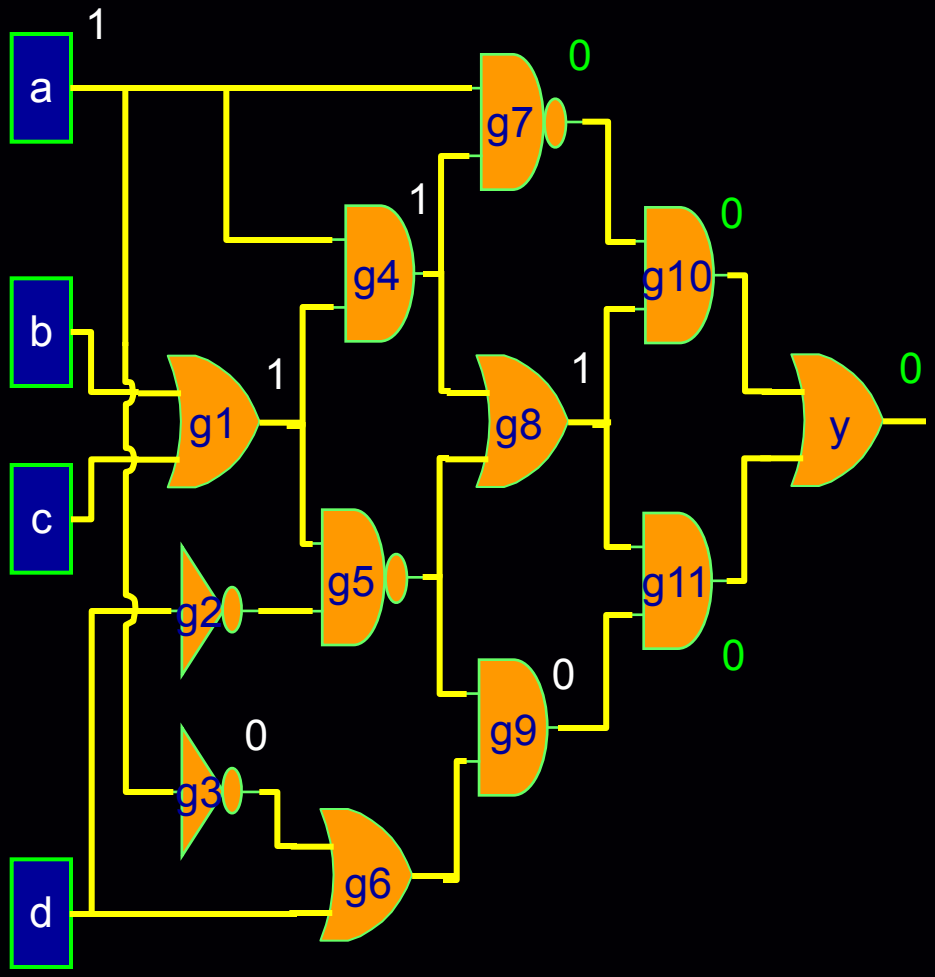


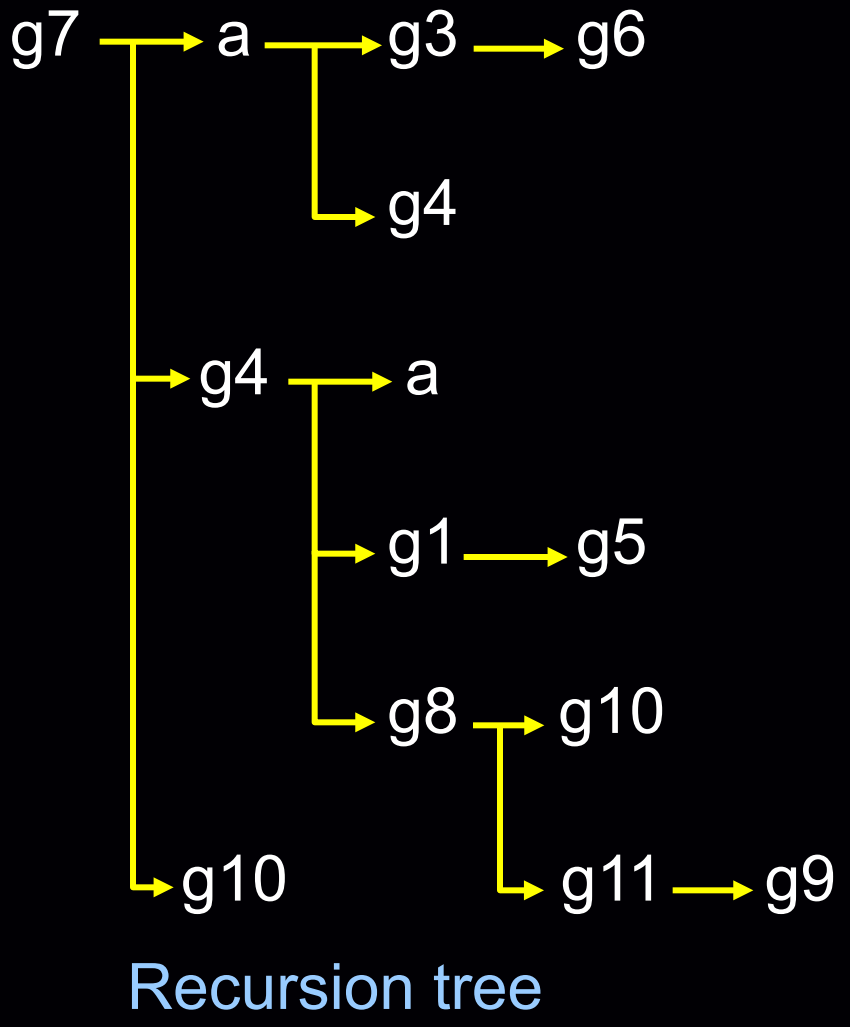
How to schedule and evaluate these implications??

Take 1: Logic Implication by Recursion

```
1.  bool logicImplication(Gate* g, value v)
2.  {
3.      if (g->hasConflictValue(v)) return false;
4.      if (g->getValue() != 'x') return true;
5.      // g has value 'x' → can be implied
6.      g->setValue(v);
7.      // fanins and fanouts that may be
8.      //    implied by g
9.      List scheduleList = checkImplication(g);
10.     for_each_gate_value(scheduleList, gg, vv)
11.        if (logicImplication(gg, vv) == false)
12.           return false;
13.     return true;
14. }
```

Logic Implication by Recursion Example

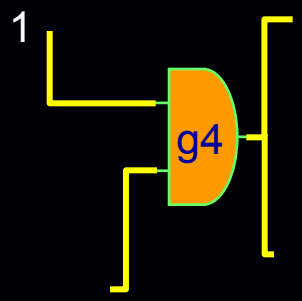




Recursion tree

Logic Implication by Recursion

- Pros
  - Easy to implement
- Cons
  - A gate may be put into **`scheduleList`** many times by different neighboring gates
    1. Value has already been implied, or
    2. Too early to evaluate

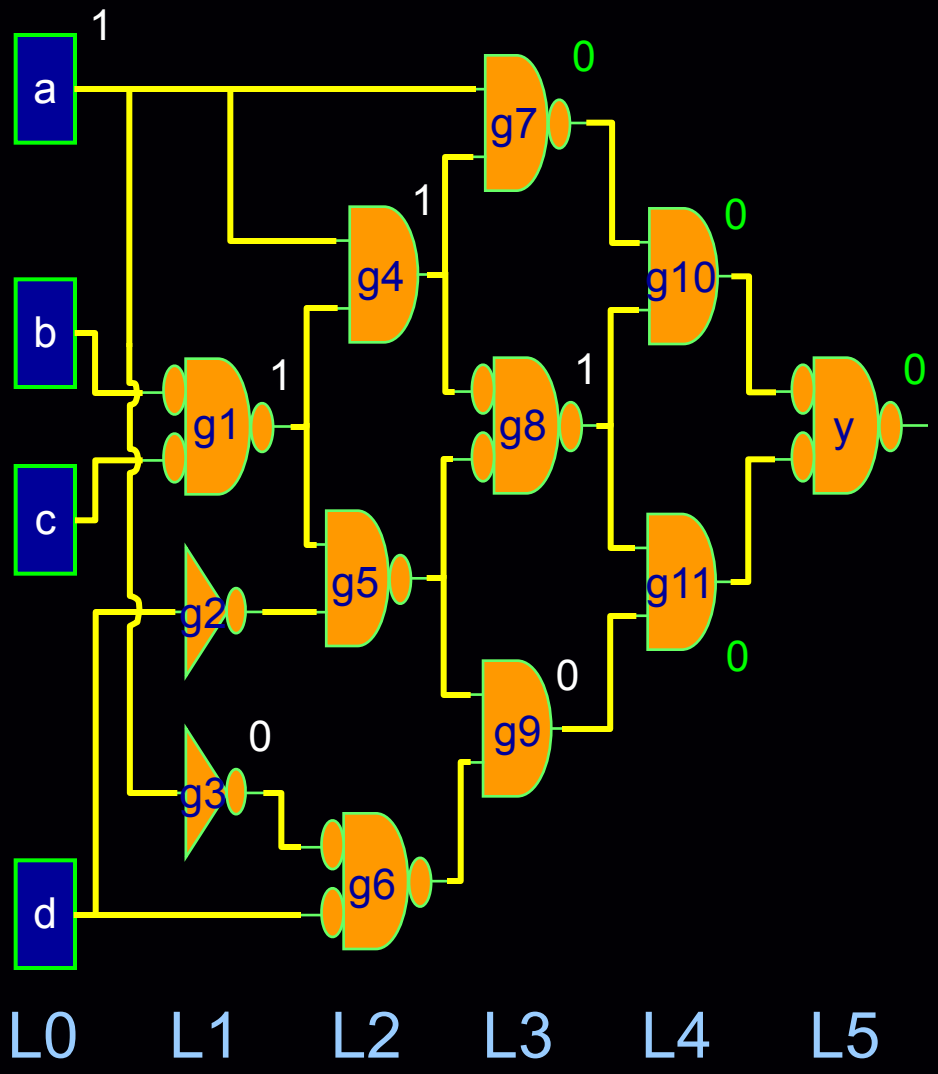


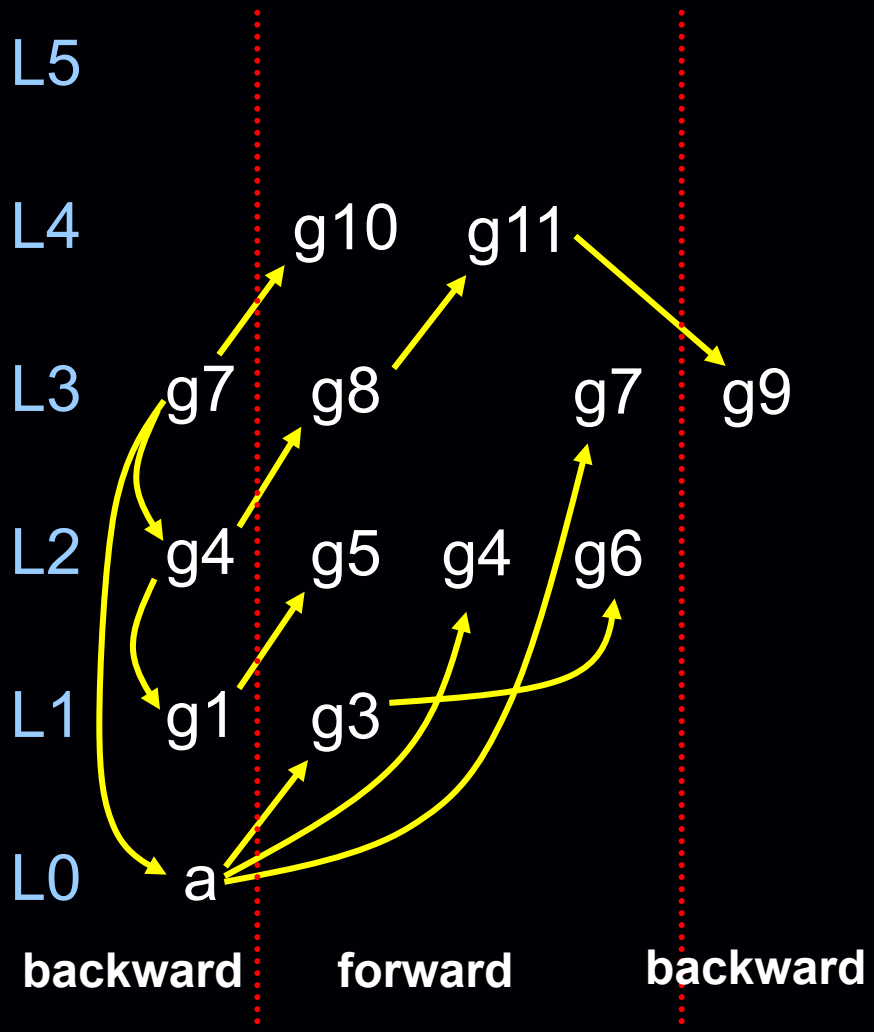
Take 2:
Logic Implication by Topological Order

- Preprocess
  - All gates are sorted topologically
  - Each gate has a field "_level" = max(fanin->_level) + 1

```
1.  bool logicImplication()
2.  {
3.      while (more scheduled gates)
4.          for scheduled gates (level = maxLevel to 0)
5.              apply backward implications and
6.                  schedule forward implications
7.              return false if any conflict;
8.          for scheduled gates (level = 0 to maxLevel)
9.              apply forward implications and
10.                 schedule backward implications;
11.             return false if any conflict;
12.     return true;
13. }
```

Logic Implication in Topological Order Example



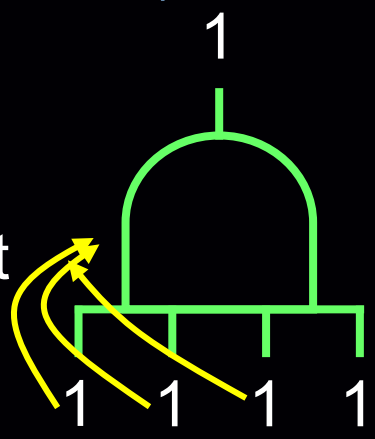


Logic Implication in Topological Order

- Pros
  - Minimize the number of scheduling
  - Each gate is checked only once in each forward or backward implication routine
- Cons
  - A gate may still be checked more times than necessary
    1. Value has already been implied, or
    2. Too early to evaluate

Fruitless Implication Checking

- Checking if a gate can be implied, but usually no implication
- For example, consider the all-1's forward implication of an AND gate
  - Only the last '1' can trigger the forward implication
    - The first (n-1) checks are useless
  - Worse case: for n-input AND gate
    - Need to check (1 + 2 + …+ n) times of fanins
    - $O(n^2)$



put to scheduleList

Trying to avoid fruitless implication checking

- Use a counter to record how many ‘x’ are in the fanins of a gate (e.g. all 1's for an AND gate)
  - Decrement by 1 when fanin is implied
  - Increment by 1 when fanin value is backtracked
  - When no ‘x’ fanin (x count = 0) → forward implication

- Although this can avoid fruitless implication checking, yet the overhead in maintaining the counts could be an overkill…
  - ‘x’ count could be: 5 → 4 → 3 → 4 → 5 → 4 → 5...

Can we do better?

Why should we check the pins when there are more than 1 x's?

But, how do we know if there are more than 1 x's?

Logic Implication for CNF-Based SAT

- Refresh:
  - Boolean function is represented as a CNF (i.e. Product of Sum, POS format)
    - Can arbitrary Boolean function be converted into CNF?
  - e.g. (a+b+c)(a'+b'+c)(a'+b+c')(a+b'+c')
  - To be satisfied, all the clauses should be '1'

Logic implication for CNF-based SAT is simple ---

- If a literal in a clause gets an implication '1'

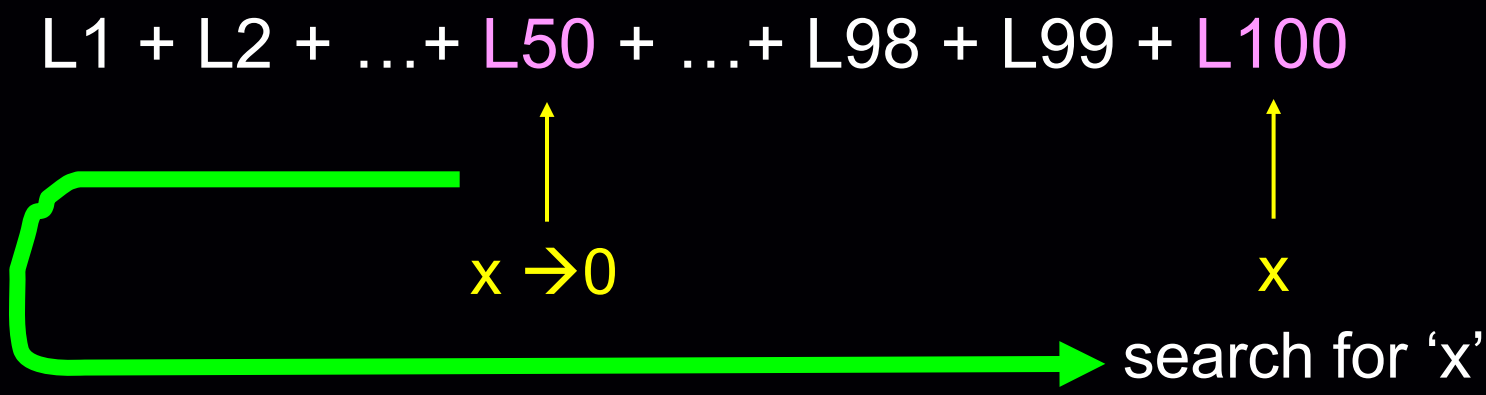
  ➔ The clause is satisfied
- If a literal in a clause gets an implication '0'

  ➔ Check: how many literals in the clause have unknown value?
  - >= 2 : no operation
  - = 1 : the remaining literal will be implied '1' (unit clause rule)
  - = 0 : the clause is evaluated to '0' ➔ a conflict !!

2-Watched-Literal Algorithm

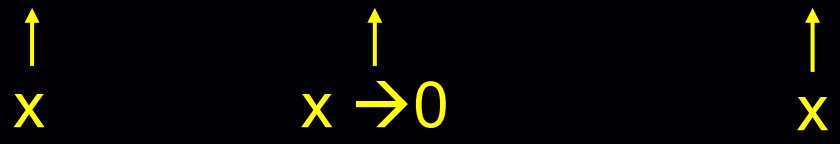
H. Zhang, SATO, CADE 97; M. Moskewicz *et al*, Chaff, DAC 2001

- For each clause, keep 2 pointers on 2 literals that have "non-0" values
  - If any watched literal gets implication '0'
    - Scan in the clause for another literal with "non-0" value
    - If found, update the watched literal pointer
      Else, if the other watched literal is '0' => return Conflict
      - Else, imply the other watched literal with value '1'



Watching Clauses of a Literal

L1 + L2 + …+ L50 + …+ L98 + L99 + L100



- Let L50 and L100 be the watched literals of the clause
- What happens if L98 gets an implication x -> 0? Update watch?
  - No action
  - But how does this clause know that it doesn't have to update the watched literals?
  - Check if L98 is a watched literal?

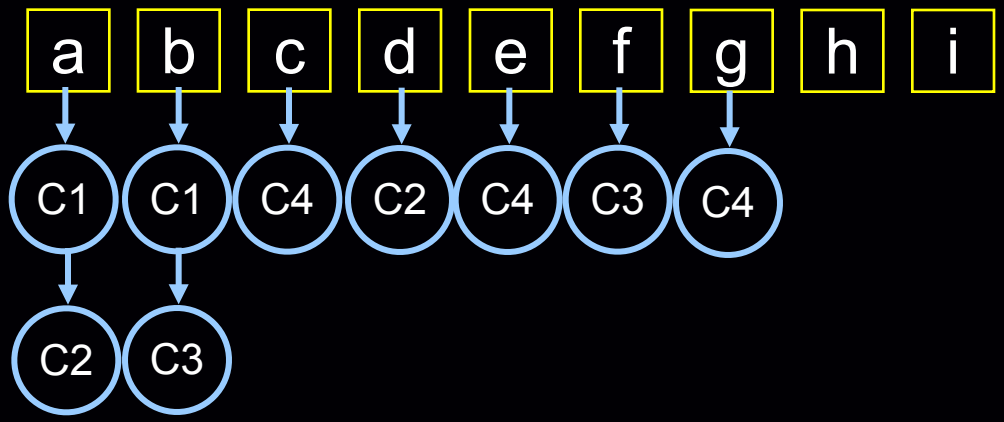
No!! By keeping a list of "watching clauses" for each literal (who is watching me), when a literal is implied, it only has to update the clauses that watch this literal

2-Watched-Literal Algorithm Example

Each clause stores:
2 watched literal pointers

Each literal stores:
A list of watching clauses

C1: (a + b + c + d)
C2: (a + d + e + f + g)
C3: (b + f
C4: (c + e + g + h + i)



c ← 0

- Update watched literal pointer for C4 (for example, to ‘g’)
- Erase c’s watching-clause list
- Add ‘C4’ to g’s watching-clause list

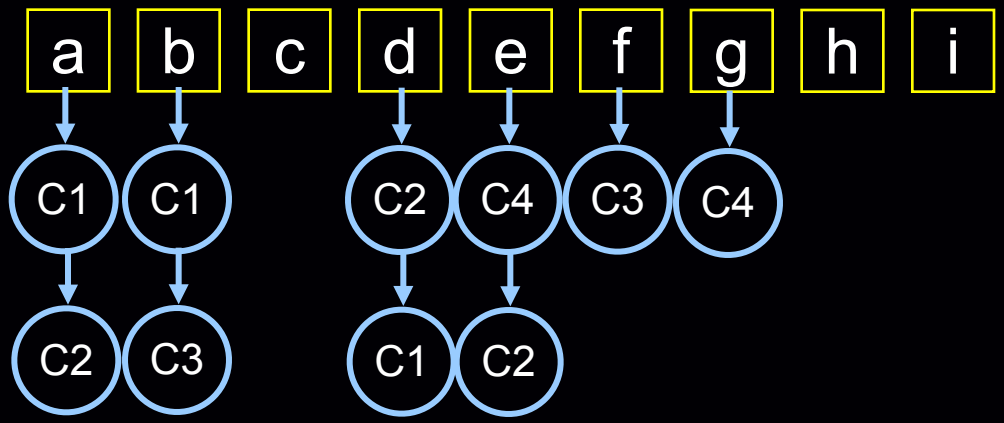
[Note] Don’t need to check ‘C1’

2-Watched-Literal Algorithm Example

Each clause stores:
2 watched literal pointers

Each literal stores:
A list of watching clauses

C1: (a + b + c + d)
C2: (a + d + e + f + g)
C3: (b + f
C4: (c + e + g + h + i)



a ← 0

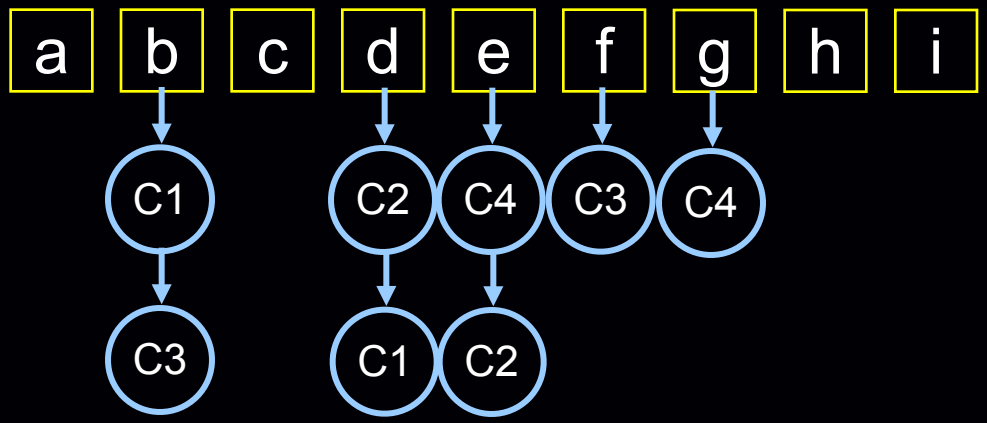
- Update watched literal pointer for C1 (only choice, to ‘d’)
- Update watched literal pointer for C2 (for example, to ‘e’)
- Erase a’s watching-clause list
- Add ‘C1’ to d’s and ‘C2’ to e’s watching-clause lists

2-Watched-Literal Algorithm Example

Each clause stores:
2 watched literal pointers

Each literal stores:
A list of watching clauses

C1: (a + b + c + d)
C2: (a + d + e + f + g)
C3: (b + f)
C4: (c + e + g + h + i)



b ← 0

- No more unknown literal for C1 : d = 1
- No more unknown literal for C3 : f = 1

[Note] No change on watched literals

zChaff's improvements on the 2-watch scheme

- In the original 2-watch scheme paper (SATO[97]), watched literals are restrained to head and tail of the non-zero literals
  - Updates of the watched literals can only move in specific directions

  - When backtracking, need to reset the watched literals to the previous positions
- In zChaff, updates of the watched literals can move in any direction
  - Usually follow a single (forward) direction so that the watched literals will move to the "last implied positions"
  - No need to reset the watched literals when backtracking
  - What's the advantage?

Caching Effect: Reducing from O(n) to amortized O(C)

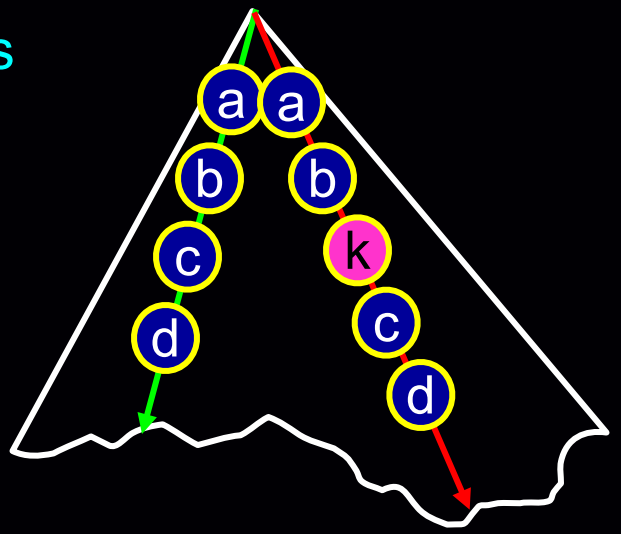
- The fact
  - Most of the time, the decision orderings at different parts of the decision tree are quite similar during a proof (or even from proof to proof)

  ➔ Literals in a clause get the implications almost by the same order every time
- Watched literals

  → point to the last implied literals

  ➔ Don't update watched literals after backtrack.

  ➔ After backtracks, very likely no updates will be evoked for the clauses of the other unwatched literals.



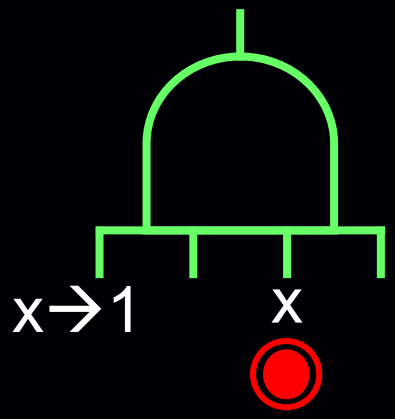
(L1 + L2 + L3 + L4 + L5 + L6)

Logic implication can be very efficient for CNF-based SAT by using “watch” scheme.

Can this idea be applied to circuit-based SAT?

Watched-fanin concept

- (Trial) In the n-input AND gate case, keep a pointer to one of its fanin that has value ‘x’ (watched fanin)
  - If other fanin gets implication ‘1’ ➔ no operation
  - If this watched fanin gets implication ‘1’, try to find another ‘x’ fanin to be new watched fanin
    - If found, update the pointer
    - If not found ➔ imply ‘1’ on the gate output



Sounds good for all-1's forward implication of an AND gate, but what about 0 implication, backward implication, and other types of gates?

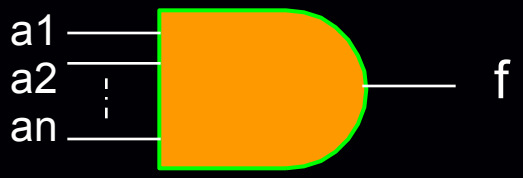
Different watched schemes?

(Could be very complex…)

That's one of the reasons why simpler data structure like CNF-based SAT engine can be more efficient sometimes

Difference between circuit and CNF SAT

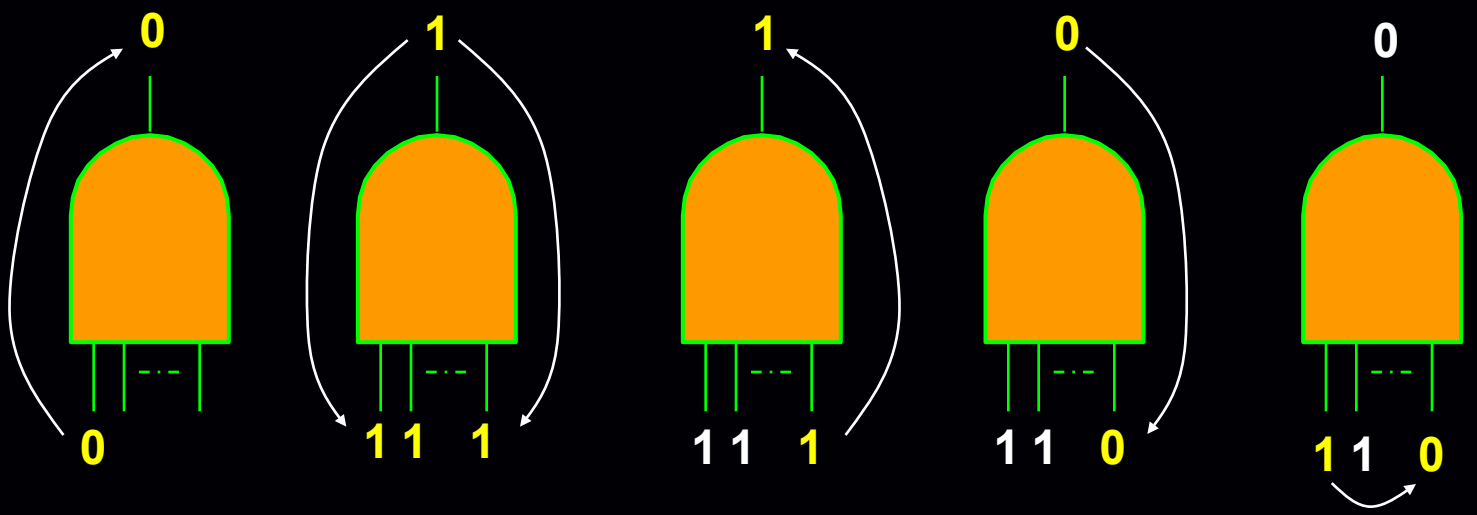
- Circuit-based SAT: gates



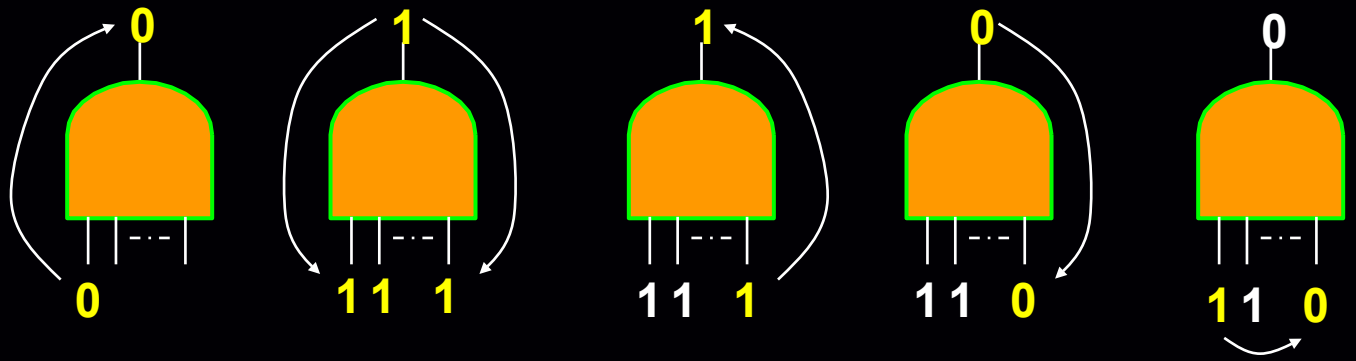
- CNF-based SAT: clauses, variables, literals
  - n+1 clauses
    - (a1 + f ')(a2 + f ')...(an + f ')
    - (a1' + a2' + ... + an' + f)
  - 1 variable (f)
  - 2 literals (f, f ')

Any problem?

- Remember there is only one type of implication in CNF SAT
  - Logic implication can be very efficient
- But implications on an AND gate are trickier

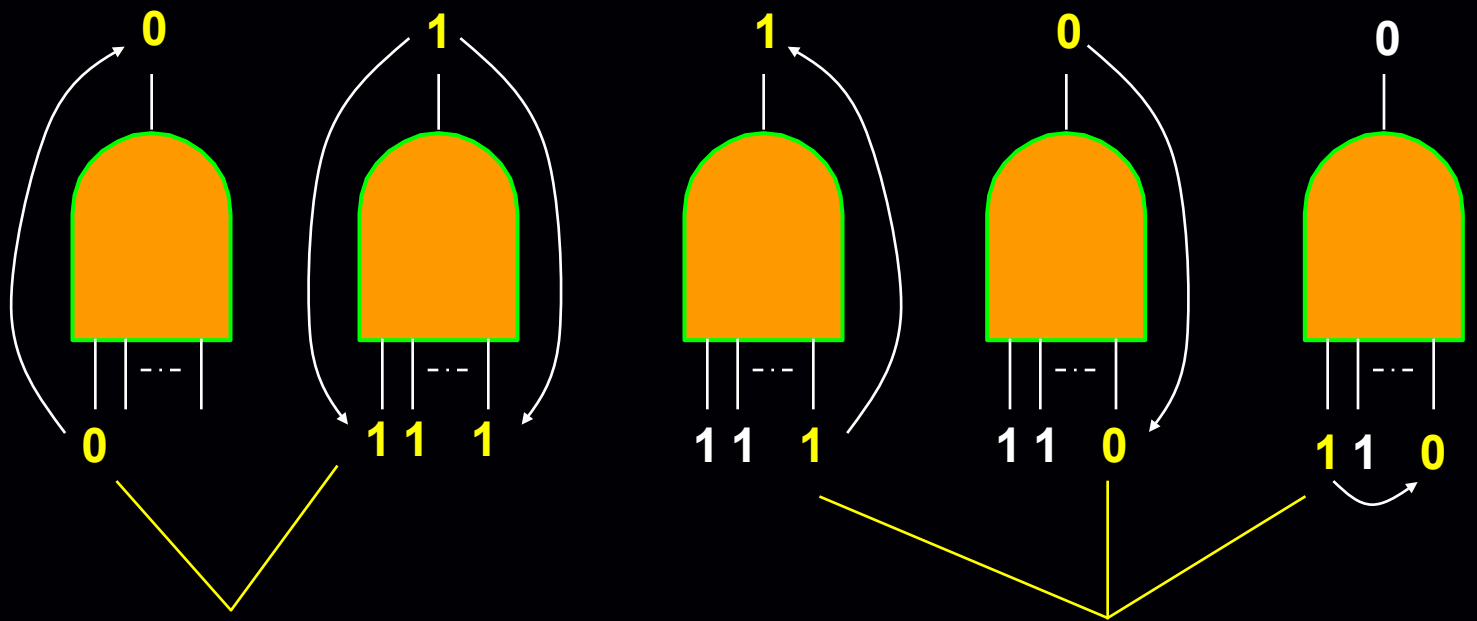


Can circuit SAT do 2-watch?



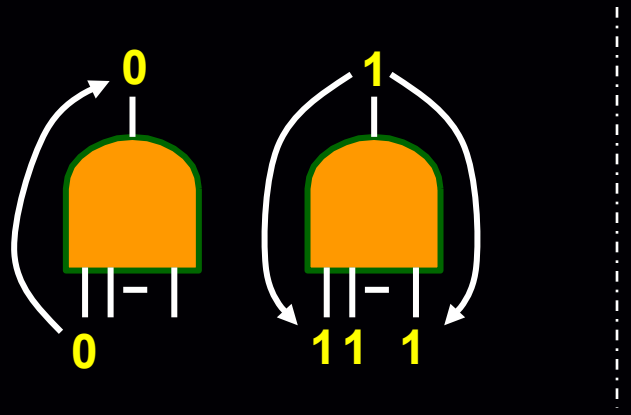
- Watch 2 fanins?
  - What's the watch value?
  - How about gate output?
  - How about OR, NOR, NAND,.. gates?
  - How about XOR, MUX, ... complex gates?

A Closer Look



Different implications on circuit-based SAT actually map to the same clause on CNF SAT

Direct vs. Indirect Implications



1. Direct implication
   - Corresponding to n 2-literal clauses in CNF SAT
   - Single implication source

   ➔ No need to watch

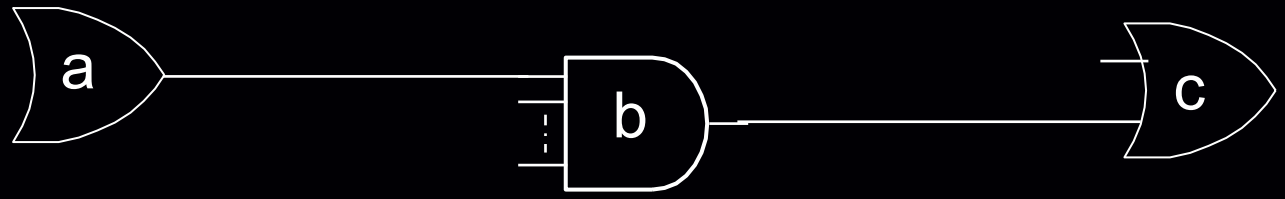
Direct Implication

1. Single source for each implication
2. Only depends on netlist structure; has nothing to do with the proving process (e.g. decisions, etc)
3. Should never encounter “CONFLICT” during the proof process (assume circuit is a DAG)

Direct Implication

➔ Construct a "direct implication graph" in the preprocessing step

➔ Apply direct implications whenever a gate is implied to a value



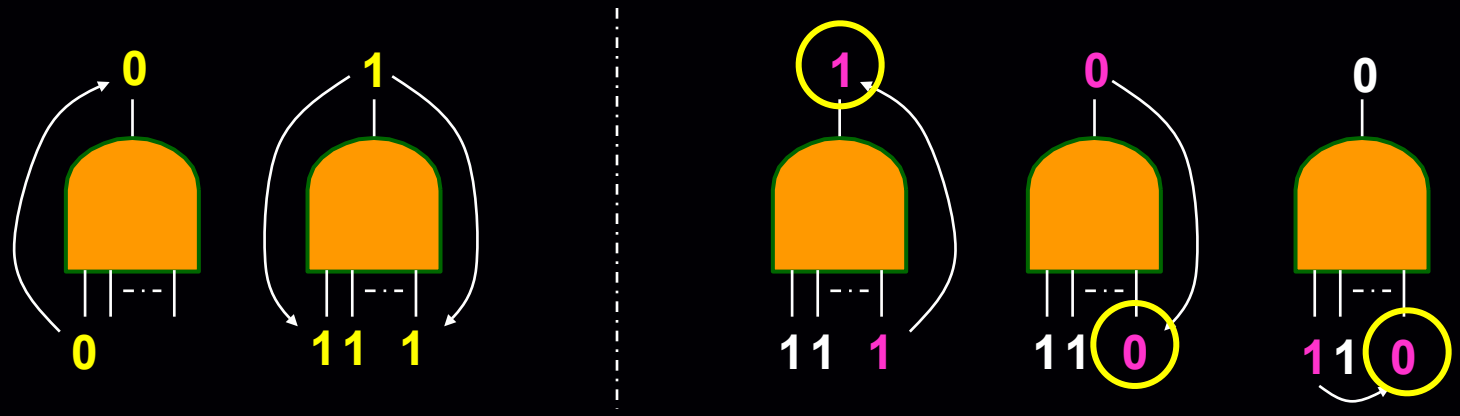
_0DirImps: { $\overline{b}$, ... }
_1DirImps: { ... }

_0DirImps: { ... }
_1DirImps: { a, c, ... }

_0DirImps: { $\overline{b}$, ... }
_1DirImps: { ... }

➔ For example, when b is implied '1', directly iterate through its _1DirImps and imply a = 1 and c = 1

Direct vs. Indirect Implications



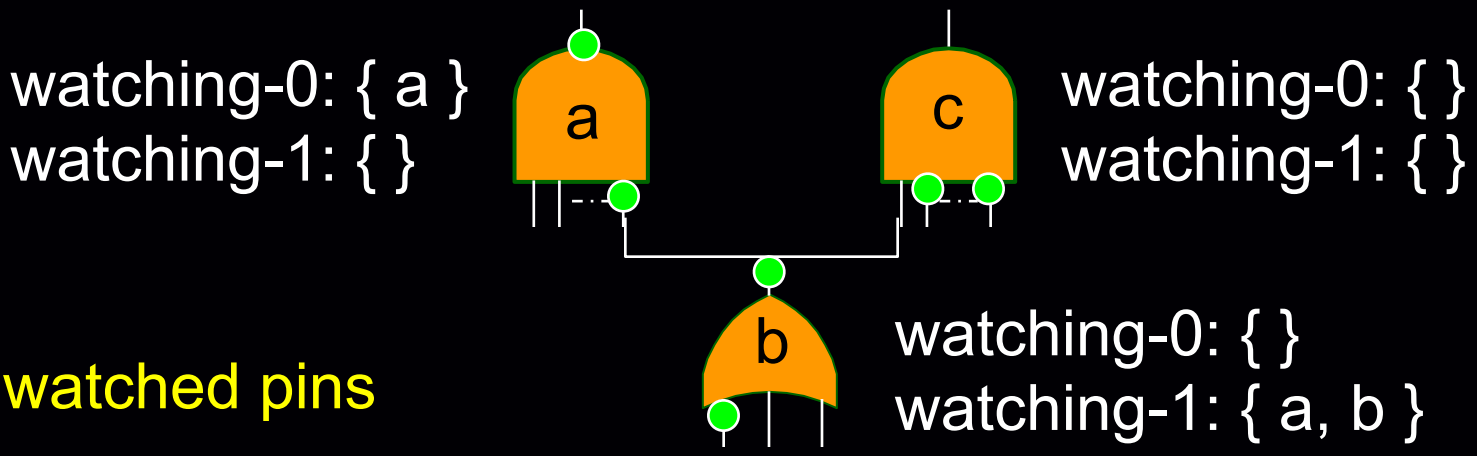
1. Direct implication
   - Corresponding n 2-literal clauses in CNF SAT
   - Single implication source

   ➔ No need to watch
2. Indirect implication
   - Corresponding to the same (n+1)-literal clause
   - Only the last implied pin has different value

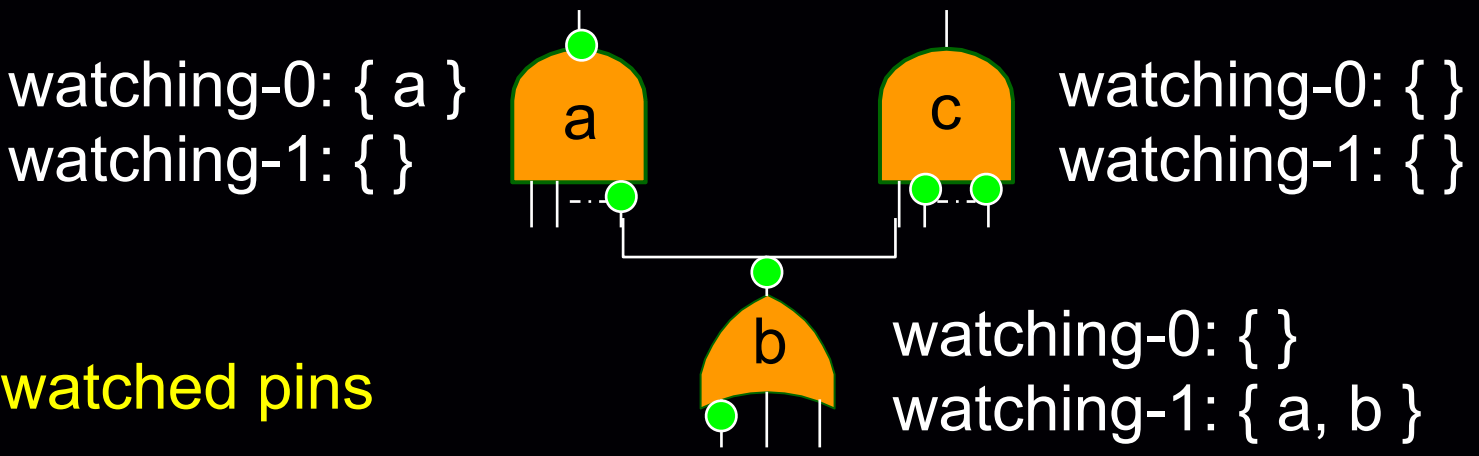
   ➔ 2 watches: among all fanins and the gate itself

Indirect Implication (AND/OR gate)

- Select 2 pins (fanins or the gate itself) in a gate to watch
  - Almost the same as CNF SAT, except that the “watched value” depends on the gate type and I/O
    - For AND (OR) gate, watch 1 (0) for inputs, and 0 (1) for output
    - Watched pins must have non-watched values
  - For each gate, two lists of watching gates
    - When a gate gets a value, perform direct implication and then update watch for the gates on the watching list



Indirect Implication (AND/OR gate)



- watched pins

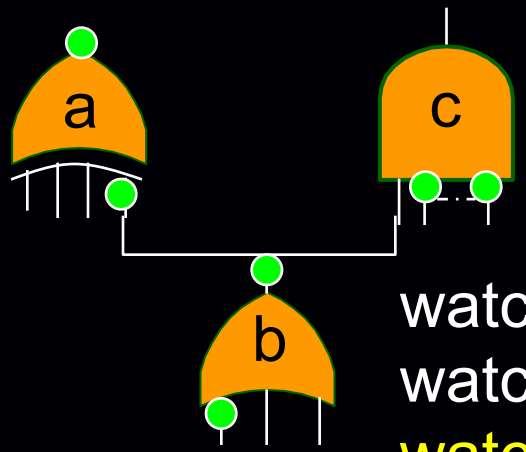
- 'a' watches on itself (output, watch-0) and one of its fanins (watch-1)
  - 'a' is in the watching-0 list of itself, and is in the watching-1 list of 'b'
- If 'b' is implied '1' --
  - Pick 'a' and 'b' from its watching-1 list to update the watched pins
    - 'a' has to update the watched pin for its fanning (to another non-1 fanin)
    - 'b' has to update the watched pin for its fanning (to another non-0 fanin)
    - No action needed for 'c' (because the corresponding fanin is not watched)

Watch Scheme for XOR Gate (How?)

- n-input XOR gate
  - 2^n (n+1)-literal clauses
  - e.g. $(a + b + \overline{f})\ (\overline{a} + b + f)\ (a + \overline{b} + f)\ (\overline{a} + \overline{b} + \overline{f})$
- Implication occurs only when n variables become "known"

  ➔ 2-watch; watch-known

watching-0: { }
watching-1: { }
watching-known: { a }



watching-0: { }
watching-1: { }
watching-known: { }

watching-0: { }
watching-1: { b }
watching-known: { a }

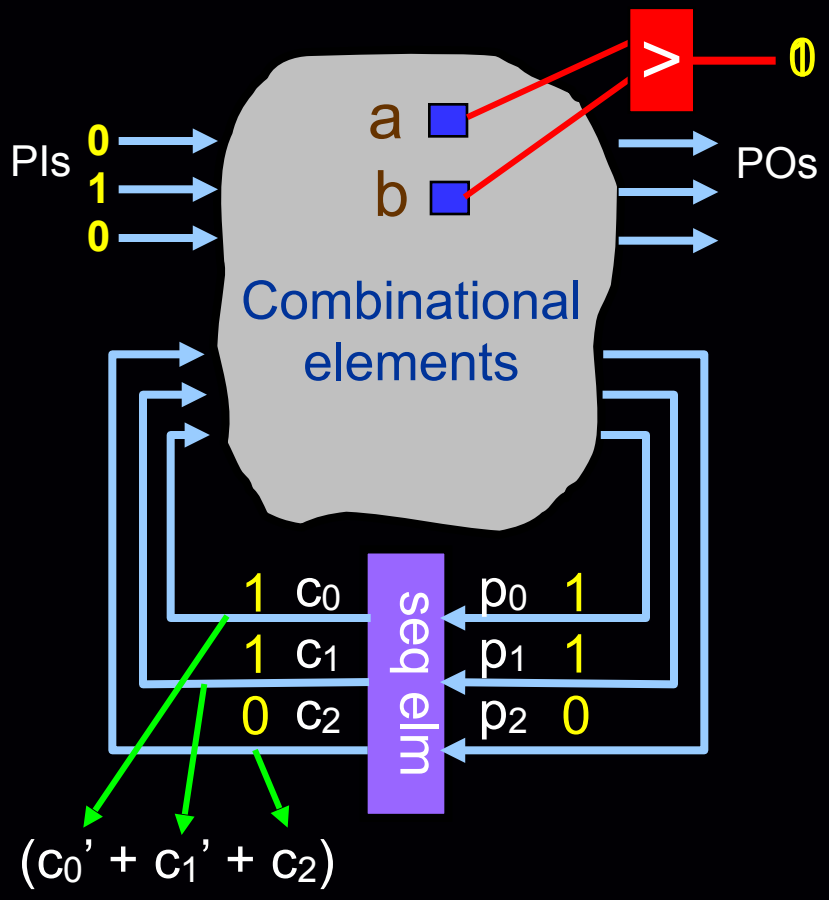
Generic Watch Scheme

- It can be shown that the watch scheme can be extended to complex gates such as MUXes, Pseudo Boolean gates, etc.
- For more details, please refer to:
  - "QuteSAT: A Robust Circuit-based SAT Solver for Complex Circuit Structure", DATE 2007.

Think.... BDD vs. SAT

- For BDD, we compute the set of reachable states by iteratively applying TR on current set of states
- However, SAT is a propositional constraint solver. How to "record" the set of states?
- How to apply SAT in sequential design verification (i.e. Model Checking)?

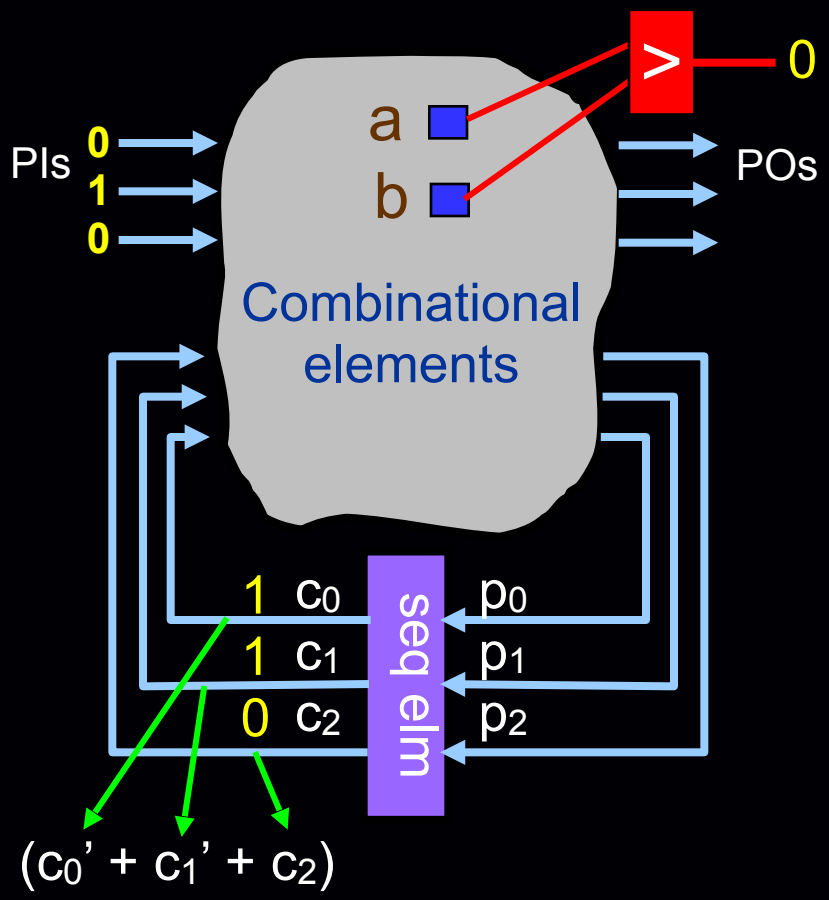
A Naive Approach: Using "Blocking Clauses" for Sequential SAT



Suppose we are solving the property "a > b"

1. Use SAT to get a solution on the registers (for !p)
   e.g. (c_0, c_1, c_2) = (1,1, 0)
2. Add a "blocking clause" (c_0' + c_1' + c_2) to the original CNF
   ➔ Won't get the same state again!!
3. Repeat 2 for another solution in the same timeframe..., or
4. Apply the solution "(c_0, c_1, c_2) = (1, 1, 0)" to the previous state as "(p_2, p_1, p_0) = (1, 1, 0)" and continue to the search in the previous timeframe (for p)

A closer look on the above algorithm...



- There are several calls to SAT
  - Call SAT(p == 0)
  - Put the current state value (e.g. 110) to the previous state variables and call SAT($p_0p_1p_2$ = 110)
- Can the proof efforts/results among different SAT calls be shared?
  - Yes, by "assumpProve()"
  - Also an "incremental SAT" approach

Using Blocking Clauses for Sequential SAT

- The above process needs to continue until ---
  1. The initial state is reached
  2. No new state can be found (i.e. all in blocking clauses)

  ➔ BFS or DFS (in terms of timeframe traversal)?

- However, in the above approach, we are solving one state (cube) at a time.

  Comparing to BDD, which finds all the reachable state in one timeframe at once.

  SAT seems inefficient...

The bottomline is---

- SAT is a satisfiability solver (i.e. to answer "satisfiability"; to find ONE solution)

➔ It is NOT natural for it to enumerate ALL the solutions

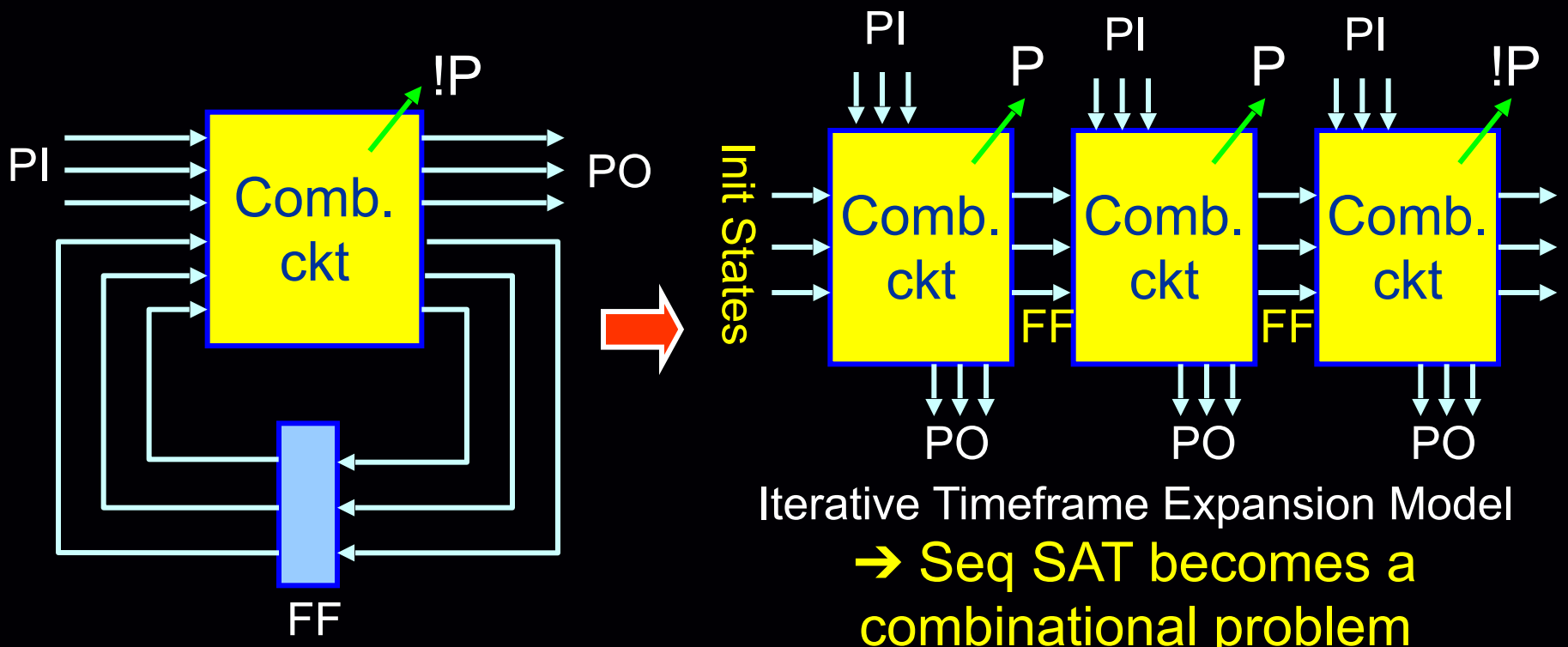
➔ It is NOT a structure for data storage (e.g. hash, BDD)

- SAT solves only propositional constraints

➔ No temporal logic

Iterative Timeframe Expansion Model

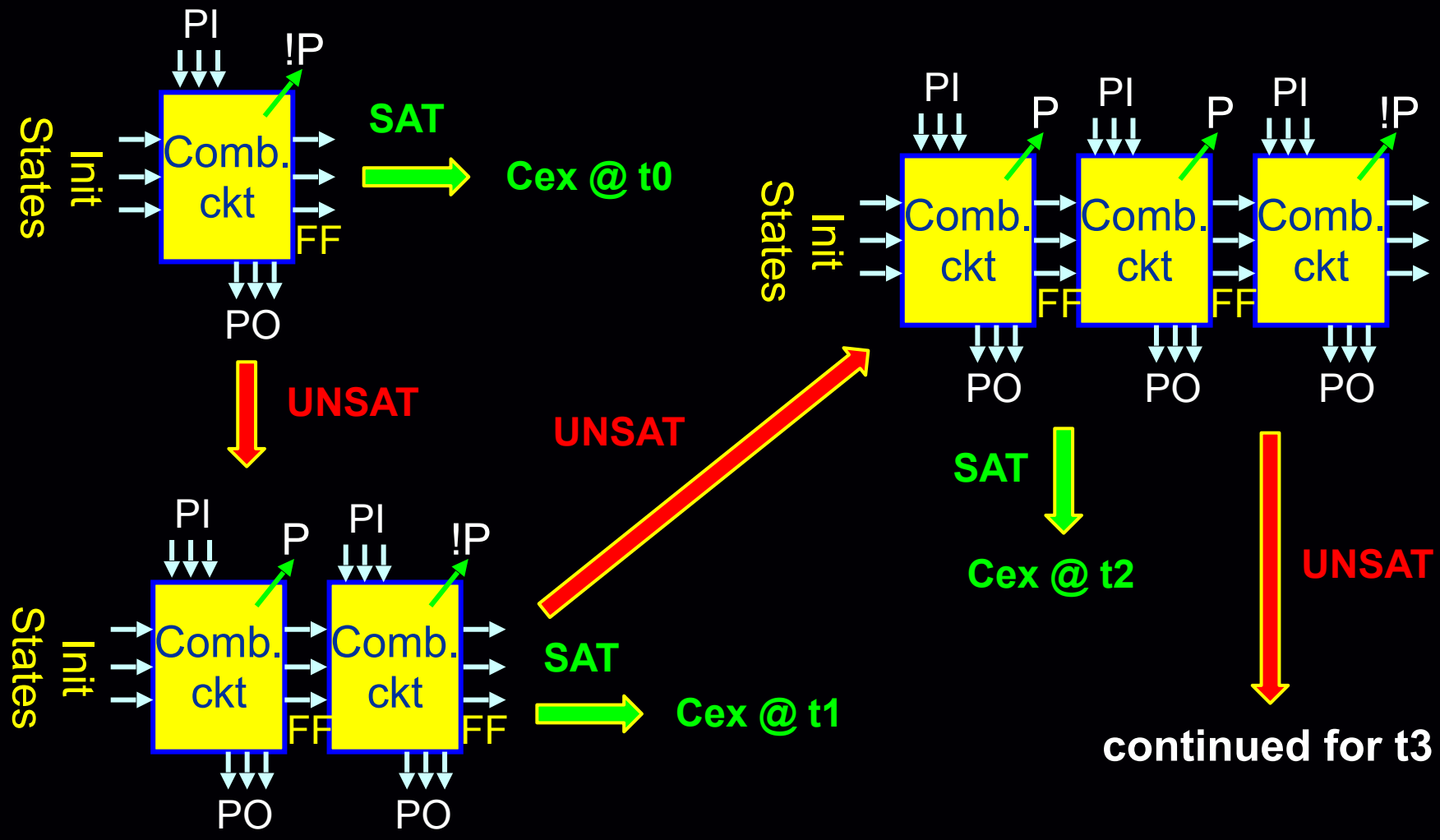
- Here's one way of using SAT for sequential property checking



Iterative Timeframe Expansion Model

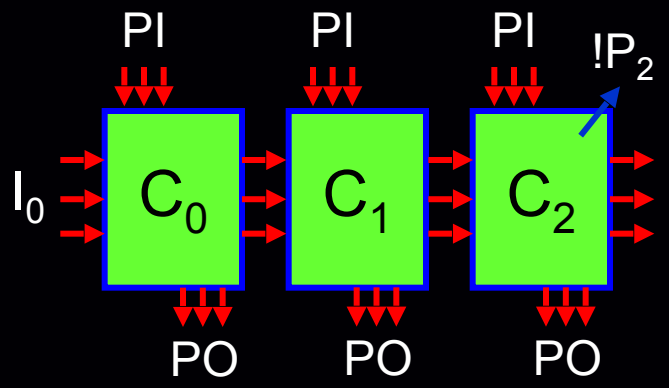
➔ Seq SAT becomes a combinational problem

In other words...



Bounded Model Checking (BMC) Algorithm

- Let 'C' be the set of constraints on the combinational circuit
  ➔ For an iterative model that unfolds the circuit for n times, let 'C_i' correspond to the i-th iteration of the circuit constraint (0 <= i <= n - 1)
- Let 'I_0' be the initial state value
- Let 'P' be the property to prove



- **BMC(P) {**

```
    let k = 1;
  loop:
    if (SAT(I0∧C0∧...∧Ck-1∧!Pk-1))
       return "Find a counter-example @ (K-1)";
    k = k + 1;
    goto loop;
}
```

How far should we go?

- What's the limit of K?

  (How many iterations do we need before concluding the property is always true?)

  ➔ Impossible to know in the above BMC algorithm

  ➔ A loose upper bound is 2^N (N is the number of registers)

Application of BMC

- BMC is particularly useful when BDD encounters the memory explosion problem
- If the property is false, BMC can find a counter-example with the shortest length
- However, BMC cannot conclude that a property is true...

➔ It can only conclude that the property holds up to certain number of timeframes

➔ NOTE: BMC timeframe is different from the number of cycles in a simulation trace!!!

(BMC is best used in “bug-finding”)

Extension of BMC for Unbounded Proof

- BMC, combined with various techniques, can be extended to unbounded model checking

1. K-step Induction
2. Simple-path constraint
3. Counter-example-based abstraction
4. Proof-based abstraction
5. Image computation by SAT
6. Over-approximated image computation using interpolation

etc...

We will cover these...

K-induction

SSS2000

- Induction:

$$\frac{P(s_0) \qquad \forall i: P(s_i) \Rightarrow P(s_{i+1})}{\forall i: P(s_i)}$$

- k-step induction:

$$\frac{P(s_{0..k-1}) \qquad \forall i: P(s_{i..i+k-1}) \Rightarrow P(s_{i+k})}{\forall i: P(s_i)}$$

* Some of the following slides in this lecture note are adopted and modified from Dr. Ken McMillan's CAV03 tutorial

K-induction with a SAT solver

- Let:

  $U_k = C_0 \wedge C_1 \wedge ... \wedge C_k$

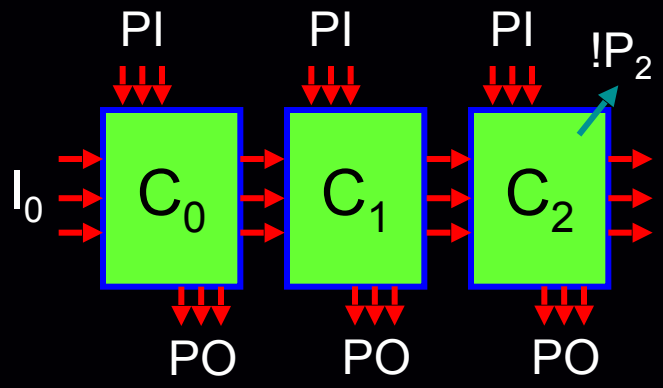
- Two formulas to check:
  - Base case:

    $I_0 \wedge U_{k-1} \Rightarrow P_0...P_{k-1}$

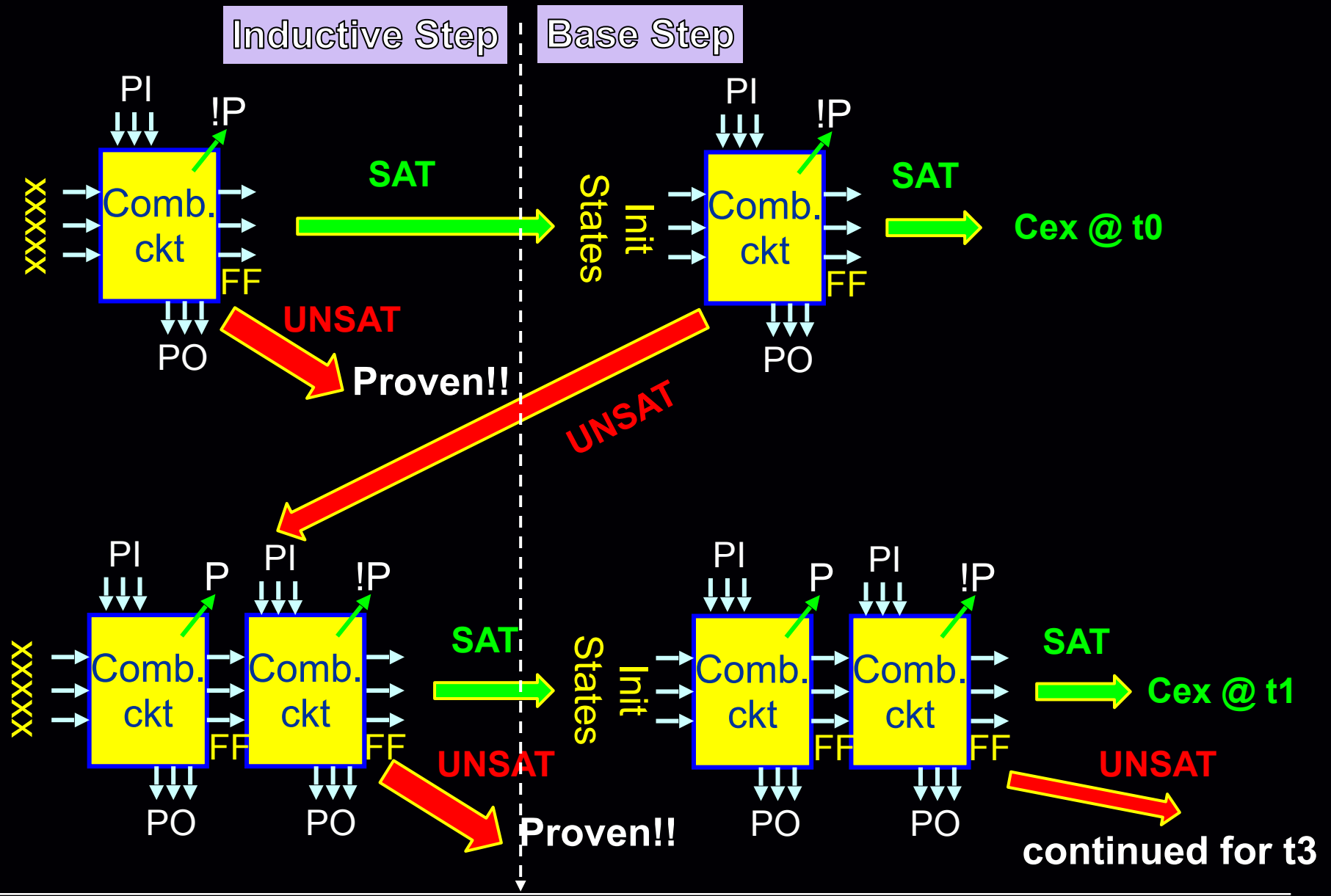
  - Induction step:

    $U_k \wedge P_0...P_{k-1} \Rightarrow P_k$

- If both are valid, then P always holds.
- If not, increase k and try again.



In other words...



Inductive SAT

```
for (k = 0 to infinity)
    S = U_k ∧ F_k  // F_k = P_0 ∧... ∧ P_k-1 ∧ !P_k
    T = I_0 ∧ S
    // induciton step
    if (SAT(S) == false)
        return NO_SOLUTION;  // i.e. P is true
    // normal proof: base case for next k
    if (SAT(T) == true)
        return HAS_SOLUTION; // i.e. CEX is found
    if (effort exceeds limit)
        return ABORT;
endfor
```

Inductive SAT

- In other words, let

  `S(k)` = $U_k \wedge F_k$ // induction step

  `T(k)` = $I_0 \wedge S$ // BMC step

- Induction SAT...

```
if (S(0) == UNSAT)   return UNSAT;
if (T(0) == SAT)     return SAT;
if (S(1) == UNSAT)   return UNSAT;
if (T(1) == SAT)     return SAT;
...
```

Does “Induction SAT” guarantee convergence?

i.e. Will we either (given enough time/memory)

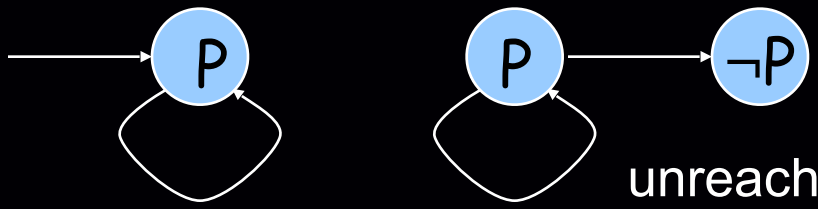
1. conclude no solution in induction step

or 2. find a counter-example in normal proof

with a finite number k ???

Simple path assumption

- Unfortunately, k-induction is not complete.
  - Some properties are not k-inductive for any k.



reachable states from I

unreachable from I, but can reach ¬P for any length of k

- Simple path restriction:
  - There is a path to ¬P iff there is a *simple* path to ¬ P (path with no repeated states).

Induction over simple paths

- Let simple($s_{0..k}$) be defined as:
  - $\forall i,j \text{ in } 0..k : (i \neq j) \Longrightarrow s_i \neq s_j$
- k-induction over simple paths:

$$\frac{P(s_{0..k-1}) \qquad \forall i: \text{simple}(s_{i..i+k}) \wedge P(s_{i..i+k-1}) \Longrightarrow P(s_{i+k})}{\forall i: P(s_i)}$$

Must hold for k large enough, since a simple path cannot be unboundedly long. Length of longest simple path is called recurrence diameter.

...with a SAT solver

- For simple path restriction, let:

  $S_k = \forall t=0..k, t'=t+1..k: \ \neg (\forall v \text{ in } V : v_t = v_{t'})$

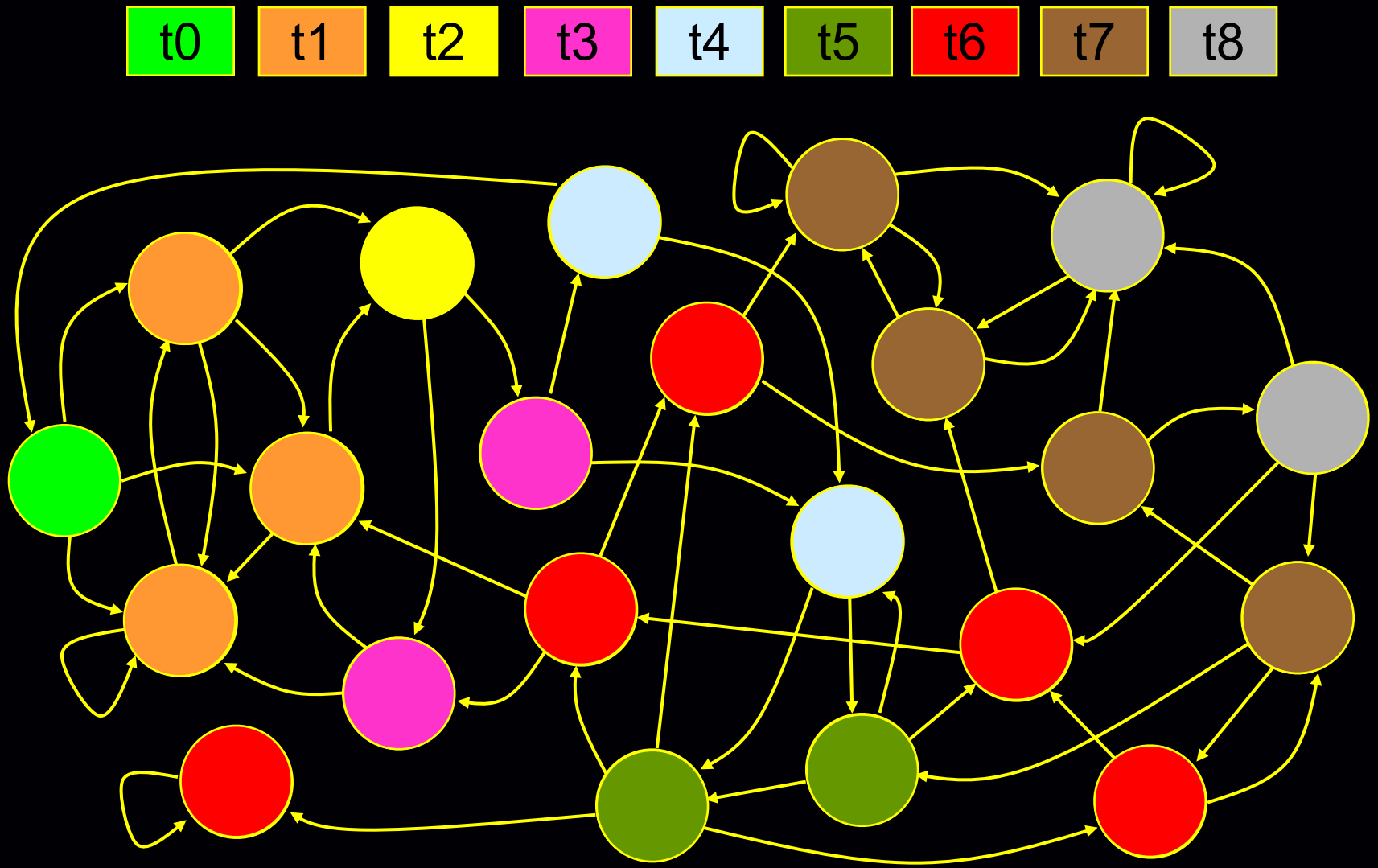
  (where V is the set of state variables).
- Two formulas to check:
  - Base case:

    $I_0 \wedge U_{k-1} \Longrightarrow P_0 ... P_{k-1}$
  - Induction step:

    $S_k \wedge U_k \wedge P_0 ... P_{k-1} \Longrightarrow P_k$
- If both are valid, then P always holds.
- If not, increase k and try again.

Is the recurrence diameter
the same as the diameter
(i.e. the distance from initial state
to any state,
i.e. depth of fixed point)??

Recurrence Diameter vs. Diameter



Termination

- Termination condition:

  k is the length of the longest simple path of the form

  $$P^* \neg P$$

- This can be exponentially longer than the diameter.
  - example:
    - loadable mod 2^N counter where P is (count $\neq 2^N-1$)
    - diameter = 1
    - longest simple path = 2^N
- Nice special cases:
  - P is a tautology (k=0)
  - P is inductive invariant (k=1)

Limitations of simple path constraint

- Although simple path constraint can make the induction-based SAT a complete algorithm for sequential proof, it has the limitation in reality that the circuitry for the simple path constraint can grow too big ($O(n^2)$)
  - Not really applicable in real cases
- What if we limit the simple path constraint to “no repeated states within k timeframes”, where k is a small enough number?
  - Is the algorithm still complete?

Extension of BMC for Unbounded Proof

- BMC, combined with various techniques, can be extended to unbounded model checking

1. K-step Induction
2. Simple-path constraint
3. Counter-example-based abstraction
4. Proof-based abstraction
5. Image computation by SAT
6. Over-approximated image computation using interpolation

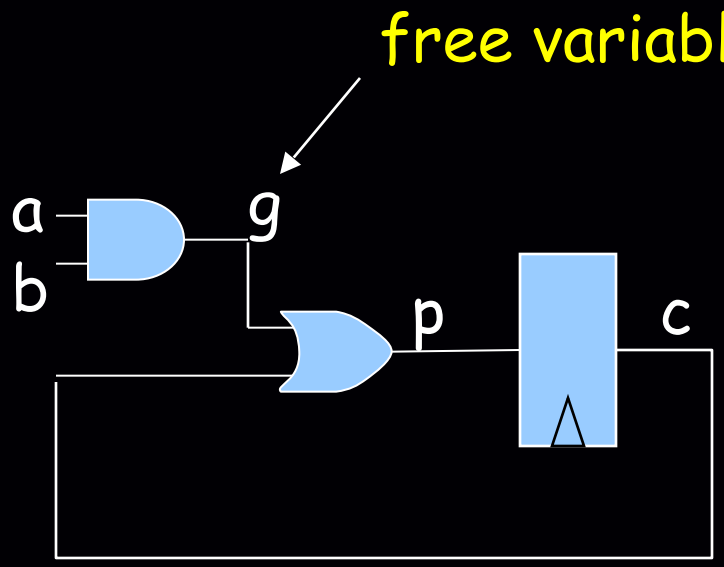
etc...

Note: Limitation of Formal Engine

- Still bounded by exponential complexity
  - When design gets big, or property becomes complex, it is very often that the formal engine cannot conclude the proof result
  - ➔ We have “coverage metric” for simulation, what about formal method?
- However, property checking by formal engine is somewhat analogous to reversely reasoning on designer’s intent
  - Designer’s intent should not be too complicated for one local module
  - It’s the interaction on the modules that make the problem difficult

Localization abstraction

- Property: G (c ⇒ X c)



Model:

C' = {

p = g ∨ c,

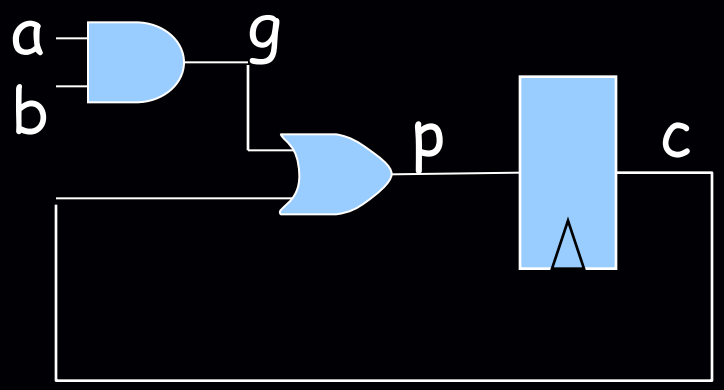
c' = p

}

$$\frac{C' \Rightarrow \text{property},\ C \Rightarrow C'}{C \Rightarrow \text{property}}$$

Constraint granularity

Most authors use constraints at "latch" granularity...



Model:

$C = \{$

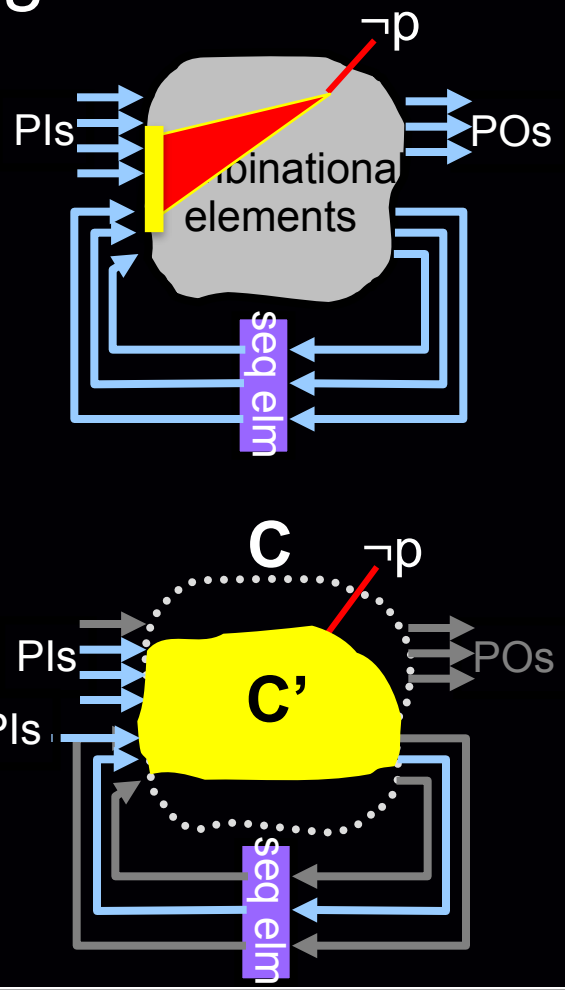
$c' = (a \wedge b) \vee c$

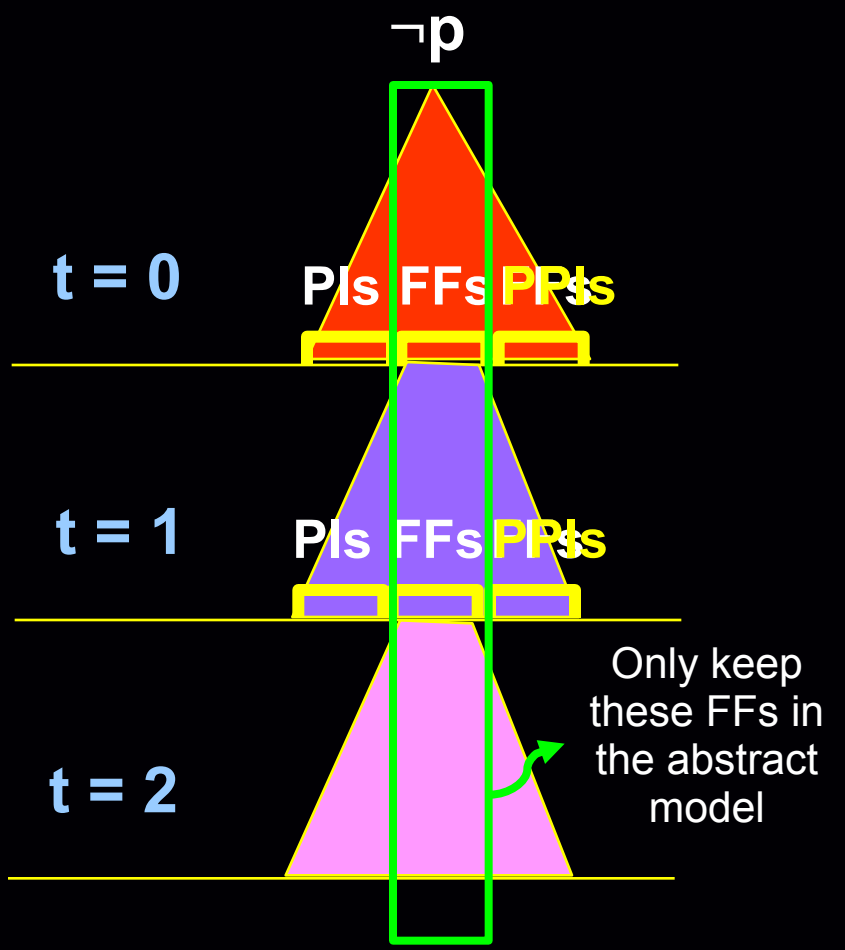
$\}$

...however, techniques we will consider can be applied at both "gate" and "latch" granularity.

Creating Initial Abstract Model C'

- e.g. Cut at FF boundary





Abstraction and Refinement on the Design Under Verification

If we can confine our search/reasoning on some boundary (e.g. local module, FF boundary), we can simplify our proof

➔ Abstraction

But simplifying something means something is ignored.... the result may not be accurate

➔ Refinement

...refined to a bigger search region

Abstraction and Refinement on Reachability Analysis

When computing the set of reachable states, if we can over-approximate the reachable set and show that the bug (!p) is unreachable, we can simplify our proof

➔ Abstraction

But over-approximating the reachability means that some unreachable states may now become reachable.... there may be spurious counter-example and the proof may be inclusive

➔ Refinement

...refined to a tighter over-approximate reachability

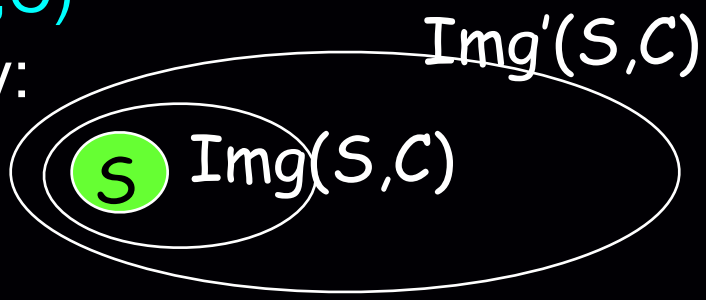
SAT-Based (Approximate) Image

- May provide a good alternative when BDDs fail.
- If we apply SAT to compute the exact image, we do not take advantage of SAT solver's ability to filter out irrelevant facts. Therefore, it may be inefficient.

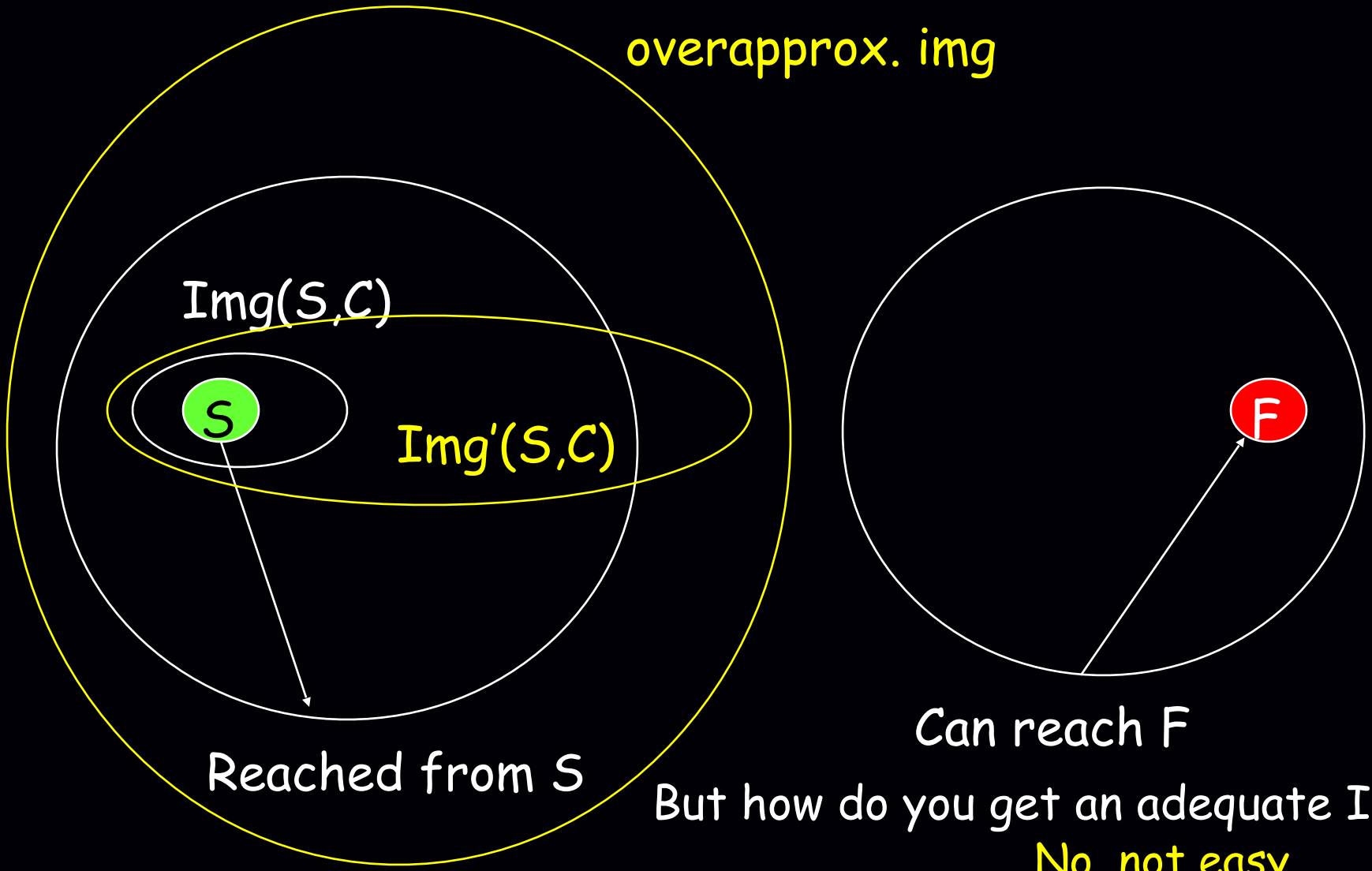
- Can we trade off the image accuracy with the efficiency?

Over-approximated reachability

- Define an over-approximated image op. Img' s.t.
  for all S, Img(S,C) implies Img'(S,C)
- Over-approximated reachability:
  $R'_0 = I$
  $R'_{i+1} = R'_i \vee Img'(R'_i,C)$
  $R' = \cup R'_i$
- Img' is adequate w.r.t. F, when
  - if S cannot reach F, Img'(S,C) cannot reach F
- If Img' is adequate, then
  - F is reachable iff $R' \wedge F \neq$ false



Adequate image



But how do you get an adequate Img'?

No, not easy...

k-adequate image operator

- Img' is k-adequate (w.r.t.) F, when
  - if S cannot reach F,

    Img′(S,C) cannot reach F within k steps
- Note, if k > diameter, then k-adequate is equivalent to adequate.

Image over-approximation

- BMC and Craig interpolation allow us to compute image over-approximation relative to property.
  - Avoid computing exact image.
  - Maintain SAT solver's advantage of filtering out irrelevant facts.

Interpolation

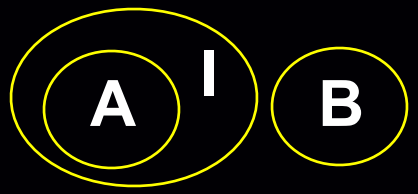
(Craig,57)

- If A ∧ B = false, there exists an *interpolant* I for (A,B) such that:

  A ⇒ I

  I ∧ B = false

  I refers only to common variables of A,B



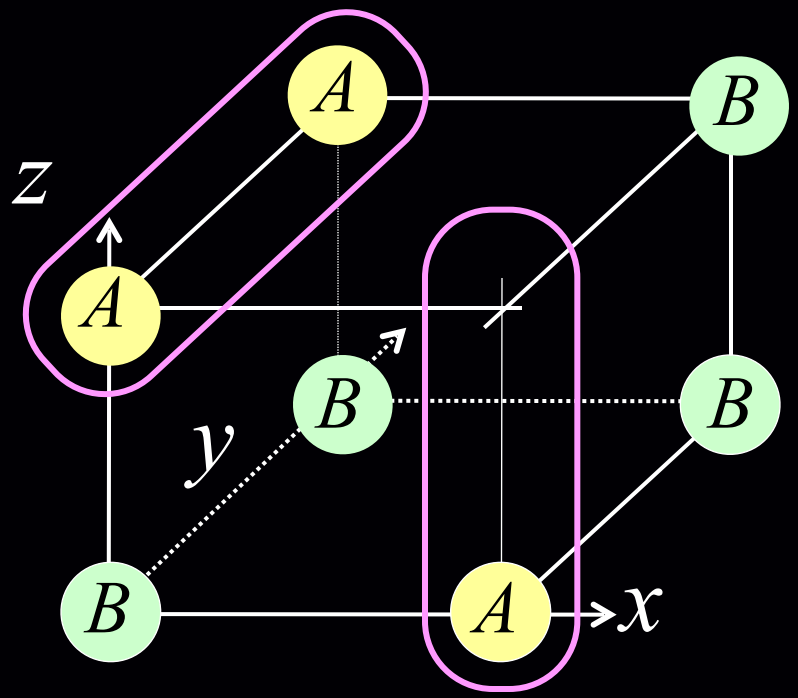
- Example:
  - A = p ∧ q, B = ¬q ∧ r, A' = q
- New result
  - given a resolution refutation of UNSAT A ∧ B, **A' can be derived in linear time.**

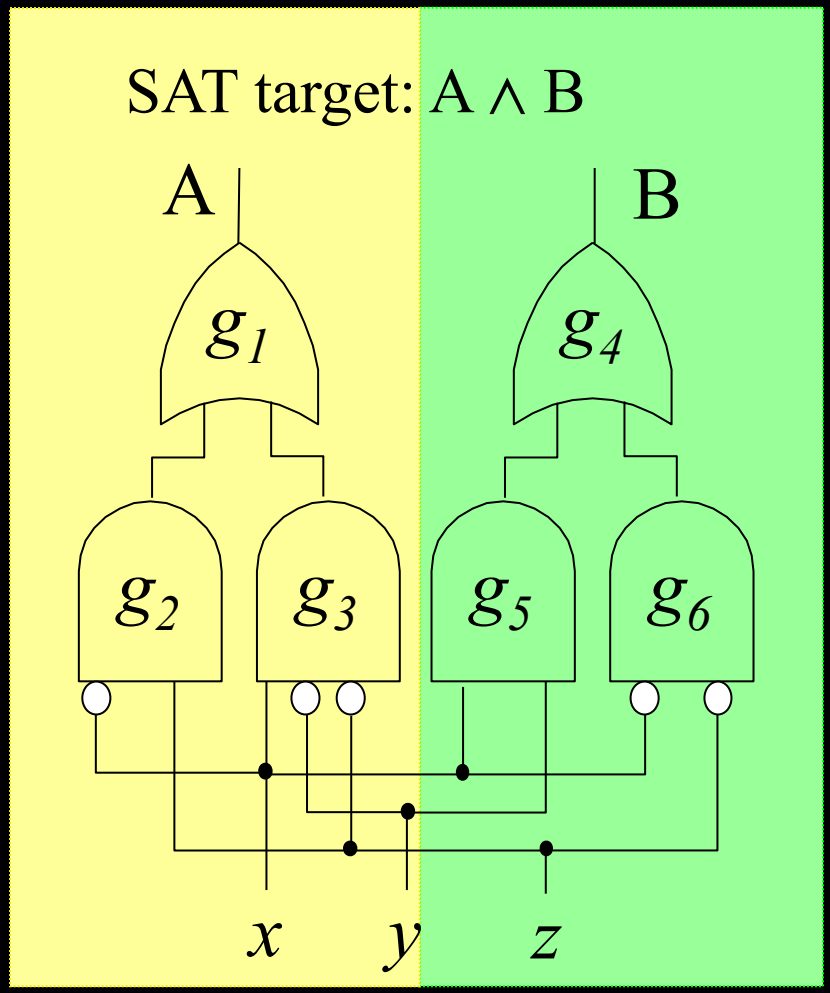
(Pudlak,Krajicek,97)

Another Example of Interpolation

A: $(\neg x \wedge z) \vee (x \wedge \neg y \wedge \neg z)$

B: $(x \wedge y) \vee (\neg x \wedge \neg z)$



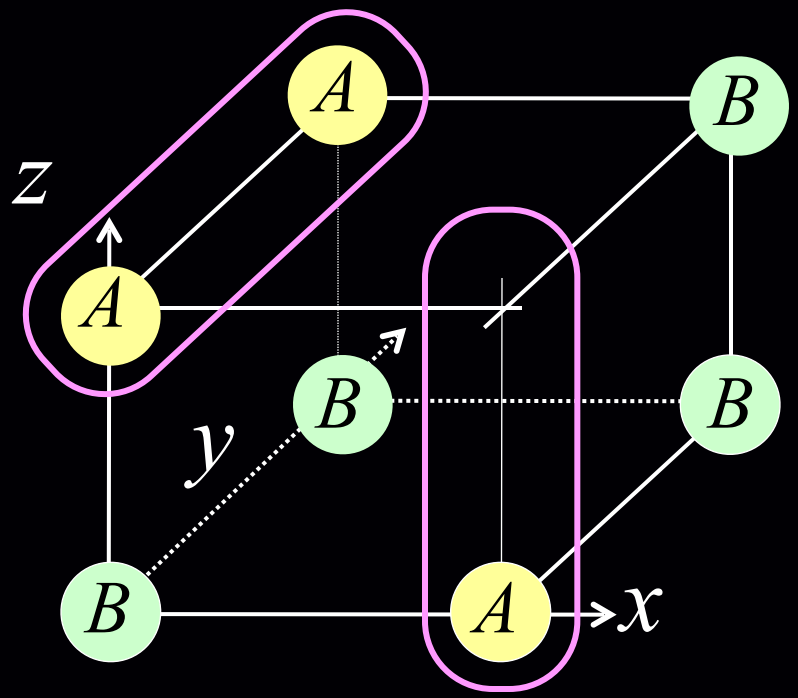


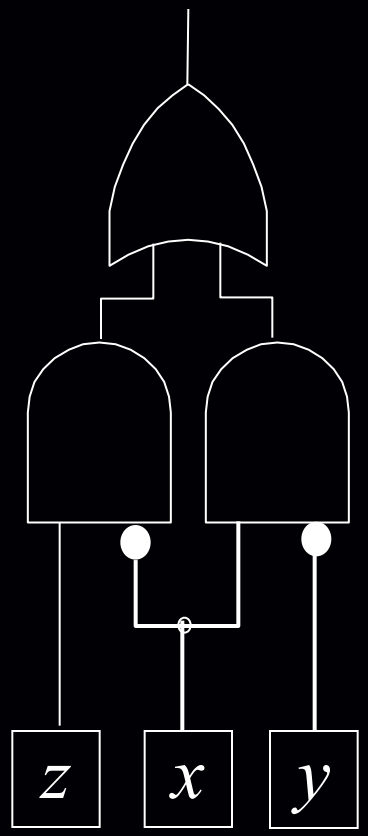
An Example of Interpolation

A: $(\neg x \wedge z) \vee (x \wedge \neg y \wedge \neg z)$

B: $(x \wedge y) \vee (\neg x \wedge \neg z)$

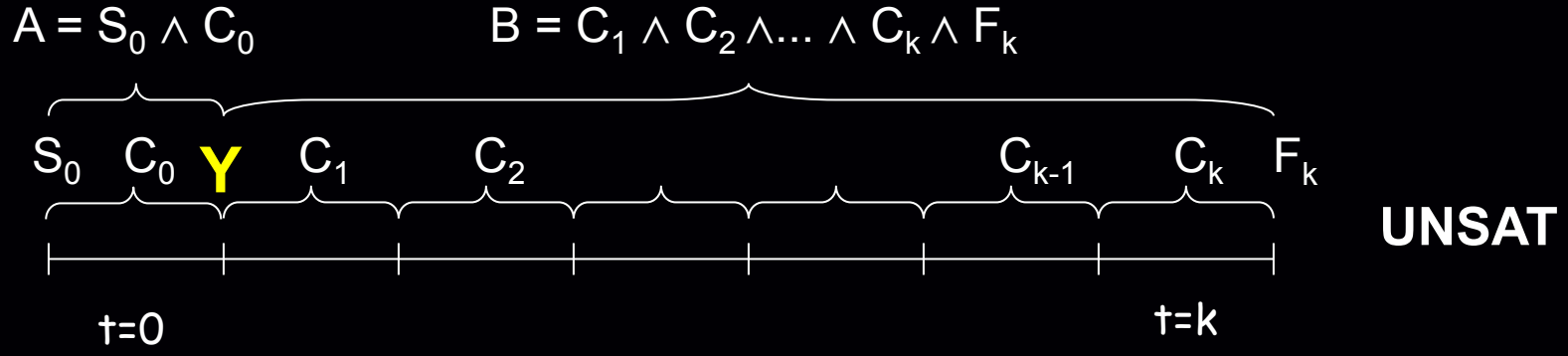
I = $(\neg x \wedge z) \vee (x \wedge \neg y)$





Interpolation-Based Over-approximated Image Computation

- Suppose k^{th}-step BMC returns UNSAT...



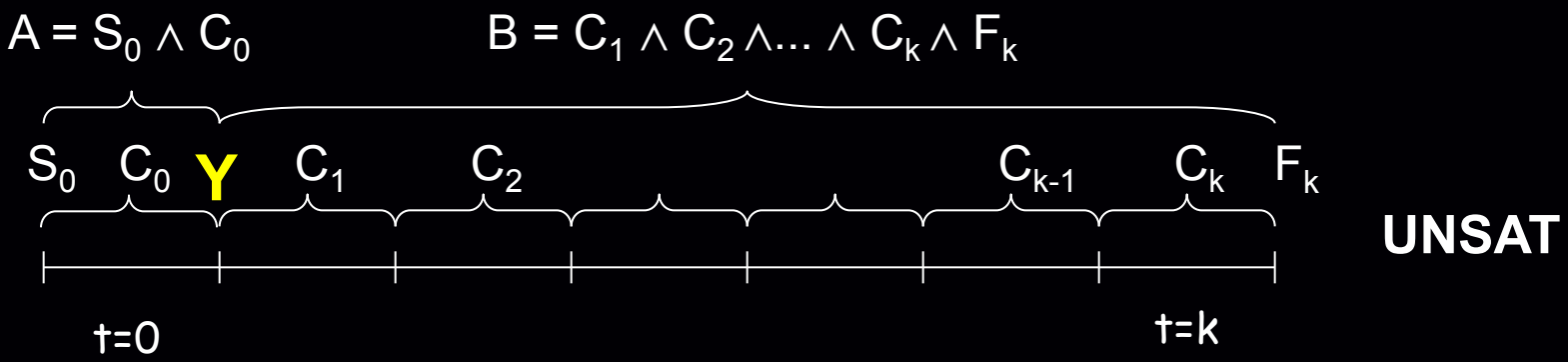
- $A \wedge B$ is UNSAT

=> We can acquire an interpolant I ---

  - $A \Rightarrow I$
  - $I \wedge B$ = false
  - I refers only to common variables (Y) of A,B

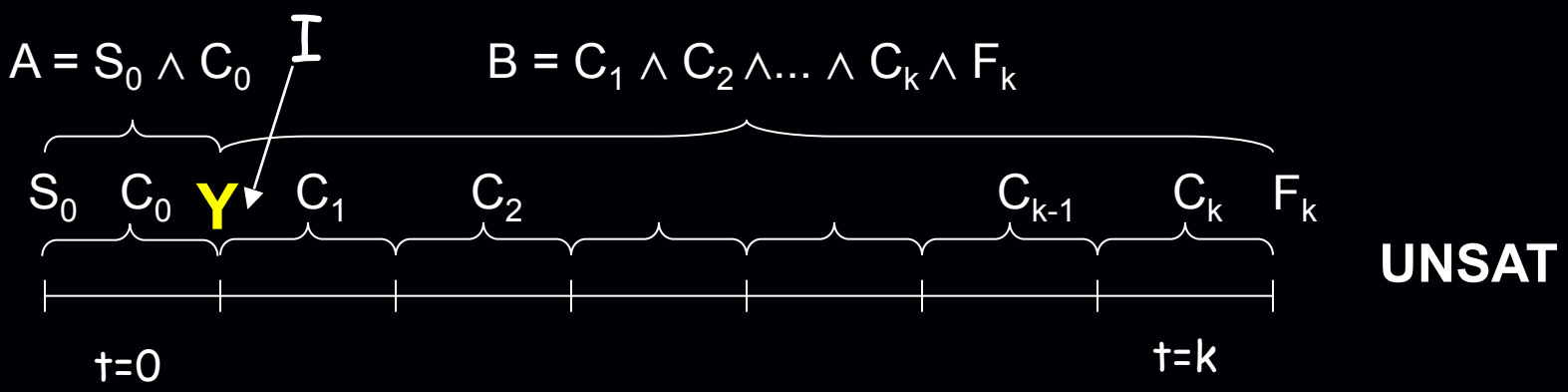
What does "I" mean?

Interpolation-Based Over-approximated Image Computation



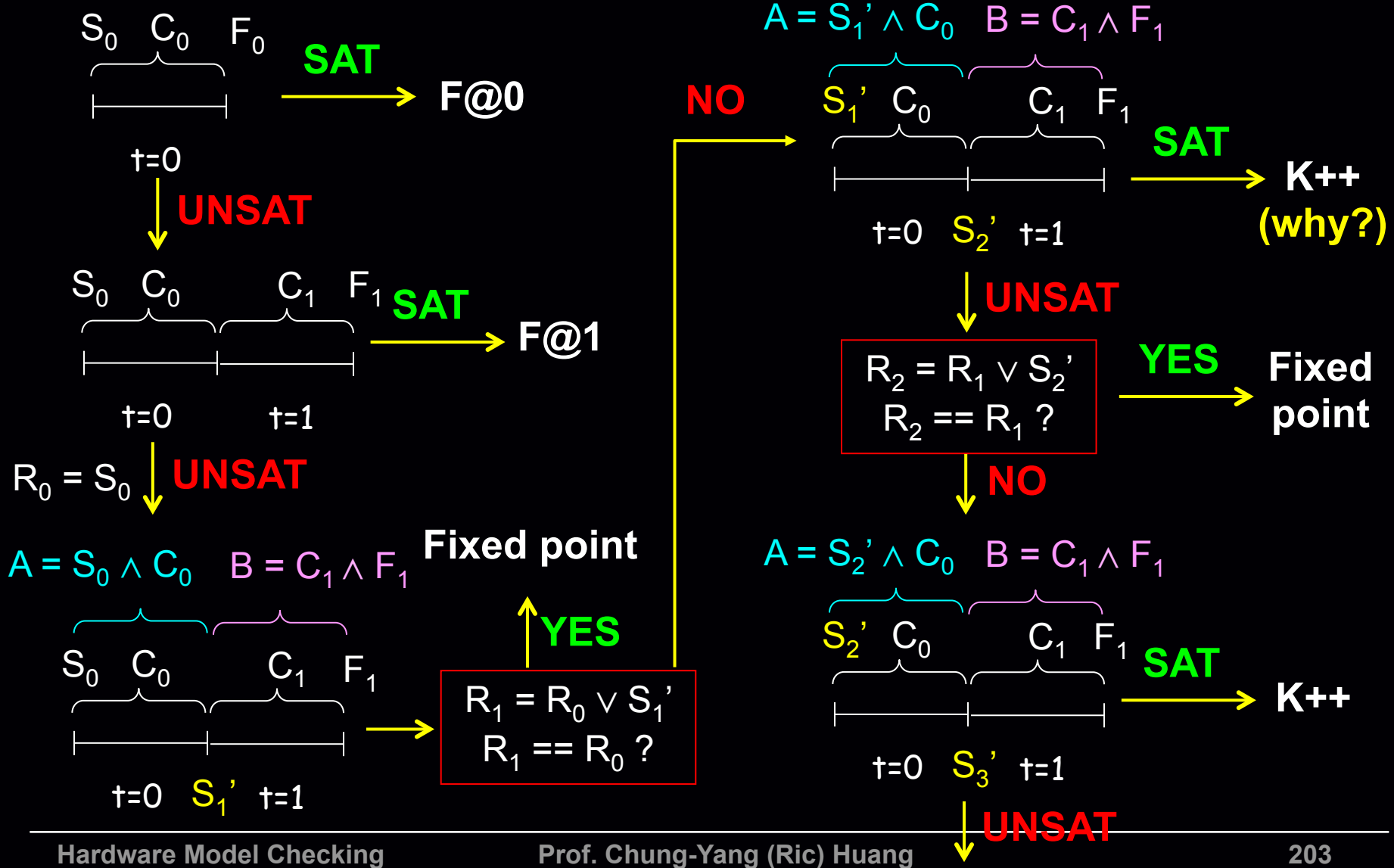
- A = $S_0 \wedge C_0$
  - Characterize S_1 (not computed), the image of S_0, on the variables Y
- B = $C_1 \wedge C_2 \wedge ... \wedge C_k \wedge F_k$
  - Characterize the set of states on variables Y that can witness (reach) F_k in k timeframes
- I = interpolation of A and B
  - A ➔ I
  - $I \wedge B = \varnothing$
  - I refers to A's and B's common variables (Y)
  - ➔ I is an over-approximated image of S_1 on Y
  - ➔ The states in I cannot reach F_k in k timeframes

Explanation



- I tells everything the SAT solver deduced about the image of S_0 in proving it can't reach F in k steps.
- Hence, I is in some sense an abstraction of the image relative to the property.
- Note: I is undefined if A ∧ B is SAT.

Interpolation-based UBMC



Interpolation-based UBMC

let k = 0
repeat_1
  if $BMC_k(S_0, F)$ = SAT, answer reachable
  R = S_0
  let i = 0
  repeat_2
    S_{i+1} = $Img'(S_i,C)$
    if ($BMC_k(S_{i+1}, F)$ = SAT) break repeat_2
    R' = R $\vee$ S_{i+1}
    if R' = R answer unreachable
    R = R'
    increase i
  end repeat_2
  increase k
end repeat_1

Interpolation-Based Unbounded Model Checking provides a pure SAT approach that greatly overcome the scalability problem of the BDD-based approach.

It iteratively computes the over-approximated image of the reachable states, and when a spurious counter-example is found (i.e. BMC returns SAT), it increases the number of timeframes and repeat.

Probably one big drawback is that:

When #timeframes is increased, previous proof efforts (i.e. interpolants) are thrown away.

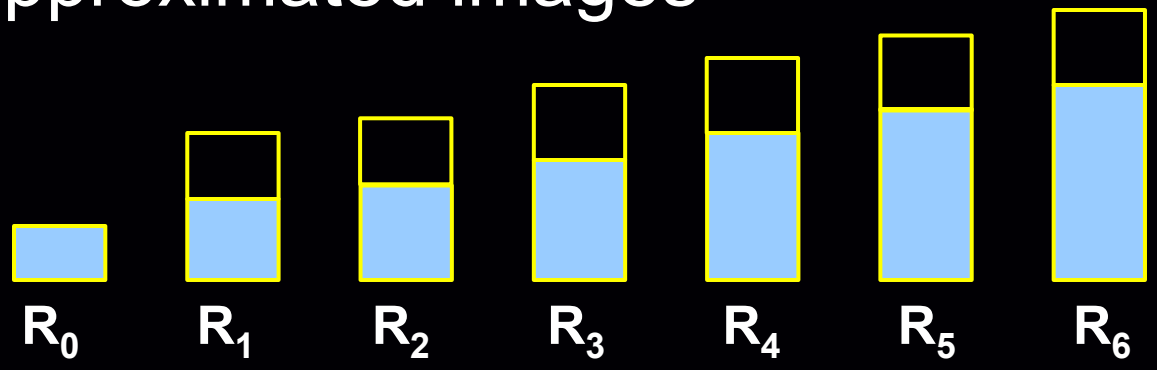
Can we reuse it to improve the efficiency?

PDR: Property Directed Reachability

- Ref:
  Niklas Een, Alan Mishchenko, and Robert Brayton, “Efficient Implementation of Property Directed Reachability”, FMCAD 2011
- Won many categories of HWMCC since 2011

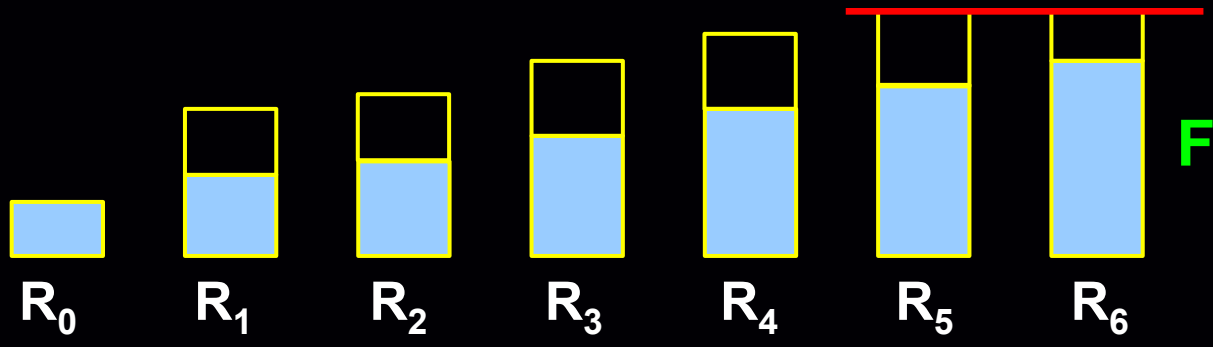
A very high-level view of the PDR algorithm

1. Compute monotonically increased approximated images



Trace of images is recorded

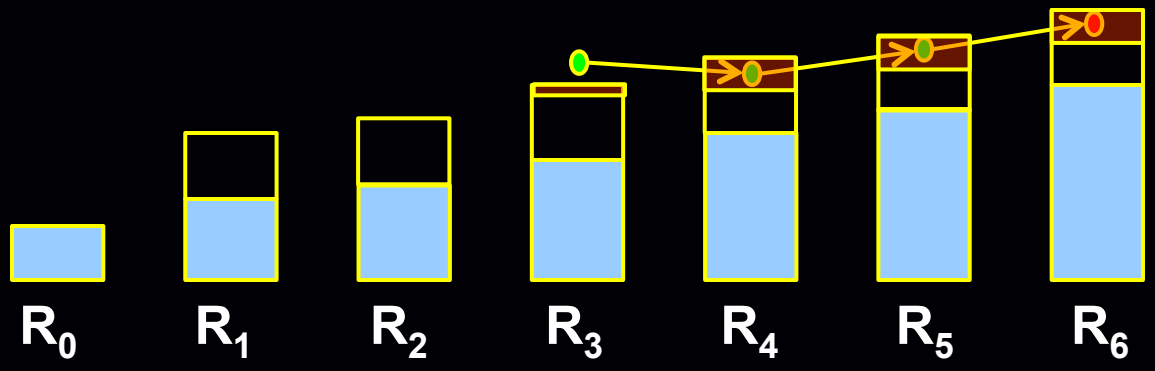
2. If fixed point is found → Property is proven



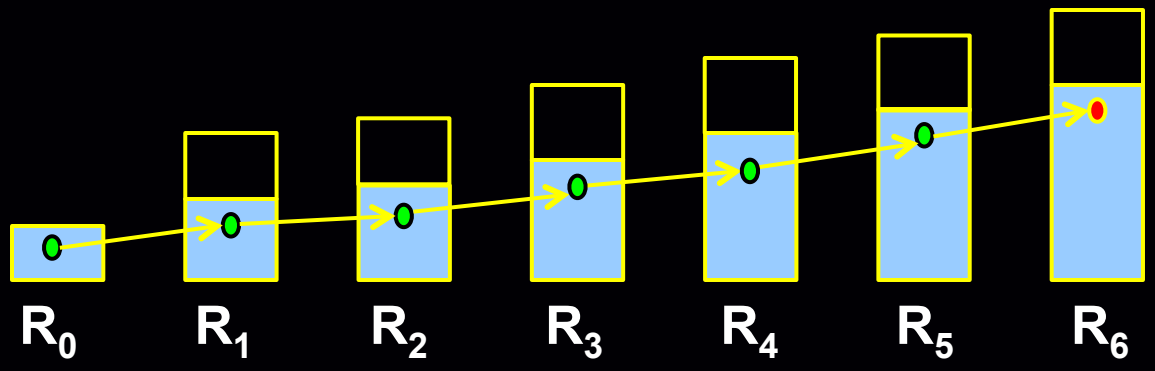
R5 == R6 Fixed point!!

A very high-level view of the PDR algorithm

3. If a spurious cex is found, refine R's



4. If a real cex is found ➔ Property is falsified



Property Directed Reachability Algorithm

- The details of PDR algorithm requires in-depth understanding of "inductive proof" and "abstraction and refinement"
- It is impossible to cover the details in this session
- However, PDR demonstrates its superiority in formal verification of IC designs. Several open-source tools/ frameworks are available:
  - IC3
  - PDR (in Berkeley ABC)
  - rIC3
- For more information and resources, you can refer to "Hardware Model Checking Competition (HWMCC)"

What we have learnt today...

- Challenges in IC design verification
- What is formal verification & Model Checking?
- BDD-based verification techniques
- Why Boolean Satisfiability (SAT) Solvers are so efficient?
- SAT-based verification techniques

Thank you!

Any questions?

Feel free to contact me: ric2k1@{Line, Messenger}