软件分析与架构设计

时序逻辑模型检测

何冬杰
重庆大学

时序逻辑模型检测算法

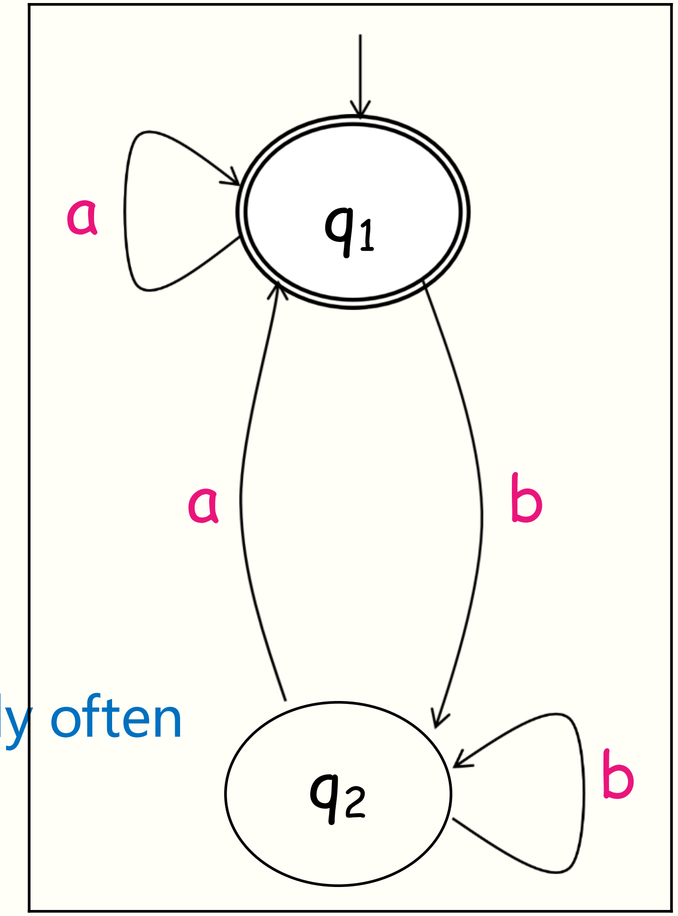
- **Explicit-state model checking:**
  - Specification: temporal logic formula
  - Checking Algorithm:
    - **Exhaustive search of the state space** of the finite-state system, to prove whether the specification is true or not
  - LTL Model Checking (automata-based)
  - CTL Model Checking (fixpoint-based)
- **Symbolic model checking**:
  - manipulate sets of states instead of individual states
  - efficient data structures (BDDs, SAT)
  - Bounded Model Checking
  - More advanced topics are out of scope of this course

LTL Model Checking

- Verify $\boldsymbol{M} \vDash \boldsymbol{\phi}$?
  - M: Kripke Structure
  - LTL formula (AP, ¬,∧,∨, X, G, F, U)
- **Core Idea (Internal Principle of Spin):**
  - $\mathrm{L}(\boldsymbol{M}) = \{\pi \mid \pi \text{ is an abstract execution of } \boldsymbol{M}\}$
  - $\boldsymbol{B}_{\boldsymbol{\phi}}$: automaton that accepts exactly those infinite sentences (π) for which $\boldsymbol{\phi}$ holds, we have $\mathrm{L}(\boldsymbol{B}_{\boldsymbol{\phi}}) = \{\pi \mid \pi \vDash \boldsymbol{\phi}\}$
  - $\mathrm{L}(\boldsymbol{M}) \cap \mathrm{L}(\boldsymbol{B}_{\boldsymbol{\phi}})$: all executions of $\boldsymbol{M}$ that hold $\boldsymbol{\phi}$
  - $\boldsymbol{M} \vDash \boldsymbol{\phi}$ iff $\mathrm{L}(\boldsymbol{M}) \cap \mathrm{L}(\boldsymbol{B}_{\boldsymbol{\phi}}) \neq \emptyset$ iff $\mathrm{L}(\boldsymbol{M}) \cap \mathrm{L}(\boldsymbol{B}_{\neg\boldsymbol{\phi}}) = \emptyset$ iff $\mathrm{L}(\boldsymbol{M} \cap \boldsymbol{B}_{\neg\boldsymbol{\phi}}) = \emptyset$
  - Construct two automata: One for M, one for $\neg\boldsymbol{\phi}$
  - Checking the emptiness of the intersection of the two automata

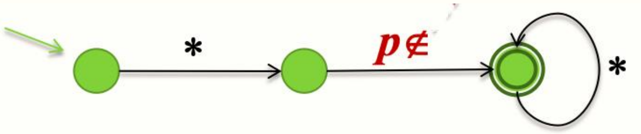
Büchi automata

- **A Büchi automaton is a tuple $(Q, \Sigma, \Delta, q_0, F)$:**
  - Q: a finite set of states
  - Σ: a finite alphabet
  - $\Delta \subseteq Q \times \Sigma \times Q$: a transition relation
  - q_0: an initial state
  - $F \subseteq Q$ defines the acceptance condition:
    - only those runs with some states in F appearing infinitely often
- **Büchi automata vs Normal Automaton**
  - Normal Automaton：输入是有限串 $w = a_0\ a_1 \ldots a_n$
    - 接受条件：读完输入后，停在接受状态 $\delta(q_0, w) \in F$
  - Büchi automata: 输入是无限串 $w = a_0 a_1 a_2 \ldots$
    - 接受条件：存在一个接受状态，被无限次访问 $\mathrm{Inf}(w) \cap \mathrm{F} \neq \emptyset$
  - 响应式系统是无限运行的
    - 操作系统、服务器、控制系统、飞控/嵌入式控制

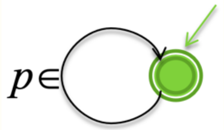


Expressing LTL formula as Büchi

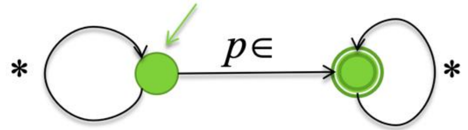
- **Any LTL formula can be converted to a Büchi automaton**
  - Spin implements an automated conversion algorithm
  - Unfortunately, quite complex; see Clarke's Model Checking book
- **A few common transformations**



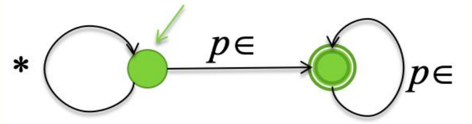
¬X p



G p

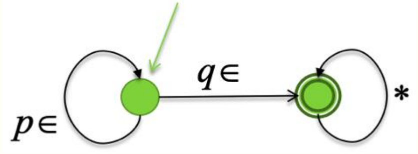


E p

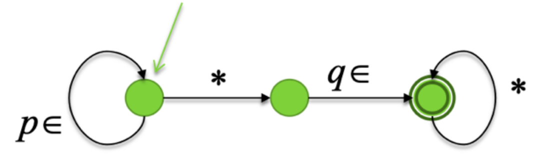


EG p

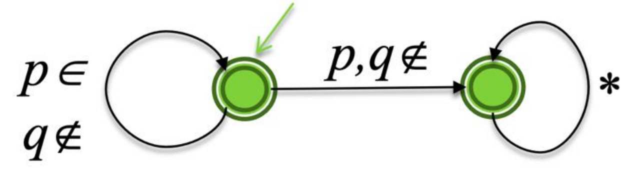
p∉ *Stands for all subsets of Prop that do not contain p; thus implying "p does not hold".*



p U q



p U Xq



¬(p **U** q) = $p \land \neg q$ **W** $\neg p \land \neg q$

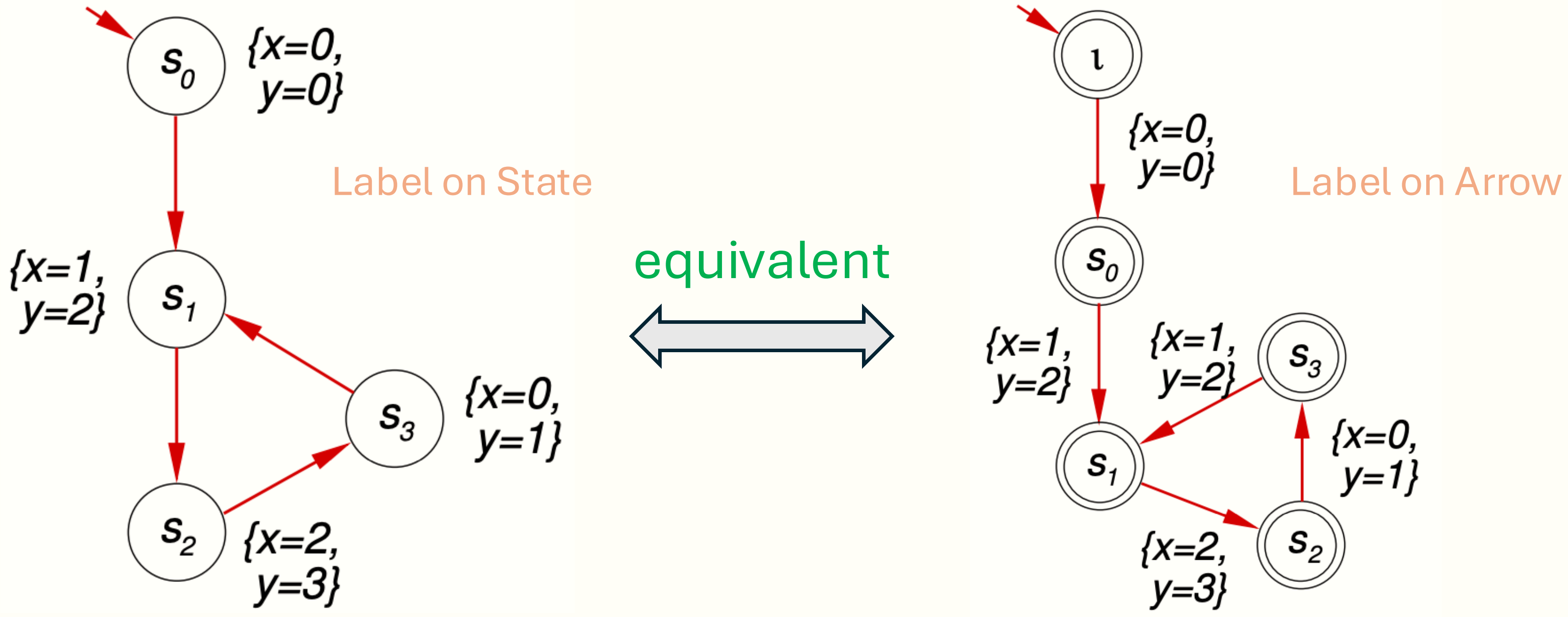
p∈ *Stands for all subsets of Prop that contain p; thus implying "p holds".*

Hard cases: (X p) U q, (p U q) U r

Kripke stuctures to Büchi automata

Any Kripke structure M has an equivalent Büchi automaton $\mathcal{A}$

- M and $\mathcal{A}$ are equivalent iff the sequence of state labels on any path π in M has a correspondent word in $L(\mathcal{A})$.



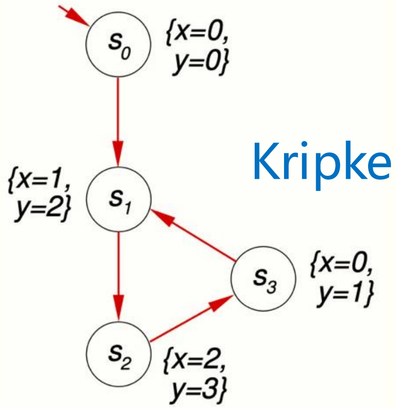
Kripke structure

Büchi automaton

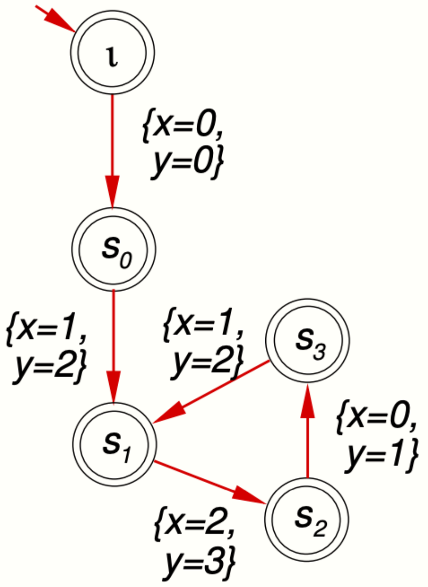
Kripke stuctures to Büchi automata

- ❑ M: **Kripke Structure** $(S, S_0, R, L: S \to \mathcal{P}(AP))$
- ❑ **Write Büchi automata** $\boldsymbol{\mathcal{A}} = (\boldsymbol{\Sigma}, \boldsymbol{Q}, \Delta, \boldsymbol{Q_0}, \boldsymbol{F})$**:**
  - ➢ **alphabet** $\boldsymbol{\Sigma} := \mathcal{P}(AP)$: the set of subsets of AP
  - ➢ Add an **initial state** ι
    - ○ $Q_0 := \{\iota\}$, $Q := S \cup \{\iota\}$, and $F := S \cup \{\iota\}$
  - ➢ New **transitions** are:
    - ○ $(\iota, \alpha, s) \in \Delta$ iff $s \in S_0$ and $\alpha = L(s)$
    - ○ $(s, \alpha, s') \in \Delta$ iff $s, s' \in S$, $(s, s') \in R$ and $\alpha = L(s')$

Label on State



Kripke structure

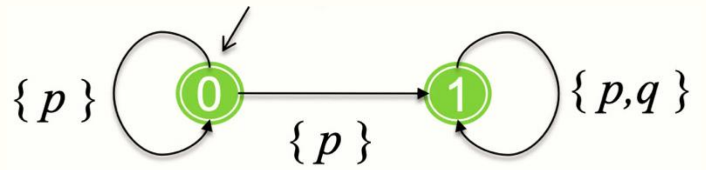


Büchi automaton

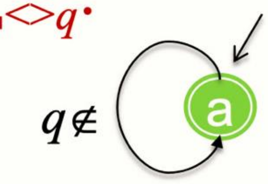
Constructing Intersection of Automata

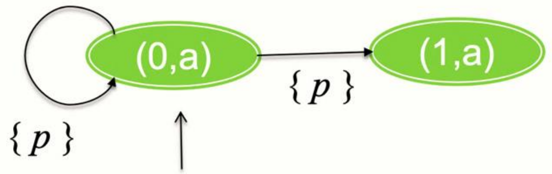
- ❑ **Two Büchi automata over the same alphabet:**
  - ➢ $\mathcal{A}_A = (\boldsymbol{\Sigma}, \boldsymbol{Q}_A, \Delta_A, \boldsymbol{Q}_{A_0}, \boldsymbol{F}_A)$, $\mathcal{A}_B = (\boldsymbol{\Sigma}, \boldsymbol{Q}_B, \Delta_B, \boldsymbol{Q}_{B_0}, \boldsymbol{F}_B)$
  - ➢ $\boldsymbol{F}_A$ is ass , increment $\boldsymbol{Q}_A$
- ❑ $\mathcal{A}_A \cap \mathcal{A}_B = (\boldsymbol{\Sigma}, \boldsymbol{Q}_A \times \boldsymbol{Q}_B, \Delta, (\boldsymbol{Q}_{A_0}, \boldsymbol{Q}_{B_0}), \boldsymbol{F}_A \times \boldsymbol{F}_B)$:
  - ➢ lock-step execution of both:
    - ○ Can make a transition only if $\mathcal{A}_A$ and $\mathcal{A}_B$ can both make the transition on $\boldsymbol{\Sigma}$

B_M



$B_{\neg\diamond q}$:



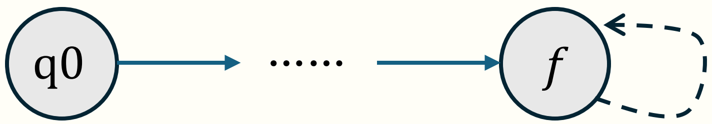


$B_M \cap B_{\neg\diamond q}$:

LTL Model Checking Algorithm

- $\boldsymbol{M} \vDash \boldsymbol{\phi}$ iff $\mathrm{L}(\boldsymbol{M} \cap \boldsymbol{B}_{\neg\boldsymbol{\phi}}) = \emptyset$
- **Let $\boldsymbol{C} = \boldsymbol{M} \cap \boldsymbol{B}_{\neg\boldsymbol{\phi}}$ be the intersection automaton**

> $\mathrm{L}(C) \neq \emptyset$ iff there is a finite path from C's initial state to an accepting state f, followed by a cycle back to f.



- So, it comes down to a cycle finding in a finite graph! Solvable.
- The path leading to such a cycle also acts as a counting example!

- **Approaches:**
  - **A1**: calculate SCCs with Tarjan and check if there is an SCC containing an accepting state, reachable from C's initial state
  - **A2**: based on Depth First Search (DFS), $\boldsymbol{C}$ can be built lazy

LTL Model Checking Algorithm

❑DFS-based Approach (used by Spin)

```
DFS(u)  {
  if (u ∈ visited) return;
  visited.add(u);
  for each (s ∈ next(u))  {
    if (u ∈ accept)  {
      visited2 = ∅;
      checkCycle(u,s);
    }
  }
}
```

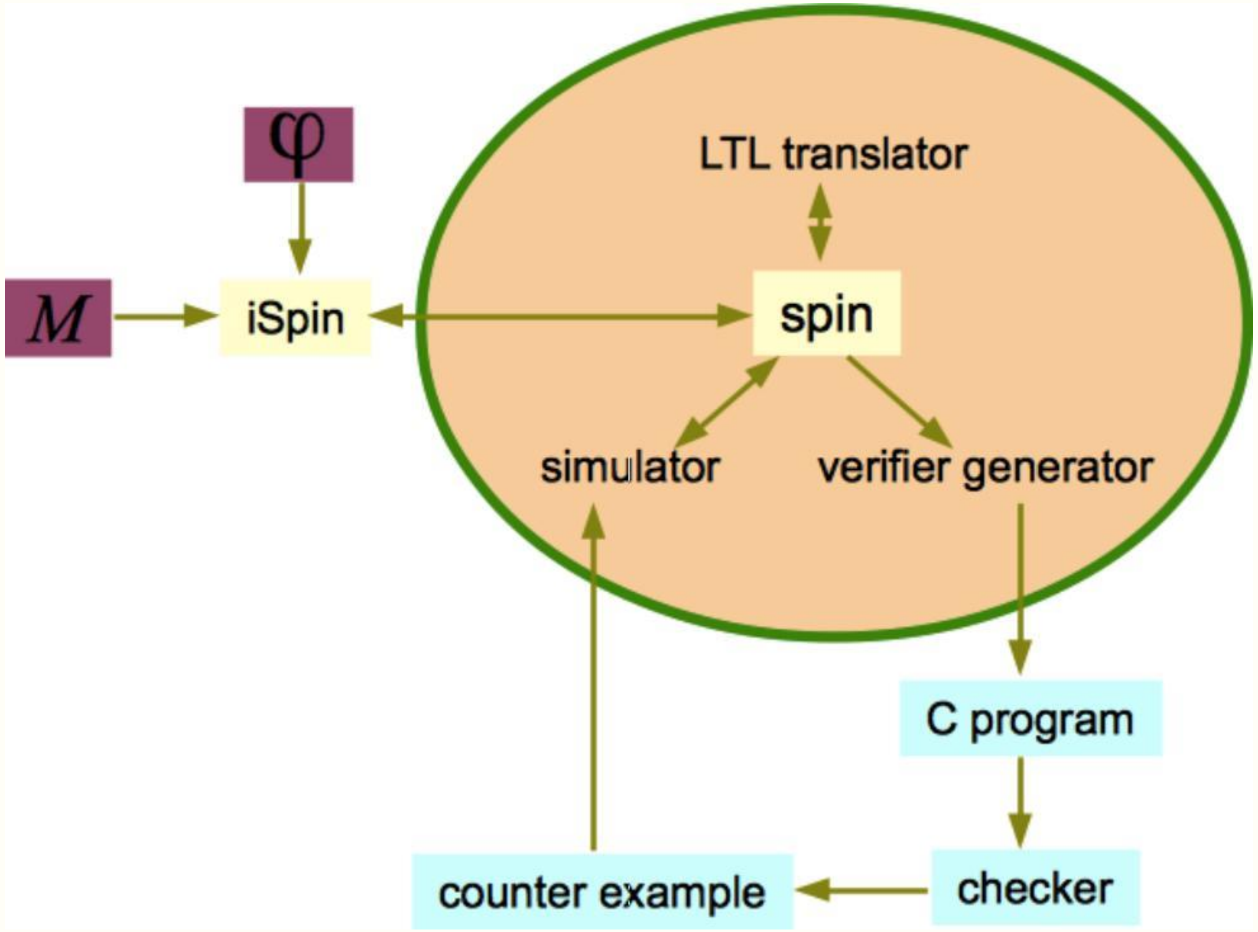
```
checkCycle(root,s)  {
  if(s == root) throw CycleFound;
  if(s ∈ visited2) return;
  visited2.add(s);
  foreach(t ∈ next(s))
    checkCycle(root, t);
}
```

思考：如何修改使能追踪从初始state到cycle的路径（即counter example）？

❑Lazily constructing the intersection automaton C

- Check `s ∈ next(u)` is to check $(s_1, s_2) \in next_C(u_1, u_2)$
  - Check if there is a label L, s.t., $s_1 \in next_M(u_1, \mathrm{L}) \wedge next_{\neg\phi}(u_2, \mathrm{L})$
- Check `u ∈ accept:` $(u_1, u_2) \in accept_C \equiv u_2 \in accept_{\neg\phi}$
- Benefit：if verification fails, effort is not wasted to construct the full C

(i)Spin Architecture



CTL Model Checking

- **CTL has a simpler model-checking algorithm than LTL**
- **Kripke structure** $K = (\mathrm{W}, \boldsymbol{S_0}, \curvearrowright, \mathrm{v})$ **with finite states**
- **CTL model checking algorithm:**
  - Computes the set of all states of $\boldsymbol{K}$ in which CTL formula ϕ is true
  - $[\![\phi]\!] \stackrel{\text{def}}{=} \{s \in W \ : \ s \models \phi\}$
- **Sound axioms for the computation structures of CTL:**

$$\text{(EG)} \quad \mathbf{EG}P \leftrightarrow P \wedge \mathbf{EXEG}P$$

$$\text{(EF)} \quad \mathbf{EF}P \leftrightarrow P \vee \mathbf{EXEF}P$$

$$\text{(EU)} \quad \mathbf{E}P\mathbf{U}Q \leftrightarrow Q \ \vee \ P \wedge \mathbf{EXE}P\mathbf{U}Q$$

$$\text{(AU)} \quad \mathbf{A}P\mathbf{U}Q \leftrightarrow Q \ \vee \ P \wedge \mathbf{AXA}P\mathbf{U}Q$$

CTL Model Checking: computing $[\![\phi]\!]$

1. $[\![p]\!] = \{s \in W \ : \ v(s)(p) = true\}$ *for atomic propositions* p
2. $[\![\neg P]\!] = W \setminus [\![P]\!]$
3. $[\![P \wedge Q]\!] = [\![P]\!] \cap [\![Q]\!]$
4. $[\![P \vee Q]\!] = [\![P]\!] \cup [\![Q]\!]$
5. $[\![\mathbf{EX}P]\!] = \tau_{\mathbf{EX}}([\![P]\!])$ *using the existential successor function* $\tau_{\mathbf{EX}}()$ *defined as follows:*

$$\tau_{\mathbf{EX}}(Z) \stackrel{\text{def}}{=} \{s \in W \ : \ t \in Z \textit{ for some state } t \textit{ with } s \curvearrowright t\}$$

6. $[\![\mathbf{AX}P]\!] = \tau_{\mathbf{AX}}([\![P]\!])$ *using the universal successor function* $\tau_{\mathbf{AX}}()$ *defined as follows:*

$$\tau_{\mathbf{AX}}(Z) \stackrel{\text{def}}{=} \{s \in W \ : \ t \in Z \textit{ for all states } t \textit{ with } s \curvearrowright t\}$$

7. $[\![\mathbf{EF}P]\!] = \mu Z.([\![P]\!] \cup \tau_{\mathbf{EX}}(Z))$ *where* $\mu Z.f(Z)$ *denotes the least fixpoint* Z *of the operation* $f(Z)$*, that is, the smallest set of states satisfying* $Z = f(Z)$.
8. $[\![\mathbf{EG}P]\!] = \nu Z.([\![P]\!] \cap \tau_{\mathbf{EX}}(Z))$ *where* $\nu Z.f(Z)$ *denotes the greatest fixpoint* Z *of the operation* $f(Z)$*, that is, the largest set of states satisfying* $Z = f(Z)$.
9. $[\![\mathbf{AF}P]\!] = \mu Z.([\![P]\!] \cup \tau_{\mathbf{AX}}(Z))$
10. $[\![\mathbf{AG}P]\!] = \nu Z.([\![P]\!] \cap \tau_{\mathbf{AX}}(Z))$
11. $[\![\mathbf{E}P\mathbf{U}Q]\!] = \mu Z.\big([\![Q]\!] \cup ([\![P]\!] \cap \tau_{\mathbf{EX}}(Z))\big)$
12. $[\![\mathbf{A}P\mathbf{U}Q]\!] = \mu Z.\big([\![Q]\!] \cup ([\![P]\!] \cap \tau_{\mathbf{AX}}(Z))\big)$

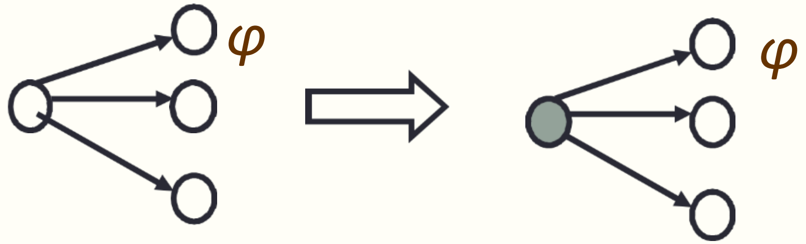
CTL Model Checking：Examples

5. $[\![\mathbf{EX}P]\!] = \tau_{\mathbf{EX}}([\![P]\!])$ *using the existential successor function* $\tau_{\mathbf{EX}}()$ *defined as follows:*

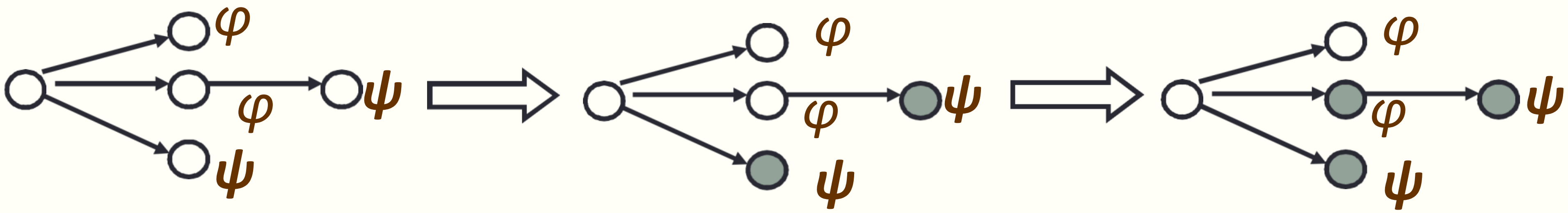
$$\tau_{\mathbf{EX}}(Z) \stackrel{\text{def}}{=} \{s \in W \ : \ t \in Z \text{ for some state } t \text{ with } s \curvearrowright t\}$$

11. $[\![\mathbf{E}P\mathbf{U}Q]\!] = \mu Z.\big([\![Q]\!] \cup ([\![P]\!] \cap \tau_{\mathbf{EX}}(Z))\big)$

- **EX φ:**
  - Label a state EX φ if any of its successors is labeled with φ



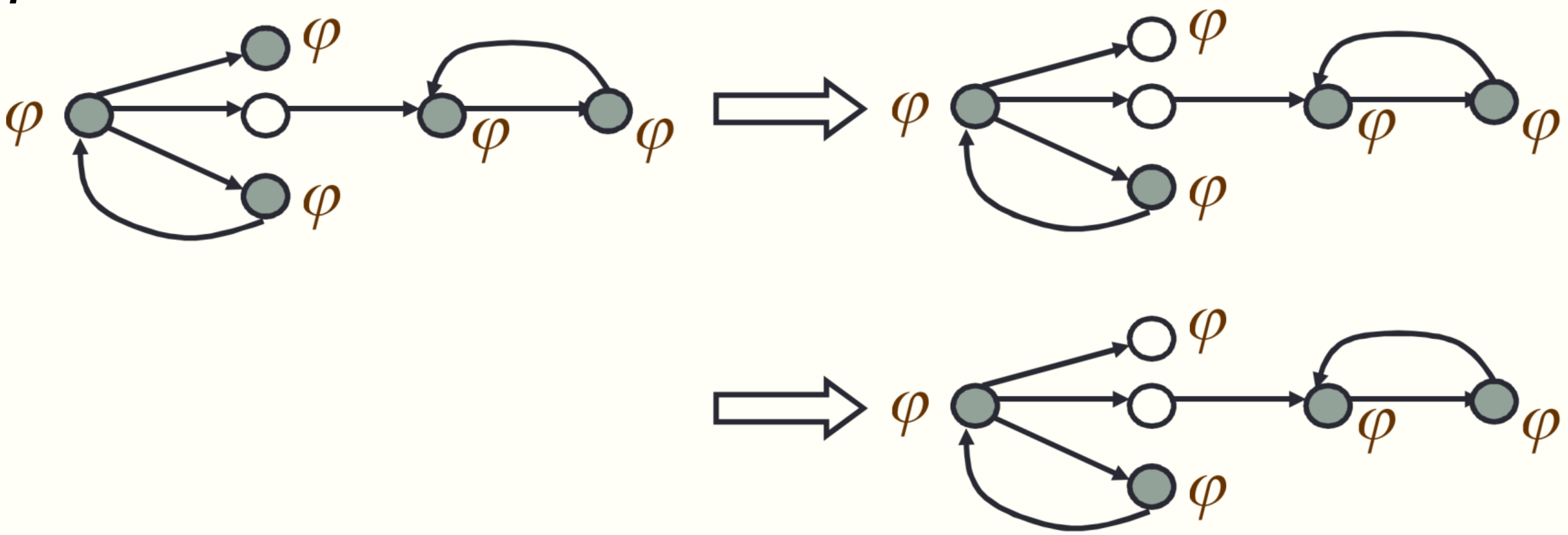
- **E φ U ψ**
  - Label a state E φ U ψ if it is labeled with ψ
  - Until there is no change: Label a state with E φ U ψ if it is labeled with φ and has a successor labeled with E φ U ψ



CTL Model Checking: Examples

8. $[\![\mathbf{EG}P]\!] = \nu Z.([\![P]\!] \cap \tau_{\mathbf{EX}}(Z))$ *where* $\nu Z.f(Z)$ *denotes the greatest fixpoint* Z *of the operation* $f(Z)$*, that is, the largest set of states satisfying* $Z = f(Z)$.

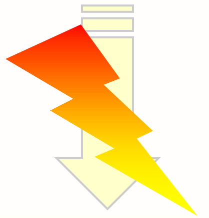
- **EG** $\boldsymbol{\varphi}$**:**
  - Label every node labeled with $\boldsymbol{\varphi}$ by EG $\boldsymbol{\varphi}$
  - Until there is no change
    - Remove label EG $\boldsymbol{\varphi}$ from any state that does not have successors labeled by EG $\boldsymbol{\varphi}$



The State Explosion Problem

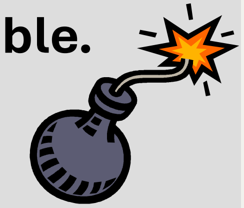
- **The idea behind CTL Model Checking:**
  - Exploit the finiteness of the state spaces of the Kripke structure K
- **However, states increase exponentially**

System Description



State Transition Graph

Combinatorial explosion of system states renders explicit model construction infeasible.



Exponential Growth of …

… global state space in number of concurrent components.

… memory states in memory size.

- **How to deal with this problem?**

Symbolic Model Checking

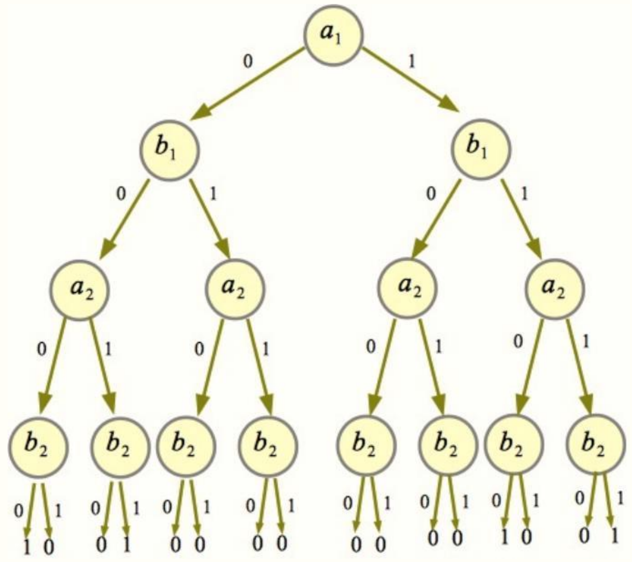
- **Main idea:**
  - Use predicates to denote sets of states of the Kripke Structure
  - the predicates have to be efficiently represented and manipulated
- **Two representative symbolic techniques:**
  - Binary Decision Diagrams (BDDs)
    - Express the transition relation by a formula, represented as BDD.
    - Manipulate these to compute logical operations and fixpoints
  - Boolean Satisfiability Problem (SAT)
    - Expand the transition relation a fixed number of steps (e.g., loop unrolling), resulting in a formula
    - For this unrolling, check whether the property holds
    - Continue increasing the unrolling until an error is found, resources are exhausted, or the diameter of the problem is reached
- **NuSMV: symbolic (un)bounded model checking**

Binary Decision Diagrams (BDDs)

❑Binary Decision Trees

- ➢a rooted directed tree with terminal and non-terminal vertices:
- ➢a non-terminal vertex v has a variable $var(v)$ and two successors, $low(v)$ and $high(v)$
- ➢a terminal vertex v has a value $value(v)$ in {0, 1}

$a1 \Leftrightarrow b1 \wedge a2 \Leftrightarrow b2$



❑Binary Decision Diagrams

- ➢a rooted, directed acyclic graph with terminal and non-terminal vertices. $value(v)$, $var(v)$, $low(v)$ and $high(v)$ are defined as for binary decision trees
- ➢Each vertex v associates with a boolean function $f_v(x_1, \cdots, x_n)$
- ➢v is a terminal vertex:
  - If $value(v) = 1$, then $f_v(x_1, \cdots, x_n) = 1$
  - If $value(v) = 0$, then $f_v(x_1, \cdots, x_n) = 0$
- ➢v is a non-terminal with $var(v) = x_i$:

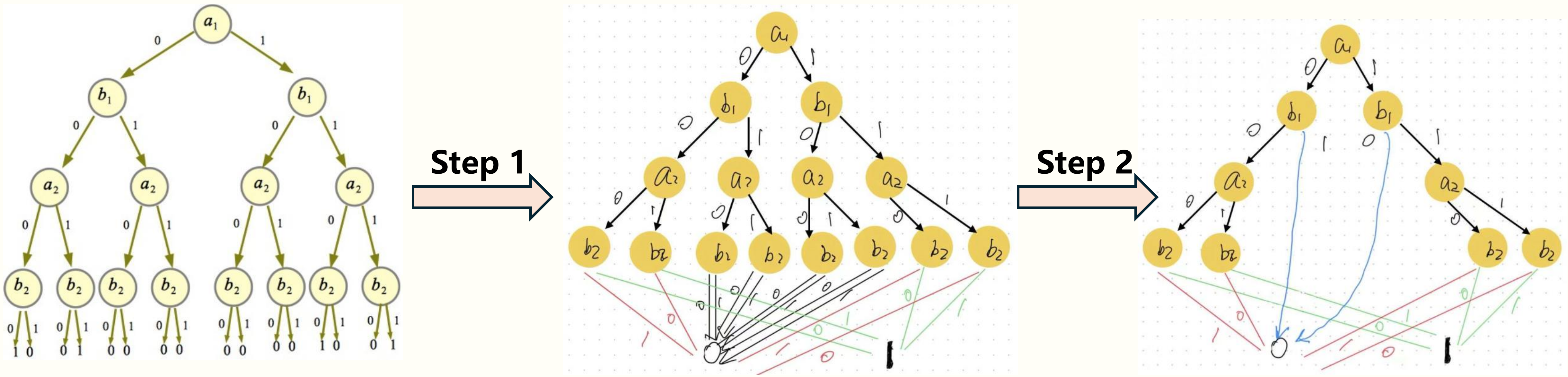
$$f_v(x_1, \cdots, x_n) = \left(\neg x_i \wedge f_{low(v)}(x_1, \cdots, x_n)\right) \vee (x_i \wedge f_{high(v)}(x_1, \cdots, x_n))$$

BDDs Canonicity

- **BDDs are in canonical form (Ordered BDDs) if:**
  - all BDDs have the same order on the variables along each path,
  - each BDD has no redundant vertices or isomorphic subtrees
- **Two Boolean functions are equivalent iff:**
  - two corresponding BDDs are isomorphic (同构)
- **Reduce Algorithm: transform a BDD into canonical form**
  - Do the following steps **repeatedly**:
    - Eliminating all terminal vertices but one of each value and redirecting arcs that used to point to deleted vertices to the kept terminal vertex with the same value
    - Eliminating duplicate non-terminals with the same variable, and the same low and high arcs (and redirect arcs)
    - If low(v) = high(v), eliminating v and redirecting arcs to low(v).

BDDs Canonicity: Example

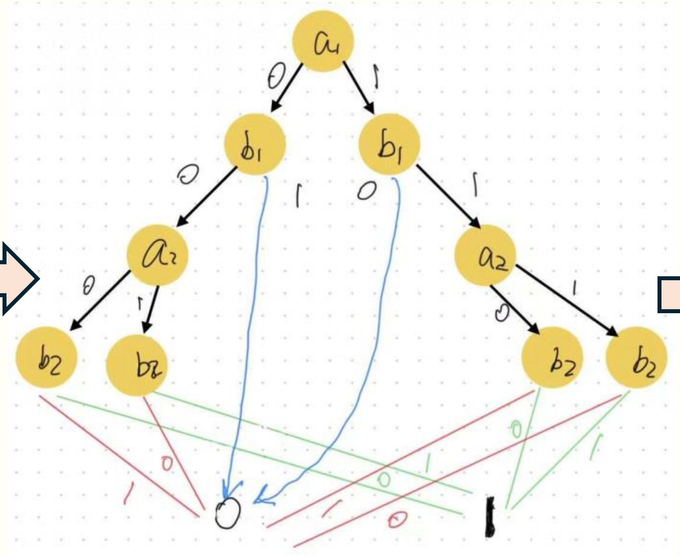
- Step 1: Eliminating all terminal vertices but one of each value and redirecting arcs that used to point to deleted vertices to the kept terminal vertex with the same value
- Step 2: If low(v) = high(v), eliminating v and redirecting arcs to low(v).



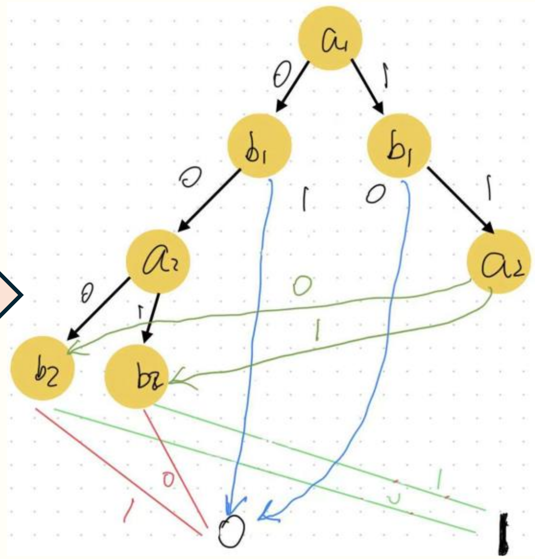
BDDs Canonicity: Example

- Step 3: Eliminating duplicate non-terminals with the same variable, and the same low and high arcs (and redirect arcs)

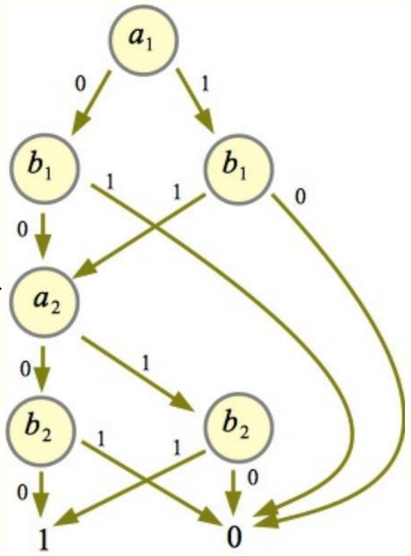
Step 2



Step 3



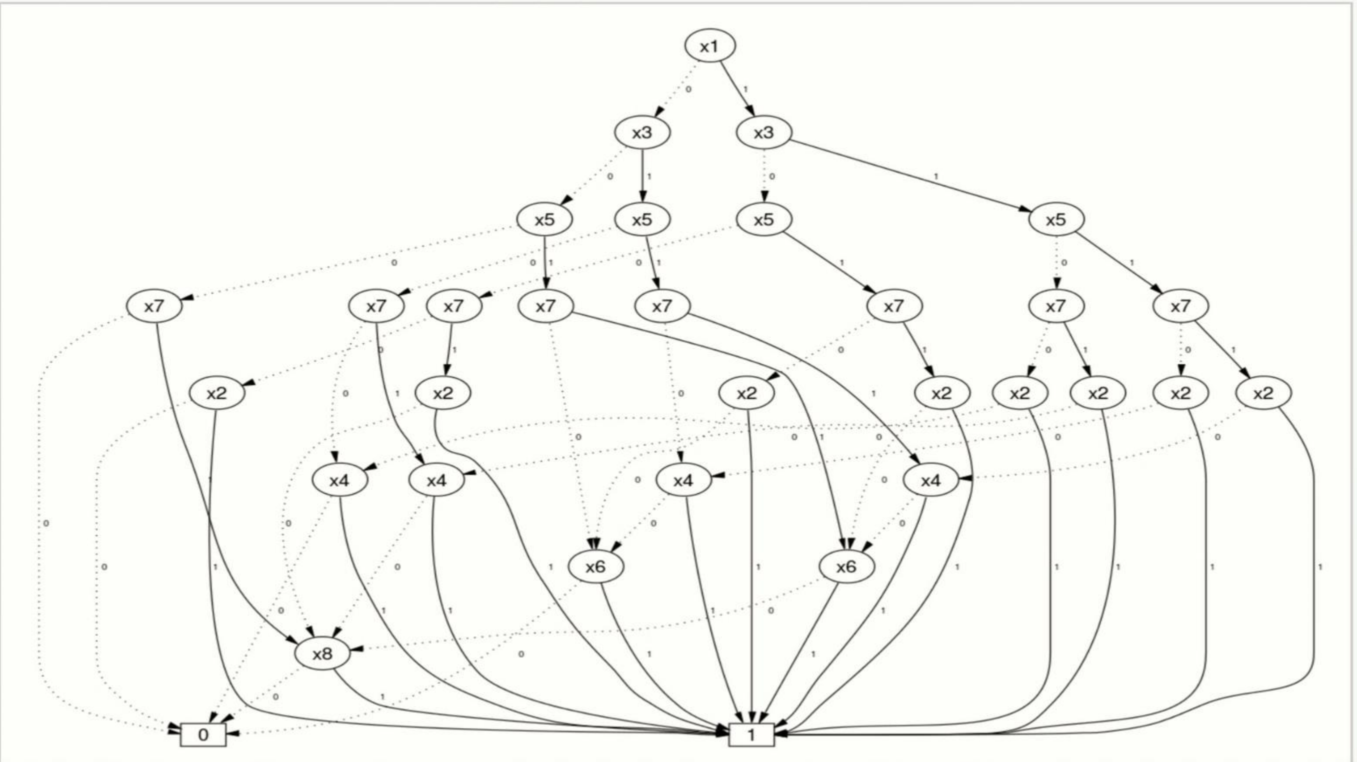
Step 3



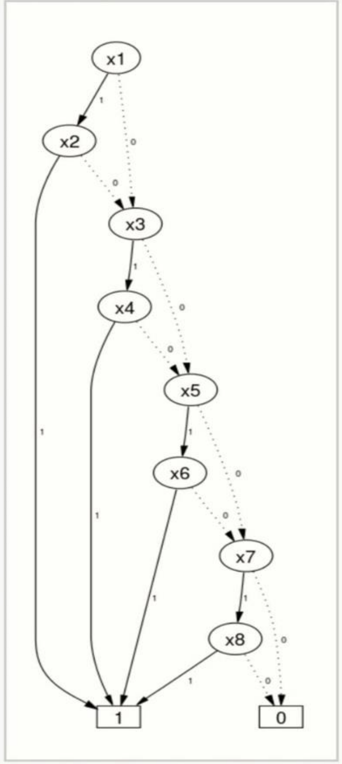
Canonical Form

BDDs：Variable Ordering is important

- ❑**Good orderings can be exponentially more compact**
  - ➢Finding a good ordering is NP-complete
  - ➢Exists formulas that have no non-exponential ordering



BDD for the function $f(x_1, \ldots, x_8) = x_1x_2 + x_3x_4 + x_5x_6 + x_7x_8$ using bad variable ordering



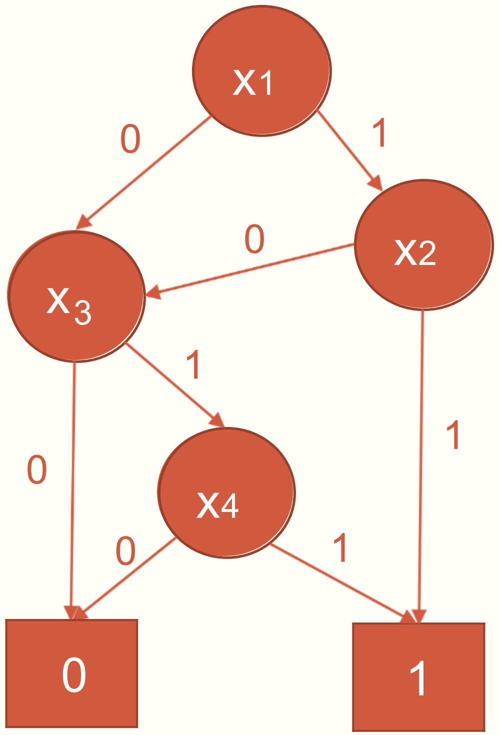
Good variable ordering

Logical Operations on BDDs

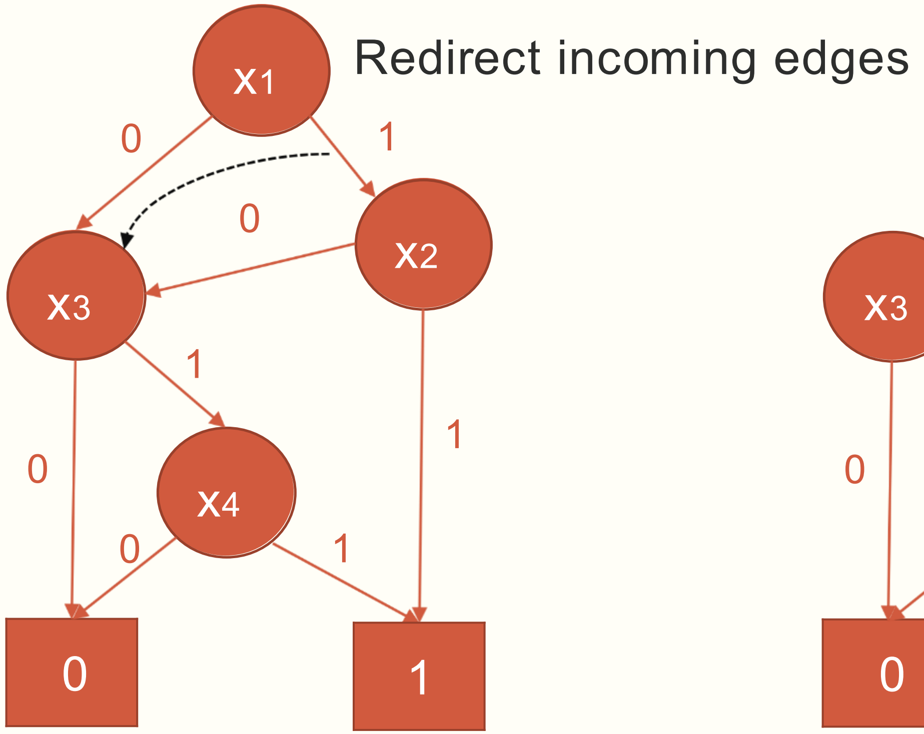
- **Shanon Expansion with respect to variable x:**
  - On f: $\mathrm{f} = (\neg \mathrm{x} \wedge \mathrm{f}_{|\mathrm{x} \leftarrow 0}) \vee (\mathrm{x} \wedge \mathrm{f}_{|\mathrm{x} \leftarrow 1})$
  - On $(f\ op\ g)$: $\mathrm{f\ op\ g} = (\neg \mathrm{x} \wedge (\mathrm{f}_{|\mathrm{x} \leftarrow 0}\ \mathrm{op}\ \mathrm{g}_{|\mathrm{x} \leftarrow 0})) \vee (\mathrm{x} \wedge (\mathrm{f}_{|\mathrm{x} \leftarrow 1}\ \mathrm{op}\ \mathrm{g}_{|\mathrm{x} \leftarrow 1}))$
    - op is a binary operation
- **Negation ($\neg \mathrm{f}$): swap leaves ($0 \rightarrow 1$ or $1 \rightarrow 0$)**
- **Arbitrary binary operation op:**
  - Let $\boldsymbol{f}_{|\boldsymbol{x} \leftarrow \mathbf{0}}$ (f is fixing x to be 0), similarly, $\boldsymbol{f}_{|\boldsymbol{x} \leftarrow 1}$
  - Apply Algorithm: starting from the roots u, v of the BDDs of f, g:
    - u, v are terminal vertices: $(f\ op\ g) = value(u)\ op\ value(v)$
    - $x = var(u) = var(v)$: $(f\ op\ g) = \neg x \wedge (f_{|x \leftarrow 0}\ op\ g_{|x \leftarrow 1}) \vee x \wedge (f_{|x \leftarrow 1}\ op\ g_{|x \leftarrow 1})$
    - $x = var(u) < var(v)$: $(f \wedge g) = \neg x \wedge (f_{|x \leftarrow 0} op\ g) \vee x \wedge (f_{|x \leftarrow 1} op\ g)$
    - $x = var(\mathrm{v}) < var(u)$: $(f\ op\ g) = \neg x \wedge (f\ op\ g_{|x \leftarrow 0}) \vee x \wedge (f\ op\ g_{|x \leftarrow 1})$
    - The algorithm is applied recursively, subproblems bounded by $|f| \cdot |g|$
  - The result is not necessarily canonical, may need to use reduce afterwards

Operations on BDDs

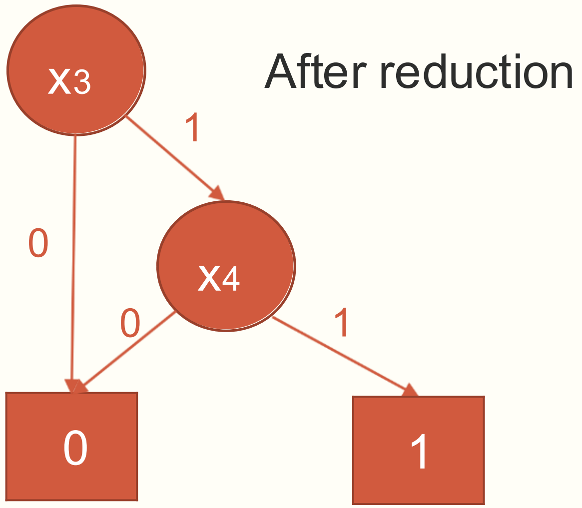
- **Cofactoring:**
  - $\boldsymbol{f}_{|x\leftarrow 0}$: replace all node v by its left sub-tree
  - $\boldsymbol{f}_{|x\leftarrow 1}$: replace all node v by its right sub-tree



$f = (x_1 \wedge x_2) \vee (x_3 \wedge x_4)$



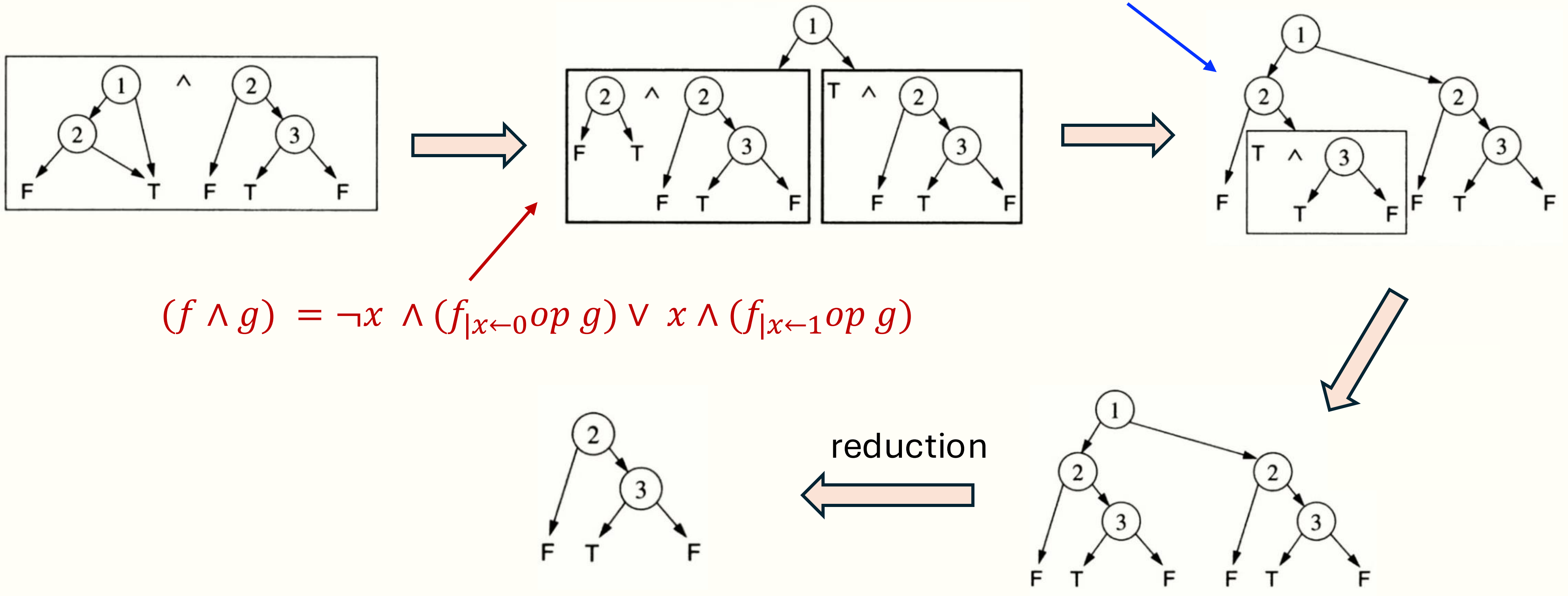
$f_{|x_1\leftarrow 0} = (x_1 \wedge x_2) \vee (x_3 \wedge x_4)\Big|_{x_1\leftarrow 0}$



Operations on BDDs: Example

- **ADD Operator:** $\mathrm{f} = x_1 \vee x_2, \mathrm{g} = x_2 \wedge \neg x_3$
  - Compute $f \wedge g$

$(f\ op\ g) = \neg x \wedge (f_{|x\leftarrow 0}\ op\ g_{|x\leftarrow 1}) \vee x \wedge (f_{|x\leftarrow 1}\ op\ g_{|x\leftarrow 1})$

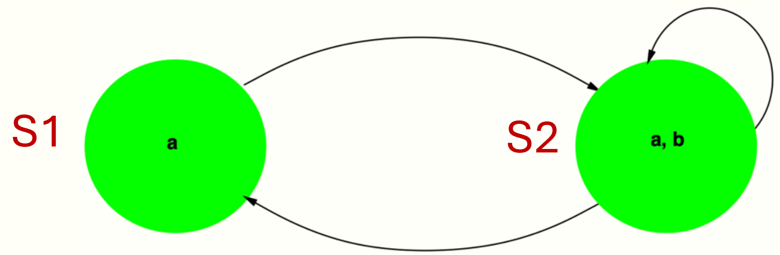


$(f \wedge g) = \neg x \wedge (f_{|x\leftarrow 0} op\ g) \vee x \wedge (f_{|x\leftarrow 1} op\ g)$

BDD-based Symbolic Model Checking

❑Represent state-transition graphs with BDDs

- ➢Use the values of n boolean state variables $v_1, v_2, \ldots, v_n$ to represent states of state-transition graphs



- Use 2 variable a, b
- S1: (a=1, b=0), or a ∧ ¬b
- S2: (a=1, b=1), or a ∧ b
- 所以每个state可以用一个BDD表示?

- ➢Transition relation R will be given as a Boolean formula:
  - $R(v_1, v_2, \ldots, v_n, v_1', v_2', \ldots, v_n')$:
  - $(v_1, v_2, \cdots, v_n)$: represents the current state
  - $(v_1', v_2', \cdots, v_n')$: represents the next state
  - Therefore, each transitcloserepresented by a BDD

- $R = R(S1, S2) \vee R(S2, S1) \vee R(S2, S2)$

- ➢CFL formula $f(v_1, v_2, \cdots, v_n)$: represents a set of states where f holds
  - If f is true under an assignment to variables v_i that represents a state S, then $S \vDash f$
  - E.g., Axiom Property a holds on S1 and S2, $a(\ldots) = S1 \vee S2 =$ a ∧ ¬b ∨ a ∧ b = a

Review: CTL Model Checking$[\![\phi]\!]$

1. $[\![p]\!] = \{s \in W \ : \ v(s)(p) = true\}$ *for atomic propositions* p
2. $[\![\neg P]\!] = W \setminus [\![P]\!]$
3. $[\![P \wedge Q]\!] = [\![P]\!] \cap [\![Q]\!]$
4. $[\![P \vee Q]\!] = [\![P]\!] \cup [\![Q]\!]$
5. $[\![\mathbf{EX}P]\!] = \tau_{\mathbf{EX}}([\![P]\!])$ *using the existential successor function* $\tau_{\mathbf{EX}}()$ *defined as follows:*

$$\tau_{\mathbf{EX}}(Z) \stackrel{\text{def}}{=} \{s \in W \ : \ t \in Z \text{ for some state } t \text{ with } s \curvearrowright t\}$$

6. $[\![\mathbf{AX}P]\!] = \tau_{\mathbf{AX}}([\![P]\!])$ *using the universal successor function* $\tau_{\mathbf{AX}}()$ *defined as follows:*

$$\tau_{\mathbf{AX}}(Z) \stackrel{\text{def}}{=} \{s \in W \ : \ t \in Z \text{ for all states } t \text{ with } s \curvearrowright t\}$$

7. $[\![\mathbf{EF}P]\!] = \mu Z.([\![P]\!] \cup \tau_{\mathbf{EX}}(Z))$ *where* $\mu Z.f(Z)$ *denotes the least fixpoint* Z *of the operation* $f(Z)$, *that is, the smallest set of states satisfying* $Z = f(Z)$.
8. $[\![\mathbf{EG}P]\!] = \nu Z.([\![P]\!] \cap \tau_{\mathbf{EX}}(Z))$ *where* $\nu Z.f(Z)$ *denotes the greatest fixpoint* Z *of the operation* $f(Z)$, *that is, the largest set of states satisfying* $Z = f(Z)$.
9. $[\![\mathbf{AF}P]\!] = \mu Z.([\![P]\!] \cup \tau_{\mathbf{AX}}(Z))$
10. $[\![\mathbf{AG}P]\!] = \nu Z.([\![P]\!] \cap \tau_{\mathbf{AX}}(Z))$
11. $[\![\mathbf{E}P\mathbf{U}Q]\!] = \mu Z.\big([\![Q]\!] \cup ([\![P]\!] \cap \tau_{\mathbf{EX}}(Z))\big)$
12. $[\![\mathbf{A}P\mathbf{U}Q]\!] = \mu Z.\big([\![Q]\!] \cup ([\![P]\!] \cap \tau_{\mathbf{AX}}(Z))\big)$

- APs are BDDs
- Transitions are BDDs
- Logical operations are on BDDs
- Therefore, the whole CTL Model Checking can be computed on BDDs

BDD-based Model Checking: Example

- **Model checking** $\mathrm{EF}\, p$
  - $\boldsymbol{EF}\, p \;=\; \boldsymbol{Lfp}\, \mathrm{U}.\, p \;\vee \boldsymbol{EX}\, \mathrm{U}$
  - Introduce state variables:
    - $\boldsymbol{EF}\, \mathrm{p} \;=\; \boldsymbol{Lfp}\, \mathrm{U}.\, \mathrm{p}(\bar{\mathrm{v}}) \vee \exists \bar{\mathrm{v}}’[R(\bar{\mathrm{v}}, \bar{\mathrm{v}}’) \wedge U(\bar{\mathrm{v}}’)]$
  - Compute the following sequence unitl convergence:
    - $\mathrm{U}_0(\bar{\mathrm{v}}), \mathrm{U}_1(\bar{\mathrm{v}}), \mathrm{U}_2(\bar{\mathrm{v}}), \cdots$
  - We obtain the convergent formula $U_k(\bar{\mathrm{v}})$
  - A state S is in U_k iff the valuation of $\bar{\mathrm{v}}$ corresponding to S makes $U_k(\bar{\mathrm{v}})$true

- BDD-based Model Checking can handle $\sim 10^{20}$ states

SAT-based Symbolic Model Checking

❑ Bounded Model Checking

- ➢ Sacrifice completeness for quick bug-finding
- ➢ Unroll the transition system
  - o Each variable $v \in V$ gets a new symbol for each time-step, e.g., v_k is at time k
  - o Space-time duality: unrolls temporal behavior into space
- ➢ For increasing values of k, check:

$$BMC(M, p, k) = Init(s_0) \wedge \bigwedge_{i=0}^{k-1} R(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{k} \neg p(s_i)$$

- ➢ If it is SAT (via SAT Solver), return FALSE
  - o Can construct a counter-example trace from the solver model

- ➢ This approach turns out to scale very well, but it can only guarantee correctness up to a given bound

Model Checking since 1981

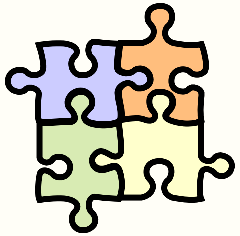
1981 Clarke / Emerson: CTL Model Checking
Sifakis / Quielle
1982 EMC: Explicit Model Checker
Clarke, Emerson, Sistla

10^5

1990s: Formal Hardware Verification in Industry: Intel, IBM, Motorola, etc.

1990 Symbolic Model Checking
Burch, Clarke, Dill, McMillan
1992 SMV: Symbolic Model Verifier
McMillan

10^{100}



1998 Bounded Model Checking using SAT
Biere, Clarke, Zhu
2000 Counterexample-guided Abstraction Refinement
Clarke, Grumberg, Jha, Lu, Veith

CBMC

10^{1000}

MAGIC

Some State oftheArt Model-Checkers

- **SMV, NuSMV, Cadence SMV**
  - CTL and LTL model-checkers
  - Based on BDDs or SAT solvers
  - Mostly for hardware and other models
- **Spin**
  - LTL model-checker
  - Explicit state exploration
  - Mostly for communication protocols
- **CBMC, SatAbs, CPAChecker, UFO**
  - Combine Model Checking and Abstraction
  - Work directly on the source code (mostly C)
  - Control-dependent properties of programs (buffer overflow, API usage, etc.)

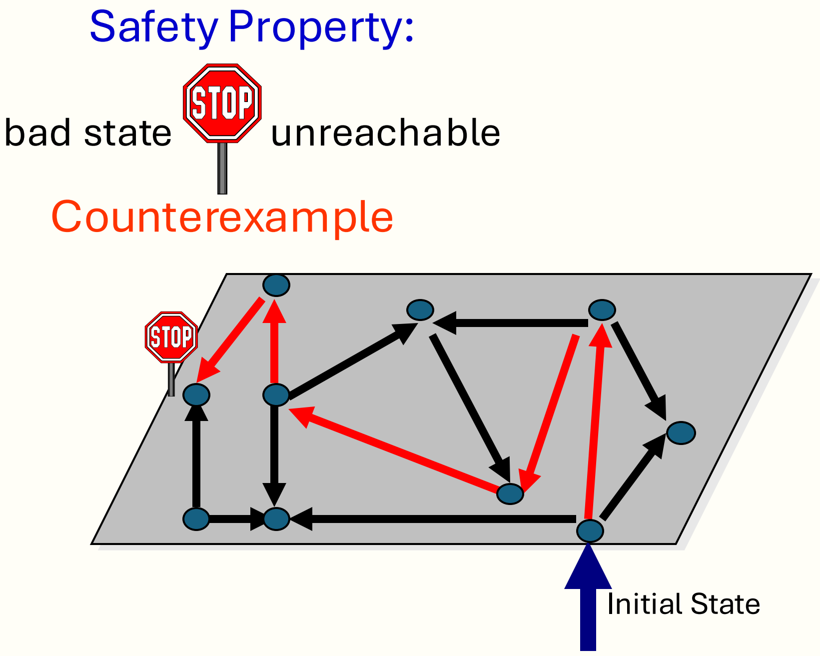
Pros and Cons of Model-Checking

❑Pros:

- ➢ No proofs! (algorithmic not deductive)
- ➢ Fast (compared to other rigorous methods)
- ➢ No problem with partial specifications
- ➢ Diagnostic counterexamples

❑Cons:

- ➢ Often cannot express full requirements
    - o Instead check several smaller properties
- ➢ Few systems can be checked directly
    - o Must generally abstract
- ➢ Works better for certain types of problems
    - o Very useful for control-centered concurrent systems
    - o E.g, Avionics software, Hardware, Communication protocols
- ➢ Not very good at data-centered systems:
    - o User interfaces, databases

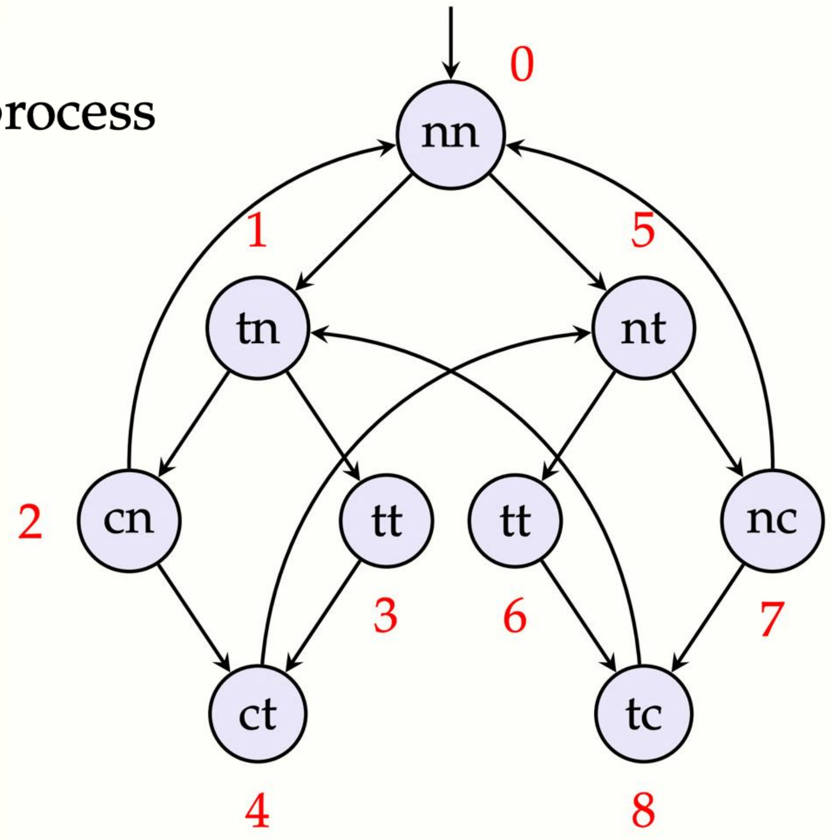


Many Industrial Successes

(Optional）作业：Mutual Exclusion

❑ **The following is a transition diagram of two processes:**

n noncritical section of an abstract process
t trying to enter critical section of an abstract process
c critical section of an abstract process



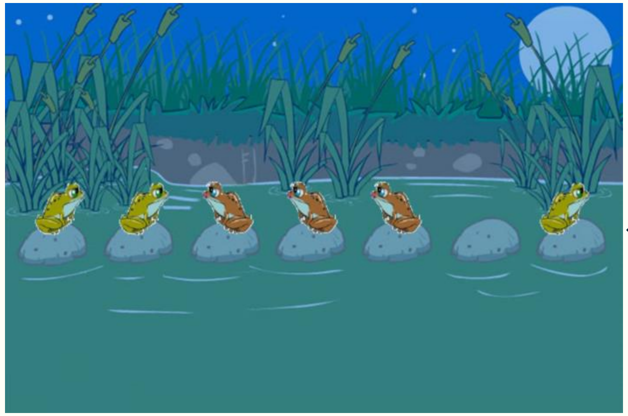
➢ Notation is the diagram:
- E.g, nt indicates $n_1 \wedge \mathrm{t}_2$, the process 1 is in the noncritical section while the process 2 is trying to get into its critical section.

➢ Check the following properties:
- Safety: $\neg\mathrm{EF}(\mathrm{c}_1 \wedge c_2)$
- Liveness: $\mathrm{AG}(t_1 \rightarrow \mathrm{AF}\ c_1) \ \wedge \mathrm{AG}(t_2 \rightarrow \mathrm{AF}\ c_2)$
- $1 \in [\![t_1 \rightarrow \mathrm{AF}\ c_1]\!]$ or $1 \in [\![\neg t_1 \vee \mathrm{AF}\ c_1]\!]$ or $1 \vDash t_1 \rightarrow \mathrm{AF}\ c_1$

(Optional) 作业：puzzling frogs in Spin

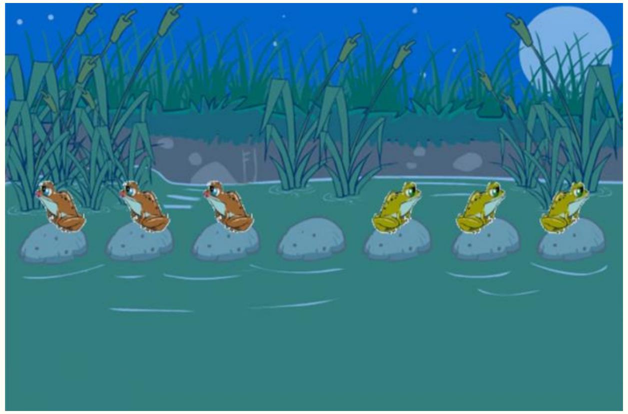
https://data.bangtech.com/algorithm/switch_frogs_to_the_opposite_side.htm

- **Build a model and a property whose counter-example (in SPIN) is a sequence of moves that allows all frogs to switch side**
  - 6 or 8 frogs, at most one on each rock, initially each half facing the other side from where it is sitting
  - can jump one step to the next rock if empty,
  - can jump over a rock if occupied and following is empty.



Stuck State

Initial State



Target State