Model Checking

Alcino Cunha

Motivation

- Concurrent and distributed systems are difficult to design and verify
- Correctness proofs typically require finding non-trivial invariants
- Can we automate verification?
  - Yes, but …

Mutual Exclusion

- A *mutual exclusion* concurrent algorithm ensures that
  - At most one process is in a critical section of code at the same time
- Can also provide other guarantees:
  - *No starvation* or *lockout freedom*: every process waiting to enter the critical section will eventually succeed
  - *Bounded waiting*: no process can enter the critical section more than *k* times while others are waiting (k = 1 equals *no takeover*)

Semaphore

```
int sem = 0;
```

```
while (true) {
  // idle
  while (testAndSet(sem) == 1);
  // critical
  sem = 0;
}
```

Peterson

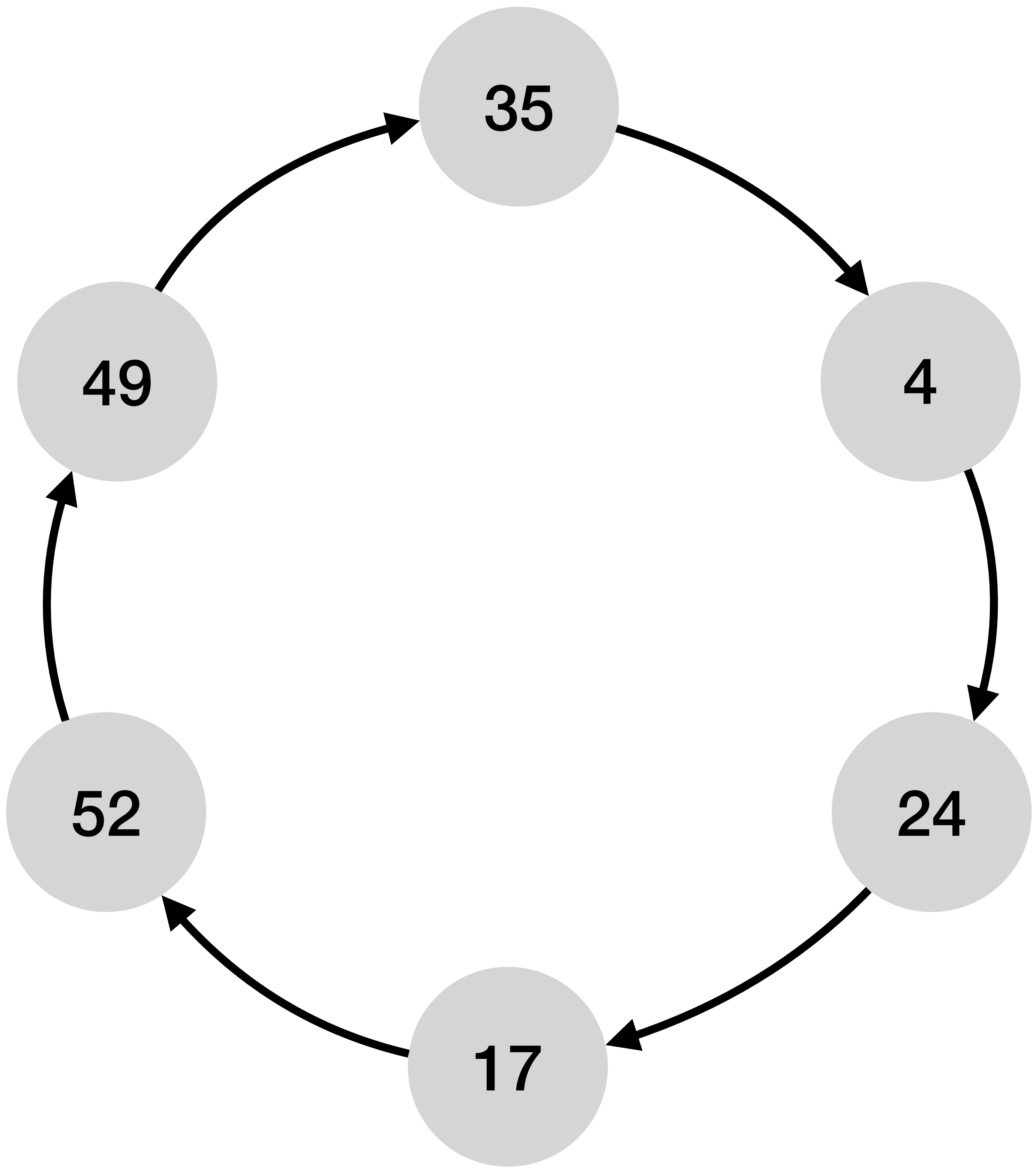
```
int level[N] = {-1, …, -1};
int last[N-1];
```

```
while (true) {
  // idle
  for (l = 0; l < N-1; l++) {
    level[i] = l;
    last[l] = i;
    while (last[l] == i && ∃ k . (k != i && level[k] >= l));
  }
  // critical
  level[i] = -1;
}
```

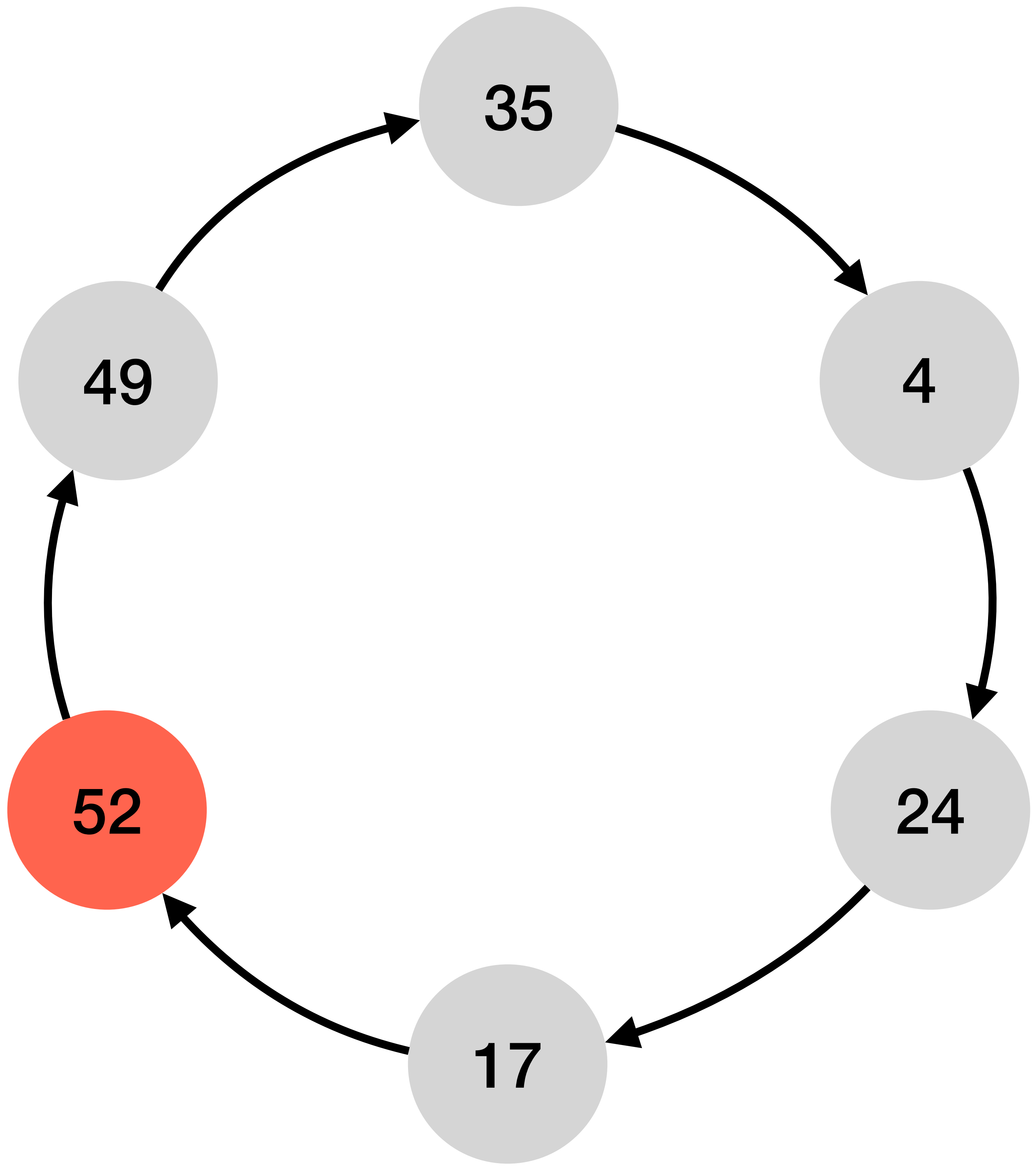
Leader election

- A leader election distributed algorithm ensures that
  - At most one leader will be elected
  - At least one leader will be elected
  - Any elected leader stays elected

Chang and Roberts



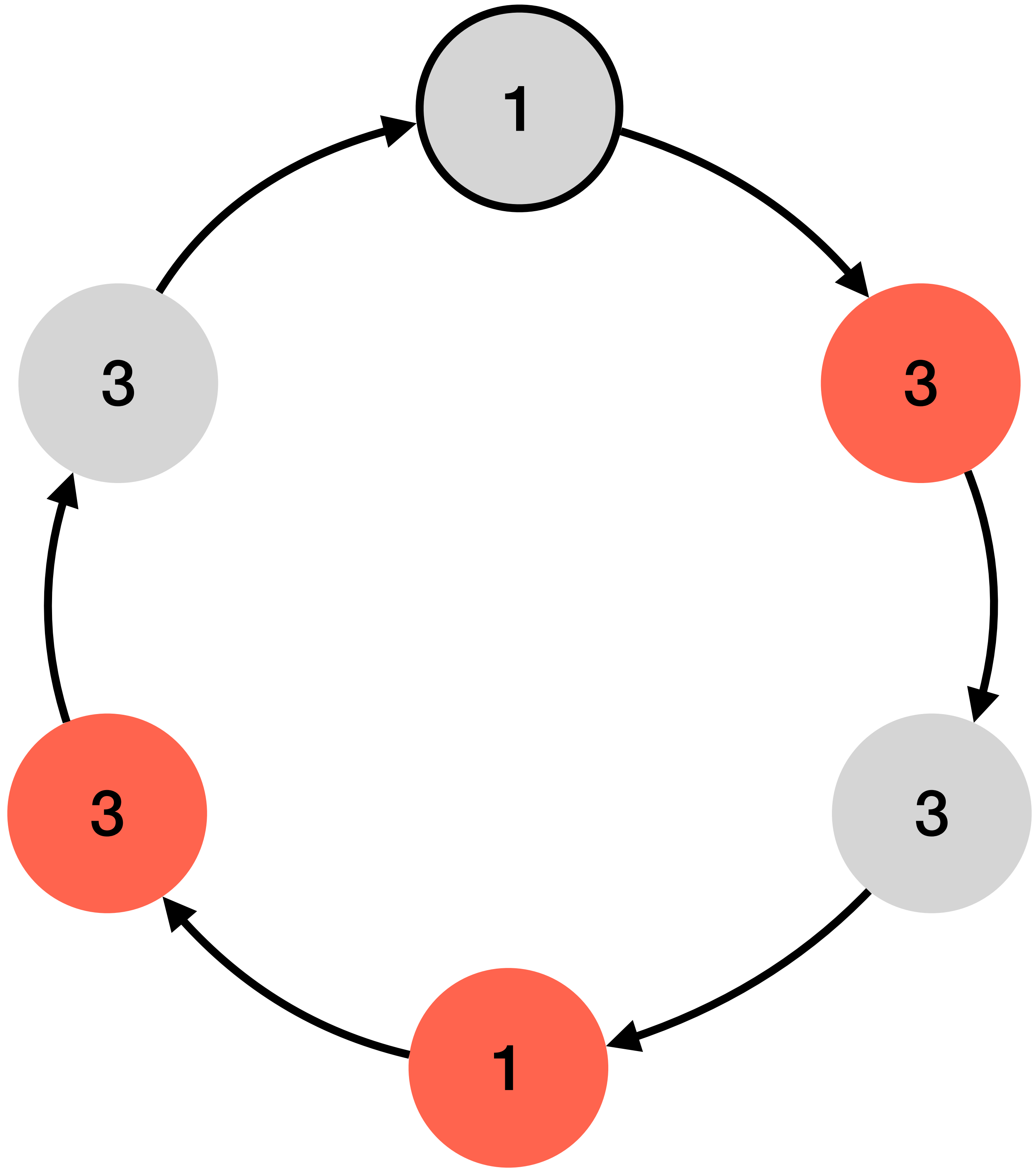
Chang and Roberts



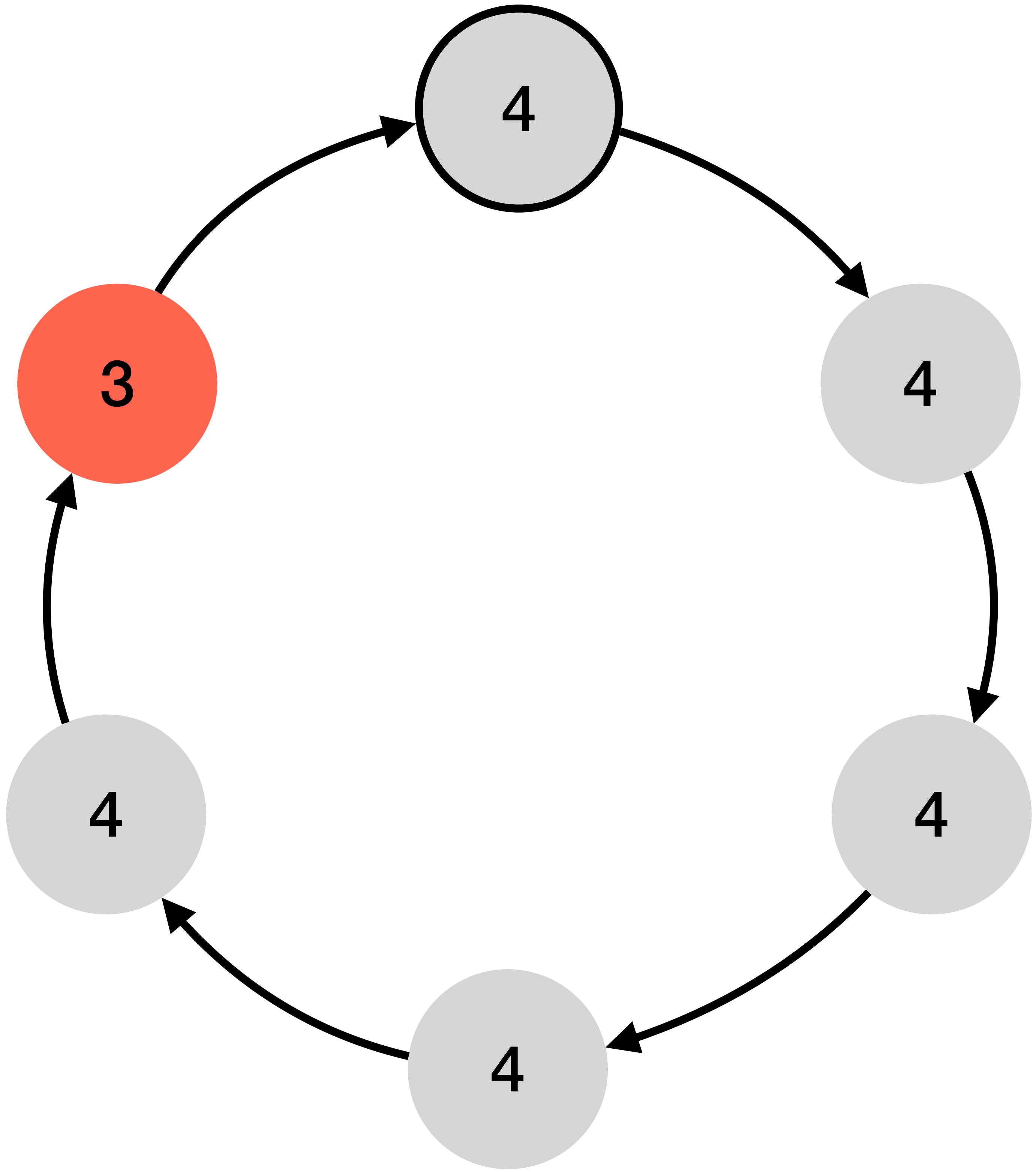
Self stabilisation

- A self stabilising distributed algorithm ensures
  - *Convergence*: starting from any state it will eventually reach a correct state
  - *Closure*: if the system is in a correct state it will stay in a correct state

Dijkstra



Dijkstra



Model Checking

- Model checking automates the verification process
- No need to find complex invariants
- But…
    - the system must described with a finite state model
    - and the desired properties formally specified using a temporal logic
- If the specification does not hold in the model, a counter-example is returned

Semaphore

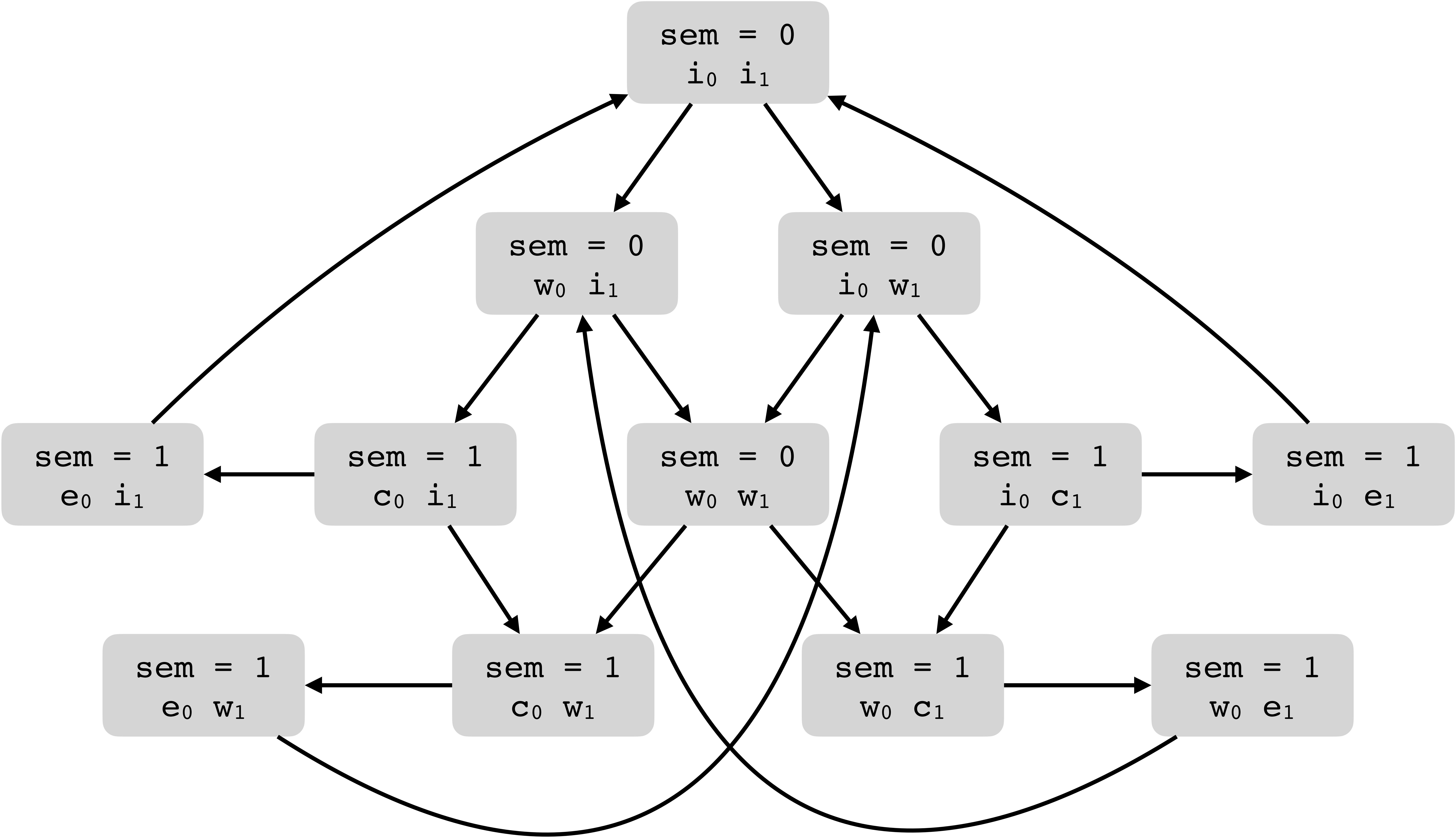
```
int sem = 0;
```

```
while (true) {
  i0: …
  w0: while (testAndSet(sem) == 1);
  c0: …
  e0: sem = 0;
}
```

||

```
while (true) {
  i1: …
  w1: while (testAndSet(sem) == 1);
  c1: …
  e1: sem = 0;
}
```

Semaphore



sem = 0
i0 i1
sem = 0
w0 i1
sem = 0
i0 w1
sem = 1
e0 i1
sem = 1
c0 i1
sem = 0
w0 w1
sem = 1
i0 c1
sem = 1
i0 e1
sem = 1
e0 w1
sem = 1
c0 w1
sem = 1
w0 c1
sem = 1
w0 e1

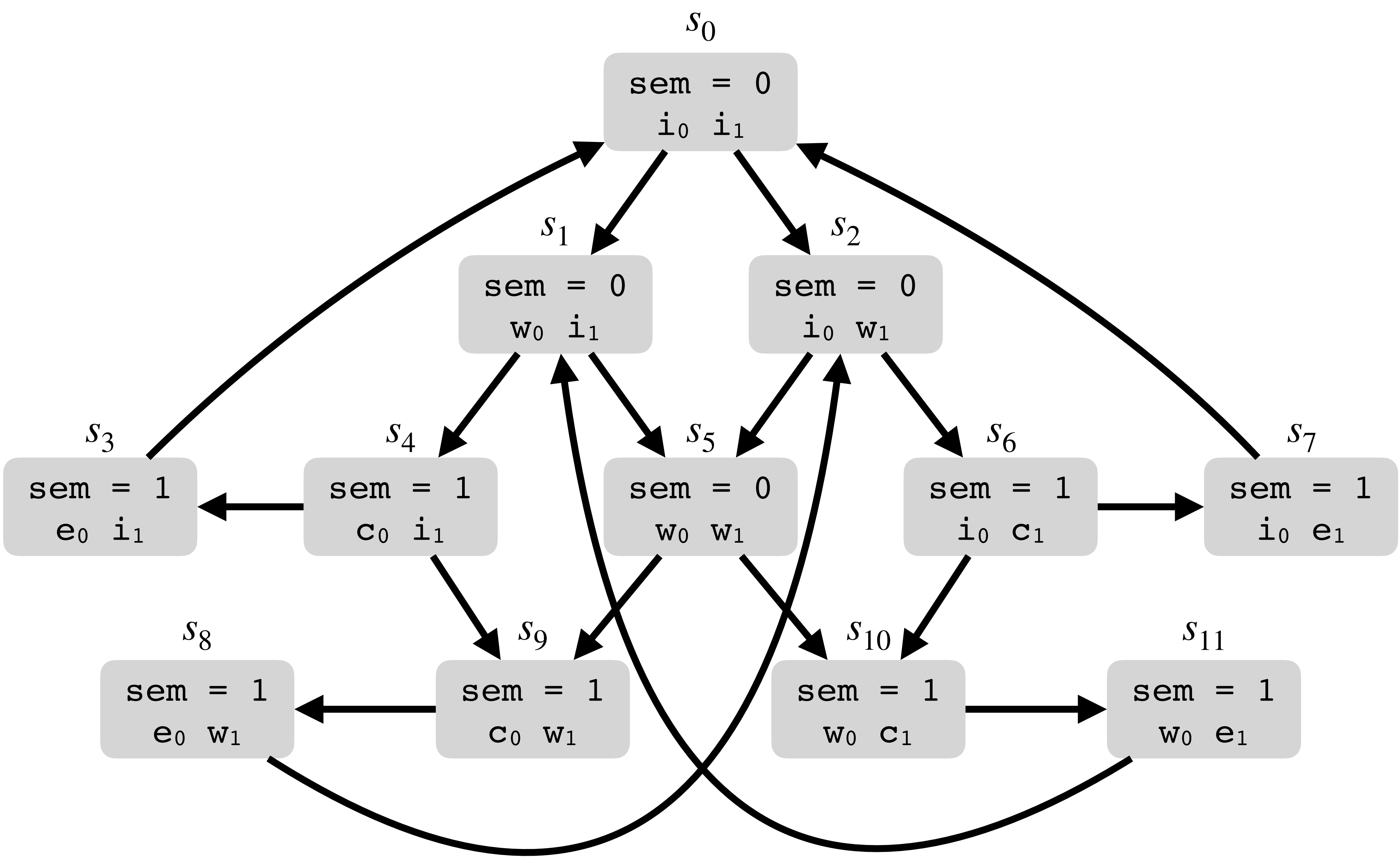
Kripke Structures

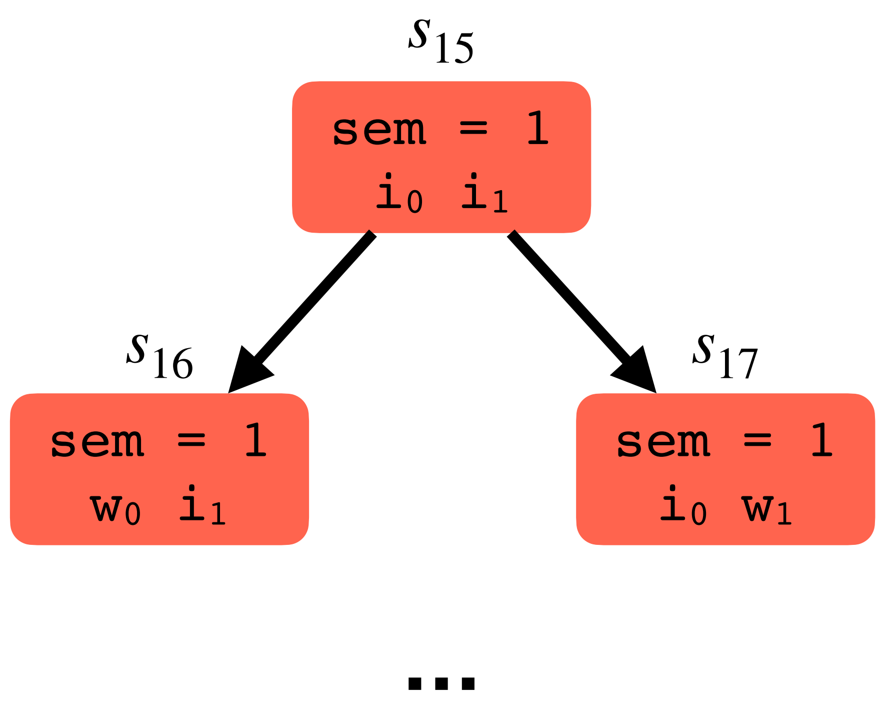
- Given a set A of atomic propositions, a *Kripke structure* M is a tuple (S, I, R, L) where:
  - S is a finite set of states
  - $I \subseteq S$ is the set of initial states
  - $R \subseteq S \times S$ is a total transition relation (every state has at least one successor)
  - $L : S \rightarrow 2^A$ is a labelling function, mapping each state $s \in S$ to the set of atomic propositions that are in s

Kripke Structures

- A *path* (or trace) π in a structure $M = (S, I, R, L)$ is an infinite sequence of states $s_0 s_1 s_2 \ldots$ such that $\forall i \geq 0 \cdot (s_i, s_{i+1}) \in R$
- Given a path π it's i-th state will be denoted by π_i and the path suffix starting in that state by π^i
- Abusing the notation, the set of all paths in M will also be denoted by M

Semaphore





$$S = \{s_0, s_1, s_2, s_3, \ldots, s_{15}, \ldots, s_{31}\}$$

$$I = \{s_0\}$$

$$R = \{(s_0, s_1), (s_0, s_2), (s_1, s_4), (s_1, s_5), \ldots, (s_{15}, s_{16}), \ldots\}$$

$$L = \{s_0 \mapsto \{\mathtt{sem} = 0, \mathtt{i}_0, \mathtt{i}_1\}, s_0 \mapsto \{\mathtt{sem} = 0, \mathtt{w}_0, \mathtt{i}_1\}, \ldots, s_{15} \mapsto \{\mathtt{sem} = 1, \mathtt{i}_0, \mathtt{i}_1\}, \ldots\}$$

Modelling

- *Modelling* is the act of defining the Kripke structure that describes a system
- Most model checkers have specific domain specific languages to do so

SMV Input Language

```
MODULE main
VAR
  sem : 0..1;
  pc : array 0..1 of {idle, wait, critical, exit};
IVAR
  proc : 0..1;
ASSIGN
  init(sem) := 0;
  init(pc[0]) := idle;
  init(pc[1]) := idle;
  next(sem) := case pc[proc] = wait & sem = 0: 1;
                    pc[proc] = exit : 0;
                    TRUE : sem;
               esac;
  next(pc[0]) := case proc = 0 & pc[0] = idle : wait;
                      proc = 0 & pc[0] = wait & sem = 0 : critical;
                      proc = 0 & pc[0] = critical : exit;
                      proc = 0 & pc[0] = exit : idle;
                      TRUE : pc[0];
                 esac;
  next(pc[1]) := case proc = 1 & pc[1] = idle : wait;
                      proc = 1 & pc[1] = wait & sem = 0 : critical;
                      proc = 1 & pc[1] = critical : exit;
                      proc = 1 & pc[1] = exit : idle;
                      TRUE : pc[1];
                 esac;
```

PlusCal

```
----------------------------- MODULE Semaphore -----------------------------
(*
--algorithm Semaphore {

    variable sem = 0;

    process (proc \in {0,1}) {
        idle: while (TRUE) {
                  skip;
            wait: await (sem = 0);
                  sem := 1;
            crit: skip;
            exit: sem := 0;
        }
    }
}
*)
=============================================================================
```

Validation

- *Validation* is the act of checking if the model correctly describes the system under analysis
- Its an inherently manual activity, few automated support

nuXmv

```
% nuXmv -int
nuXmv > read_model -i semaphore.smv
nuXmv > flatten_hierarchy
nuXmv > encode_variables
nuXmv > build_model
nuXmv > pick_state -i
***************  AVAILABLE STATES  *************
  ================= State =================
  0) -------------------------
  sem = 0
  pc[0] = idle
  pc[1] = idle
There's only one available state. Press Return to Proceed.
Chosen state is: 0
nuXmv > simulate -k 3 -i
********  Simulation Starting From State 1.1   ********
***************  AVAILABLE STATES  *************
  ================= State =================
  sem = 0
  pc[0] = idle
  pc[1] = wait
    This state is reachable through:
    0) -------------------------
    proc = 1
  ================= State =================
  pc[0] = wait
  pc[1] = idle
    This state is reachable through:
    1) -------------------------
    proc = 0
Choose a state from the above (0-1):
```

TLA+ Toolbox

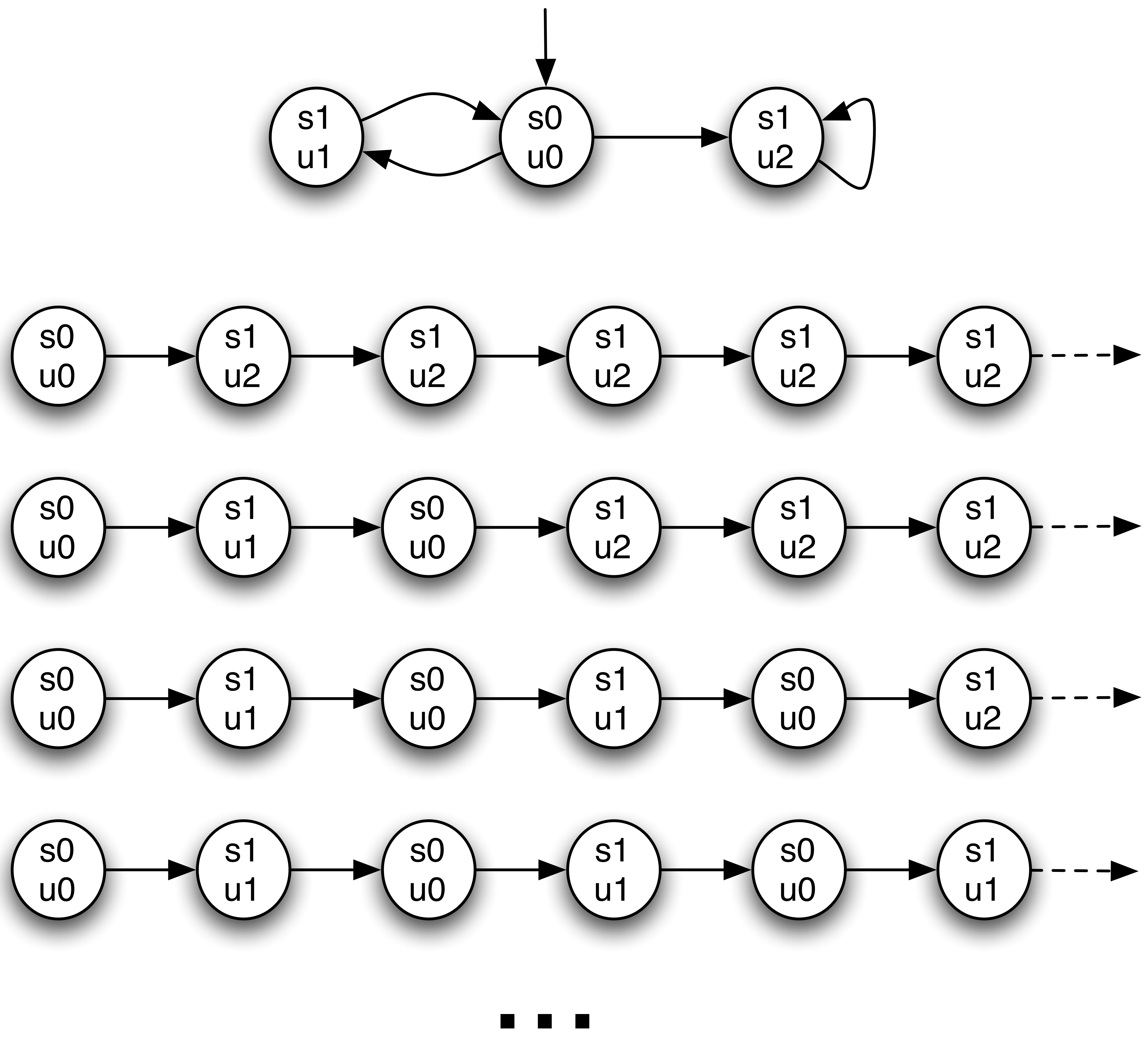
Specification

- *Specification* is the act of formalising the desired requirements in *temporal logic*

Models of Time

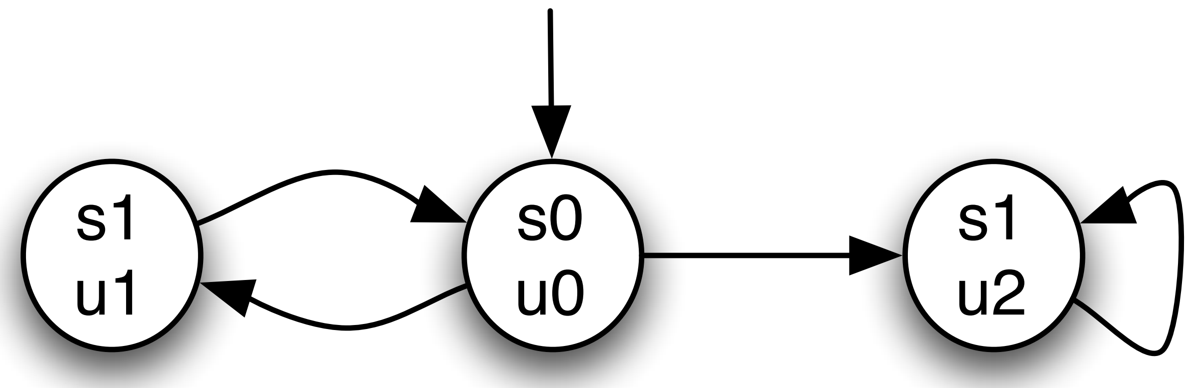
- There are two basic models of time in temporal logic:
  - *Linear Time*: the behaviour of the system is the set of all infinite paths starting in initial states.
  - *Branching Time*: the behaviour of the system is the set of all infinite computation trees unrolled from initial states.
- Both can be determined from a Kripke structure

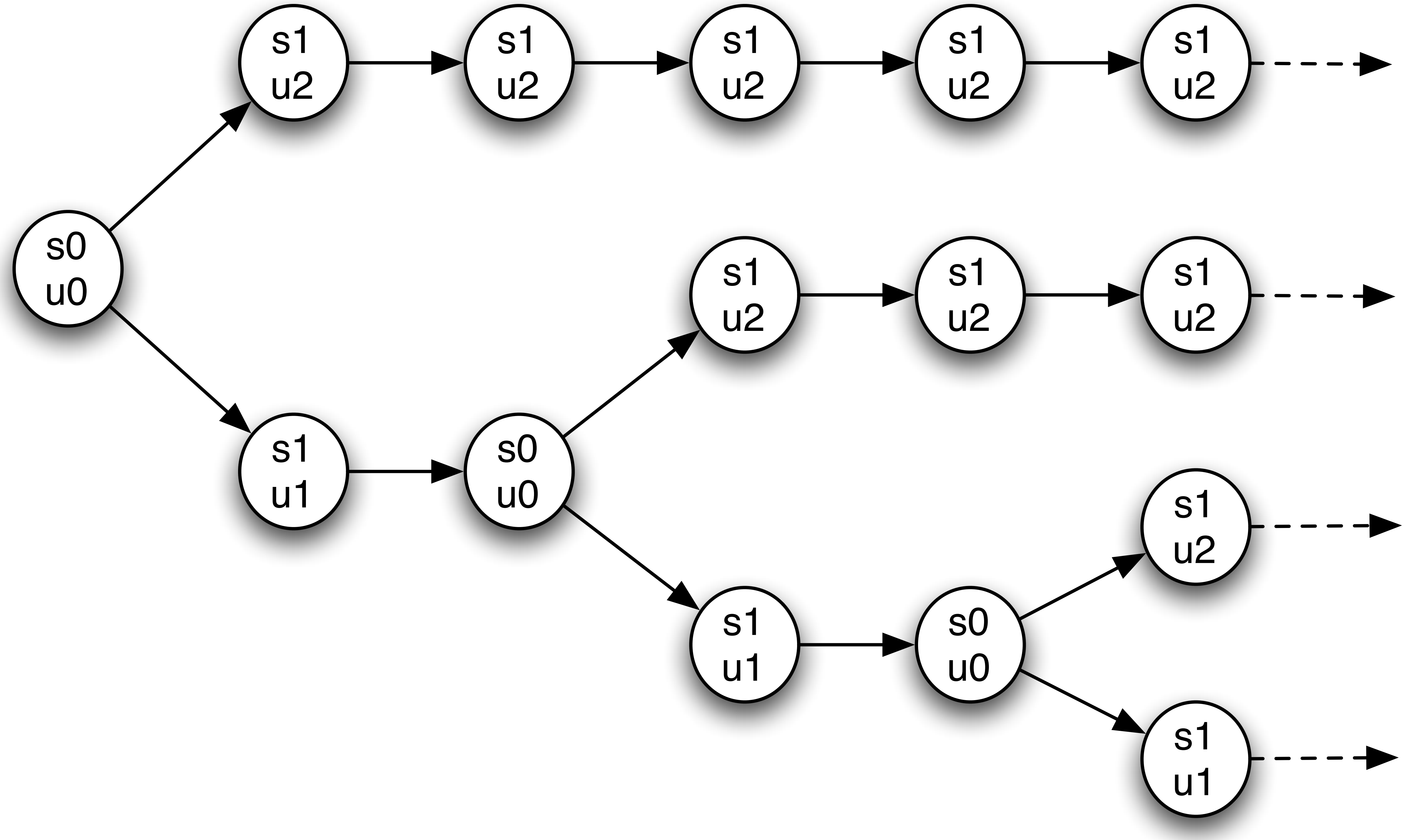
Linear Time



s1 u1
s0 u0
s1 u2
s0 u0
s1 u2
s1 u2
s1 u2
s1 u2
s1 u2
s0 u0
s1 u1
s0 u0
s1 u2
s1 u2
s1 u2
s0 u0
s1 u1
s0 u0
s1 u1
s0 u0
s1 u2
s0 u0
s1 u1
s0 u0
s1 u1
s0 u0
s1 u1
...

Branching Time





Linear Temporal Logic

- LTL is a temporal logic with a linear model of time
- All LTL formulas are evaluated in infinite paths
- Given a set A of atomic propositions, the syntax of LTL formulas is given by the following rules
  - If $p \in A$, then p is an atomic LTL formula
  - If f and g are LTL formulas, then $\top$, $\bot$, $\neg f$, $f \vee g$, $f \wedge g$, $f \rightarrow g$, X f, F f, G f, f U g, and g R f are LTL formulas

LTL Temporal Operators

X f	$\bigcirc f$	neXt, after	f will be true in the next state
G f	$\square f$	Globaly, always	f will always be true
F f	$\Diamond f$	Future, eventually	f will eventually be true
f U g		Until	g will be true and f is true until then
g R f		Release	f can only be false after g becomes true

LTL Semantics

- Given a Kripke structure $M = (S, I, R, L)$ we will denote the fact that LTL formula f holds in M by $M \models f$

$$M \models f \quad \Leftrightarrow \quad \forall \pi \in M \cdot \pi_0 \in I \rightarrow M, \pi \models f$$

LTL Semantics

$$
\begin{array}{lcl}
M, \pi \models \top & & \\
M, \pi \not\models \bot & & \\
M, \pi \models p & \Leftrightarrow & p \in L(\pi_0) \\
M, \pi \models \neg f & \Leftrightarrow & M, \pi \not\models f \\
M, \pi \models f \vee g & \Leftrightarrow & M, \pi \models f \text{ or } M, \pi \models g \\
M, \pi \models f \wedge g & \Leftrightarrow & M, \pi \models f \text{ and } M, \pi \models g \\
M, \pi \models f \rightarrow g & \Leftrightarrow & M, \pi \not\models f \text{ or } M, \pi \models g \\
M, \pi \models \mathsf{X}\, f & \Leftrightarrow & M, \pi^1 \models f \\
M, \pi \models \mathsf{F}\, f & \Leftrightarrow & \exists i \geq 0 \cdot M, \pi^i \models f \\
M, \pi \models \mathsf{G}\, f & \Leftrightarrow & \forall i \geq 0 \cdot M, \pi^i \models f \\
M, \pi \models f \;\mathsf{U}\; g & \Leftrightarrow & \exists i \geq 0 \cdot M, \pi^i \models g \wedge \forall 0 \leq j < i \cdot M, \pi^j \models f \\
M, \pi \models g \;\mathsf{R}\; f & \Leftrightarrow & \forall i \geq 0 \cdot M, \pi^i \models f \vee \exists 0 \leq j < i \cdot M, \pi^j \models g
\end{array}
$$

Minimal LTL Operators

- All LTL formulas can be expressed with $\top$, $\neg$, $\vee$, X, and U

$\bot$	$\equiv$	$\neg \top$
$f \wedge g$	$\equiv$	$\neg (\neg f \vee \neg g)$
$f \rightarrow g$	$\equiv$	$\neg f \vee g$
$\mathsf{F}\, f$	$\equiv$	$\top \,\mathsf{U}\, f$
$\mathsf{G}\, f$	$\equiv$	$\neg \mathsf{F} \neg f$
$g \,\mathsf{R}\, f$	$\equiv$	$\neg (\neg g \,\mathsf{U}\, \neg f)$

LTL Examples

- Mutual exclusion

$$\mathsf{G}\ \neg(\mathtt{c}_0 \wedge \mathtt{c}_1)$$

- Lockout freedom

$$\mathsf{G}\ (\mathtt{w}_0 \rightarrow \mathsf{F}\ \mathtt{c}_0) \wedge \mathsf{G}\ (\mathtt{w}_1 \rightarrow \mathsf{F}\ \mathtt{c}_1)$$

- No takeover

$$\mathsf{G}\ (\mathtt{w}_0 \wedge \neg\mathtt{c}_1 \rightarrow (\mathtt{c}_0\ \mathsf{R}\ \neg\mathtt{c}_1)) \wedge \mathsf{G}\ (\mathtt{w}_0 \wedge \mathtt{c}_1 \rightarrow ((\mathtt{c}_0\ \mathsf{R}\ \neg\mathtt{c}_1)\ \mathsf{R}\ \mathtt{c}_1)) \wedge \ldots$$

SMV Input Language

```
LTLSPEC
  G !(pc[0] = crit & pc[1] = crit)

LTLSPEC
  G (pc[0] = wait -> F pc[0] = crit) &
  G (pc[1] = wait -> F pc[1] = crit)

LTLSPEC
  G (pc[0] = wait & pc[1] != crit ->
        (pc[0] = crit V pc[1] != crit)) & …
```

TLA+

```
[] ~(pc[0] = "crit" /\ pc[1] = "crit")

[] (pc[0] = "wait" => <> (pc[0] = "crit")) /\
[] (pc[1] = "wait" => <> (pc[1] = "crit"))
```

Specifying Behaviour with LTL'

- LTL is expressive enough to specify the valid behaviours of a Kripke structure
- For a boolean variable b, we can define $b' = a$ as an abbreviation of $\mathsf{X}\ b \leftrightarrow a$ and likewise for other (bounded) variables
- A standard LTL extension is to support the prime operator on variables, to denote the value of the variable in the next state
- The valid behaviours can then be specified with a formula $init \wedge \mathsf{G}\ trans$
  - $init$ is a propositional formula that specifies what are the valid initial states
  - $trans$ a propositional formula (with primes) that specifies what are the valid transitions
- Thus, the Kripke structure could be left unconstrained and instead of checking f we check $init \wedge \mathsf{G}\ trans \rightarrow f$

Semaphore

$$S = \{s_0, s_1, s_2, s_3, \ldots, s_{15}, \ldots, s_{31}\}$$
$$I = S$$
$$R = R \times R$$
$$L = \{s_0 \mapsto \{\texttt{sem} = 0, \texttt{i}_0, \texttt{i}_1\}, s_0 \mapsto \{\texttt{sem} = 0, \texttt{w}_0, \texttt{i}_1\}, \ldots, s_{15} \mapsto \{\texttt{sem} = 1, \texttt{i}_0, \texttt{i}_1\}, \ldots\}$$

$$\begin{array}{rcc}
\textit{init} & \equiv & \texttt{sem} = 0 \wedge \texttt{i}_0 \wedge \texttt{i}_1 \wedge \neg\texttt{w}_0 \wedge \neg\texttt{w}_1 \wedge \neg\texttt{c}_0 \wedge \neg\texttt{c}_1 \wedge \neg\texttt{e}_0 \wedge \neg\texttt{e}_1 \\
\textit{trans} & \equiv & \textit{idle}_0 \vee \textit{idle}_1 \vee \textit{wait}_0 \vee \textit{wait}_1 \vee \textit{crit}_0 \vee \textit{crit}_1 \vee \textit{exit}_0 \vee \textit{exit}_1 \\
\textit{idle}_0 & \equiv & \texttt{i}_0 \wedge \mathsf{X}\ \neg\texttt{i}_0 \wedge \mathsf{X}\ \texttt{w}_0 \wedge \texttt{c}_0' = \texttt{c}_0 \wedge \texttt{e}_0' = \texttt{e}_0 \wedge \texttt{i}_1' = \texttt{i}_1 \wedge \texttt{w}_1' = \texttt{w}_1 \wedge \texttt{c}_1' = \texttt{c}_1 \wedge \texttt{e}_1' = \texttt{e}_1 \wedge \texttt{sem}' = \texttt{sem} \\
\textit{wait}_0 & \equiv & \texttt{w}_0 \wedge \texttt{sem} = 0 \wedge \mathsf{X}\ \neg\texttt{w}_0 \wedge \mathsf{X}\ \texttt{c}_0 \wedge \mathsf{X}\ \texttt{sem} = 1 \wedge \texttt{i}_0' = \texttt{i}_0 \wedge \texttt{e}_0' = \texttt{e}_0 \wedge \texttt{i}_1' = \texttt{i}_1 \wedge \texttt{w}_1' = \texttt{w}_1 \wedge \texttt{c}_1' = \texttt{c}_1 \wedge \texttt{e}_1' = \texttt{e}_1 \\
\cdots & \cdots & \cdots
\end{array}$$

$$(\textit{init} \wedge \mathsf{G}\ \textit{trans}) \rightarrow \mathsf{G}\ \neg(\texttt{c}_0 \wedge \texttt{c}_1)$$

SMV Input Language

```
MODULE main
VAR
  sem : 0..1;
  pc : array 0..1 of {idle, wait, crit, exit};
DEFINE
  idle0 := pc[0] = idle &           next(pc[0]) = wait & next(sem) = sem & next(pc[1]) = pc[1];
  wait0 := pc[0] = wait & sem = 0 & next(pc[0]) = crit & next(sem) = 1   & next(pc[1]) = pc[1];
  crit0 := pc[0] = crit &           next(pc[0]) = exit & next(sem) = sem & next(pc[1]) = pc[1];
  exit0 := pc[0] = exit &           next(pc[0]) = idle & next(sem) = 0   & next(pc[1]) = pc[1];
  idle1 := pc[1] = idle &           next(pc[1]) = wait & next(sem) = sem & next(pc[0]) = pc[0];
  wait1 := pc[1] = wait & sem = 0 & next(pc[1]) = crit & next(sem) = 1   & next(pc[0]) = pc[0];
  crit1 := pc[1] = crit &           next(pc[1]) = exit & next(sem) = sem & next(pc[0]) = pc[0];
  exit1 := pc[1] = exit &           next(pc[1]) = idle & next(sem) = 0   & next(pc[0]) = pc[0];
  start := sem = 0 & pc[0] = idle & pc[1] = idle;
  trans := idle0 | wait0 | crit0 | exit0 | idle1 | wait1 | crit1 | exit1;
LTLSPEC
  start & G trans -> G !(pc[0] = crit & pc[1] = crit)
```

SMV Input Language

```
MODULE main
VAR
  sem : 0..1;
  pc : array 0..1 of {idle, wait, crit, exit};
DEFINE
  idle0 := pc[0] = idle &           next(pc[0]) = wait & next(sem) = sem & next(pc[1]) = pc[1];
  wait0 := pc[0] = wait & sem = 0 & next(pc[0]) = crit & next(sem) = 1   & next(pc[1]) = pc[1];
  crit0 := pc[0] = crit &           next(pc[0]) = exit & next(sem) = sem & next(pc[1]) = pc[1];
  exit0 := pc[0] = exit &           next(pc[0]) = idle & next(sem) = 0   & next(pc[1]) = pc[1];
  idle1 := pc[1] = idle &           next(pc[1]) = wait & next(sem) = sem & next(pc[0]) = pc[0];
  wait1 := pc[1] = wait & sem = 0 & next(pc[1]) = crit & next(sem) = 1   & next(pc[0]) = pc[0];
  crit1 := pc[1] = crit &           next(pc[1]) = exit & next(sem) = sem & next(pc[0]) = pc[0];
  exit1 := pc[1] = exit &           next(pc[1]) = idle & next(sem) = 0   & next(pc[0]) = pc[0];
INIT
  sem = 0 & pc[0] = idle & pc[1] = idle;
TRANS
  idle0 | wait0 | crit0 | exit0 | idle1 | wait1 | crit1 | exit1;
LTLSPEC
  G !(pc[0] = crit & pc[1] = crit)
```

TLA+

```
---------------------------- MODULE Semaphore ----------------------------
VARIABLES sem, pc

Init == /\ sem = 0
        /\ pc = [p \in {0,1} |-> "idle"]

idle(p) == /\ pc[p] = "idle"
           /\ pc' = [pc EXCEPT ![p] = "wait"]
           /\ sem' = sem
wait(p) == /\ pc[p] = "wait"
           /\ sem = 0
           /\ sem' = 1
           /\ pc' = [pc EXCEPT ![p] = "crit"]
crit(p) == /\ pc[p] = "crit"
           /\ pc' = [pc EXCEPT ![p] = "exit"]
           /\ sem' = sem
exit(p) == /\ pc[p] = "exit"
           /\ sem' = 0
           /\ pc' = [pc EXCEPT ![p] = "idle"]

Next == \/ idle(0) \/ wait(0) \/ crit(0) \/ exit(0)
        \/ idle(1) \/ wait(1) \/ crit(1) \/ exit(1)

Spec == Init /\ [][Next]_<<sem,pc>>
=============================================================================
```

Past Time LTL

Y f	Yesterday, before	f was true in the previous state
H f	Historically	f was always true
O f	Once	f was once true
f S g	Since	g was once true and f is true since then
g T f	Triggers	f could only be false before g becomed true

Temporal Logic of Actions

- Restricts LTL' to ensure formulas are *stutter invariant*
- Stutter invariance is fundamental to check *refinement*, that is, to check that one specification is an implementation of a more abstract specification
- TLA also adds first order quantifiers to LTL'
- TLA+ is the full concrete specification language based on TLA

TLA Syntax

- The syntax of TLA formulas is given by the following rules
  - Any state predicate p (without primes) is an atomic TLA formula
  - If a is an *action* predicate (one with primes), then $\Box\, [a]_t$ and ENABLED a are atomic TLA formulas, being $[a]_t \equiv a \vee (t' = t)$
  - If f and g are TLA formulas and S is a set, then TRUE, FALSE, $\neg f$, $f \wedge g, f \vee g, f \Rightarrow g$, $\forall x \in S : f$, $\exists x \in S : f$, $\Box\, f$, $\Diamond\, f$ are TLA formulas

TLA+

```
----------------------------- MODULE Semaphore -----------------------------
EXTENDS Naturals

CONSTANT N
ASSUME N > 0

VARIABLES sem, pc

Proc == 0..(N-1)

Init == /\ sem = 0
        /\ pc = [p \in Proc |-> "idle"]

idle(p) == /\ pc[p] = "idle"
           /\ pc' = [pc EXCEPT ![p] = "wait"]
           /\ sem' = sem
wait(p) == /\ pc[p] = "wait"
           /\ sem = 0
           /\ sem' = 1
           /\ pc' = [pc EXCEPT ![p] = "crit"]
crit(p) == /\ pc[p] = "crit"
           /\ pc' = [pc EXCEPT ![p] = "exit"]
           /\ sem' = sem
exit(p) == /\ pc[p] = "exit"
           /\ sem' = 0
           /\ pc' = [pc EXCEPT ![p] = "idle"]

Next == \E p \in Proc : idle(p) \/ wait(p) \/ crit(p) \/ exit(p)

Spec == Init /\ [][Next]_<<sem,pc>>
=============================================================================
```

TLA+

```
[] (\A p \in Proc : pc[p] = "crit" =>
       (\A q \in Proc : pc[q] # "crit" \/ p = q))

\A p \in Proc : [] (pc[p] = "wait" =>
                          <> (pc[p] = "crit"))
```

Validation with TLA+

- To find scenarios where f holds just check $\neg f$
- In particular to see a scenario where action a happens check $\Box[\neg a]_t$
- It is also common to include an invariant to check type correctness

```
TypesOK == /\ sem \in {0,1}
           /\ pc \in [Proc -> {"idle","wait","crit","exit"}]
```

DEMO

Computation Tree Logic

- CTL is a temporal logic with a branching model of time
- Besides temporal operators it also has path quantifiers, that build state formulas out of path formulas
- Given a set A of atomic propositions, the syntax of CTL formulas is given by the following rules
  - If $p \in A$, then p is an CTL state formula
  - If f and g are CTL state formulas, then ⊤, ⊥, $\neg f$, $f \vee g$, $f \wedge g$, and $f \rightarrow g$ are CTL state formulas
  - If f is a CTL path formula, then A f and E f are CTL state formulas
  - If f and g are CTL state formulas, then X f, F f, G f, f U g, and g R f are CTL path formulas

CTL Semantics

- Given a Kripke structure $M = (S, I, R, L)$ we will denote the fact that a CTL (state) formula f holds in M by $M \vDash f$

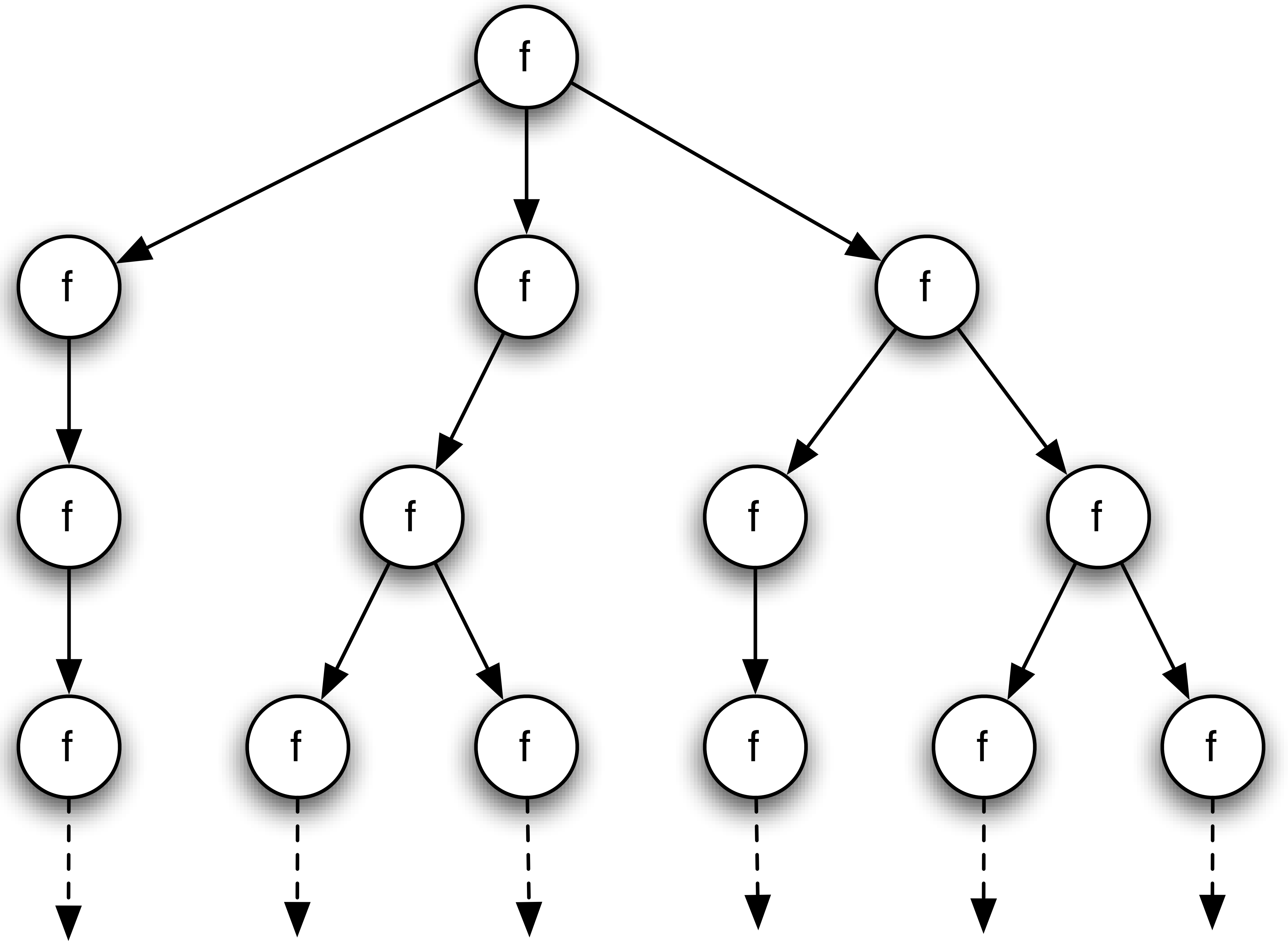
$$M \vDash f \quad \Leftrightarrow \quad \forall s \in I \cdot M, s \vDash f$$

CTL Semantics

$M, s \models \top$		
$M, s \not\models \bot$		
$M, s \models p$	$\Leftrightarrow$	$p \in L(s)$
$M, s \models \neg f$	$\Leftrightarrow$	$M, s \not\models f$
$M, s \models f \vee g$	$\Leftrightarrow$	$M, s \models f$ or $M, s \models g$
$M, s \models f \wedge g$	$\Leftrightarrow$	$M, s \models f$ and $M, s \models g$
$M, s \models f \rightarrow g$	$\Leftrightarrow$	$M, s \not\models f$ or $M, s \models g$
$M, s \models \mathsf{A}\, f$	$\Leftrightarrow$	$\forall \pi \in M \cdot \pi_0 = s \rightarrow M, \pi \models f$
$M, s \models \mathsf{E}\, f$	$\Leftrightarrow$	$\exists \pi \in M \cdot \pi_0 = s \rightarrow M, \pi \models f$
$M, \pi \models \mathsf{X}\, f$	$\Leftrightarrow$	$M, \pi_1 \models f$
$M, \pi \models \mathsf{F}\, f$	$\Leftrightarrow$	$\exists i \geq 0 \cdot M, \pi_i \models f$
$M, \pi \models \mathsf{G}\, f$	$\Leftrightarrow$	$\forall i \geq 0 \cdot M, \pi_i \models f$
$M, \pi \models f\ \mathsf{U}\ g$	$\Leftrightarrow$	$\exists i \geq 0 \cdot M, \pi_i \models g \wedge \forall 0 \leq j < i \cdot M, \pi_j \models f$
$M, \pi \models g\ \mathsf{R}\ f$	$\Leftrightarrow$	$\forall i \geq 0 \cdot M, \pi_i \models f \vee \exists 0 \leq j < i \cdot M, \pi_j \models g$

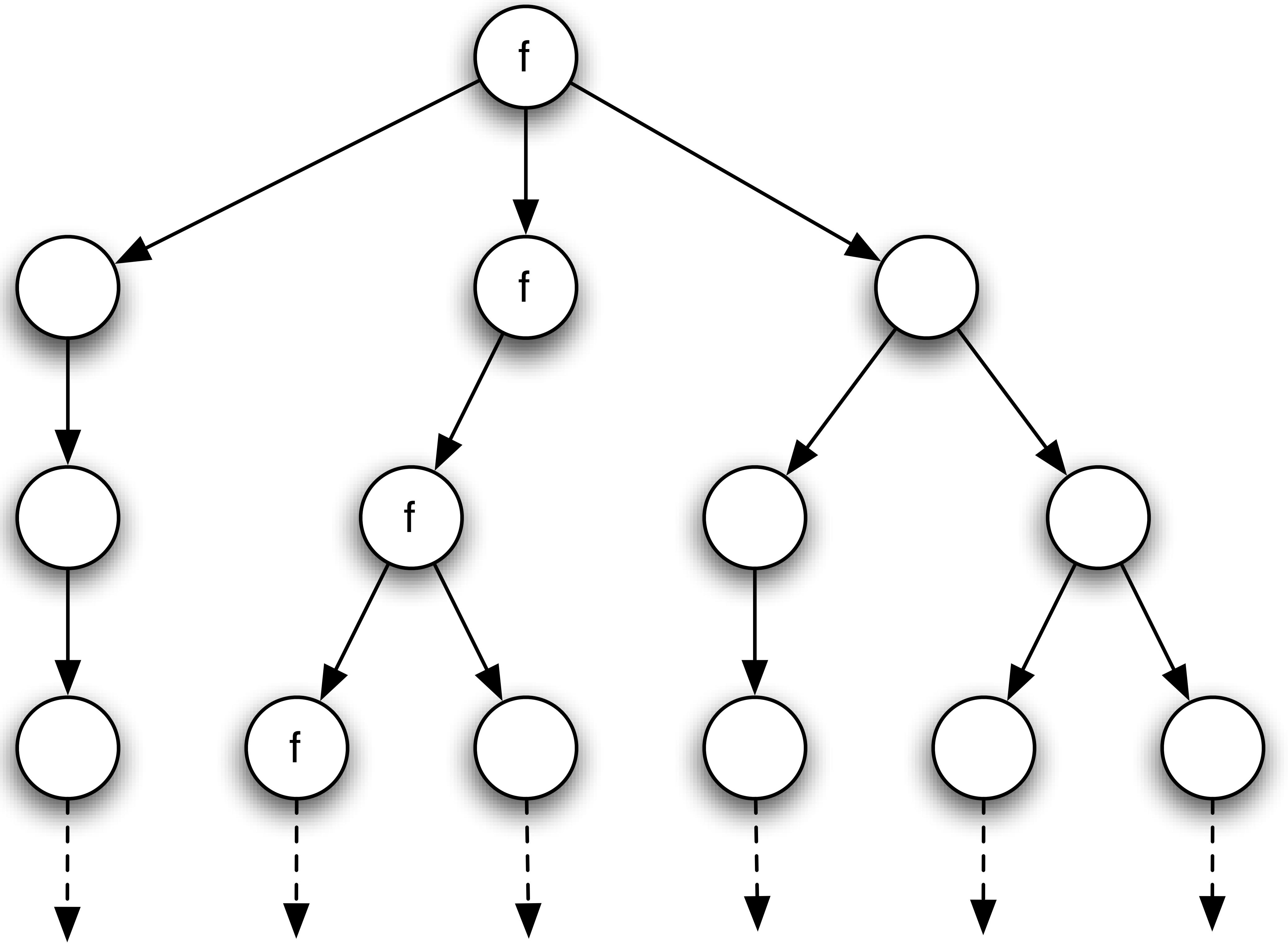
CTL Semantics

AG f



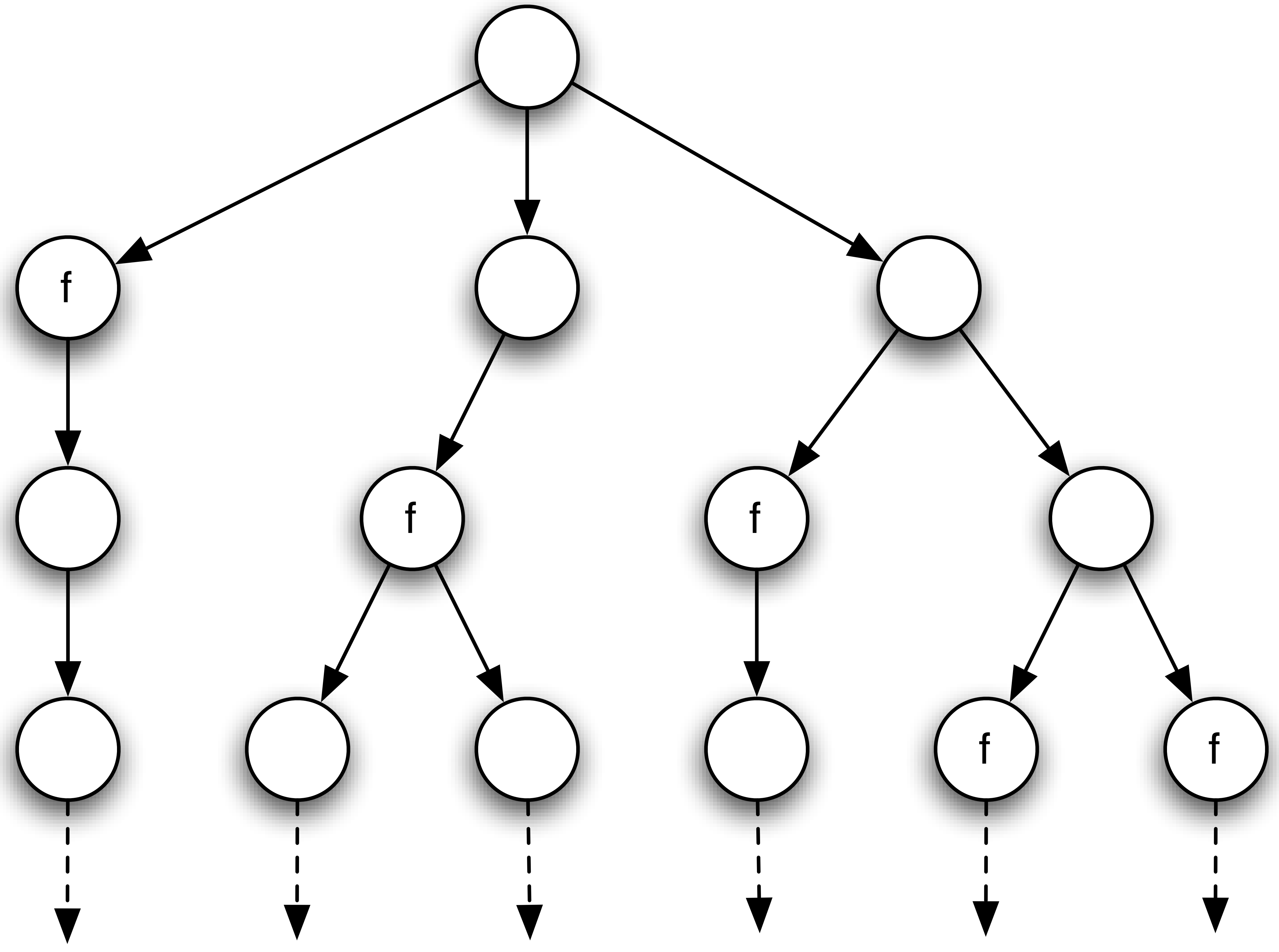
CTL Semantics

EG f



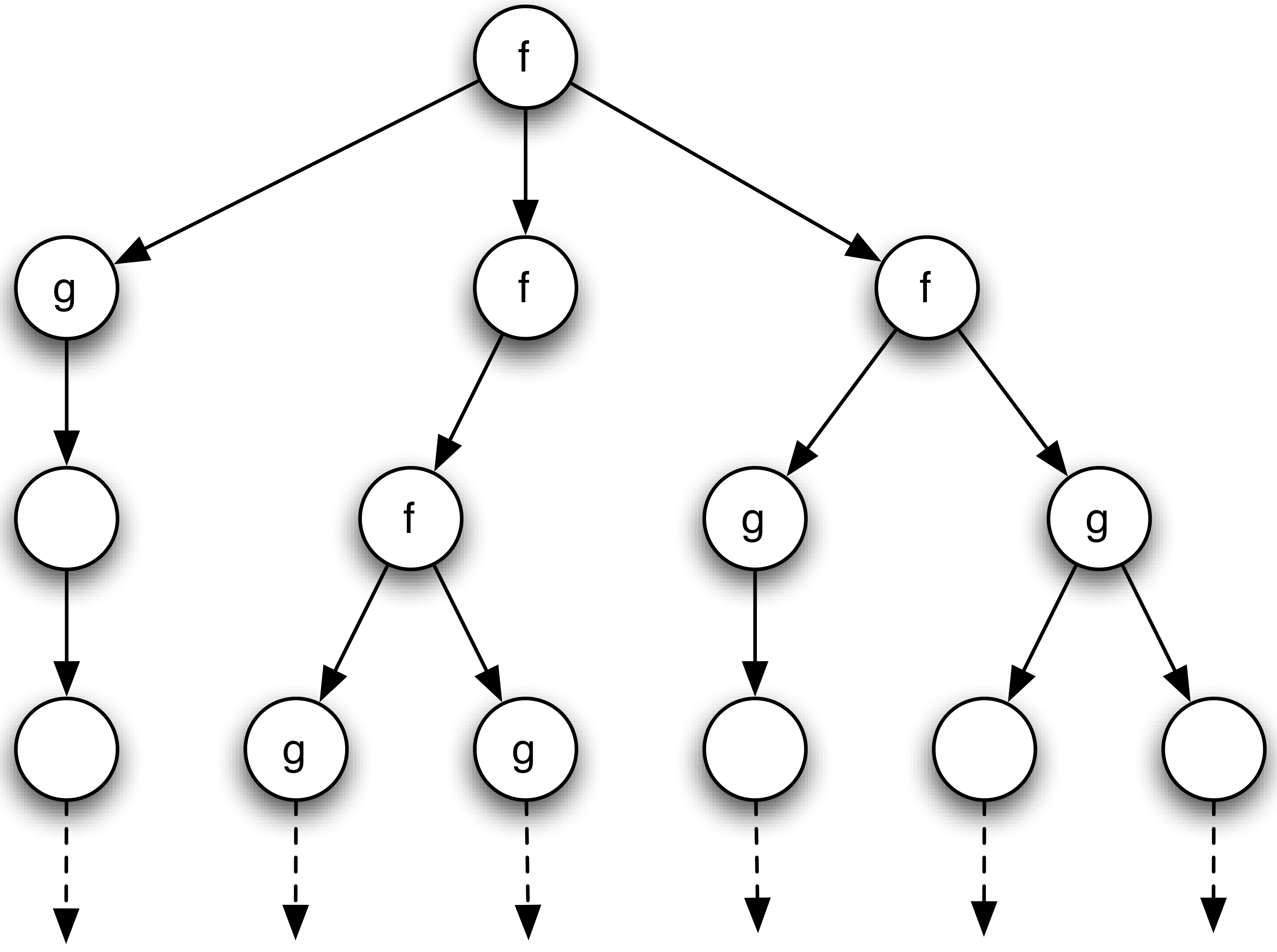
CTL Semantics

AF f



CTL Semantics

$$f \text{ AU } g$$



Minimal CTL Operators

- All CTL formulas can be expressed with ⊤, ¬, ∨, X, EX, EG, and EU

$\bot$	$\equiv$	$\neg \top$
$f \wedge g$	$\equiv$	$\neg (\neg f \vee \neg g)$
$f \rightarrow g$	$\equiv$	$\neg f \vee g$
AX f	$\equiv$	¬ EX ¬f
EF f	$\equiv$	⊤ EU f
AG f	$\equiv$	¬ EF ¬f
AF f	$\equiv$	¬ EG ¬f
g AR f	$\equiv$	¬ (¬g EU ¬f)
g ER f	$\equiv$	(EG f) ∨ (f EU ($g \wedge f$))
f AU g	$\equiv$	¬ (¬f ER ¬g)

CTL Examples

- Mutual exclusion

$$\mathrm{AG}\ \neg(\mathtt{c}_0 \wedge \mathtt{c}_1)$$

- Lockout freedom

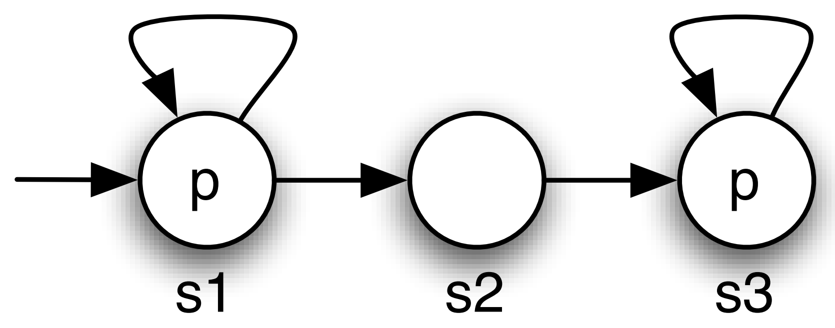
$$\mathrm{AG}\ (\mathtt{w}_0 \rightarrow \mathrm{AF}\ \mathtt{c}_0) \wedge \mathrm{AG}\ (\mathtt{w}_1 \rightarrow \mathrm{AF}\ \mathtt{c}_1)$$

- Reversibility

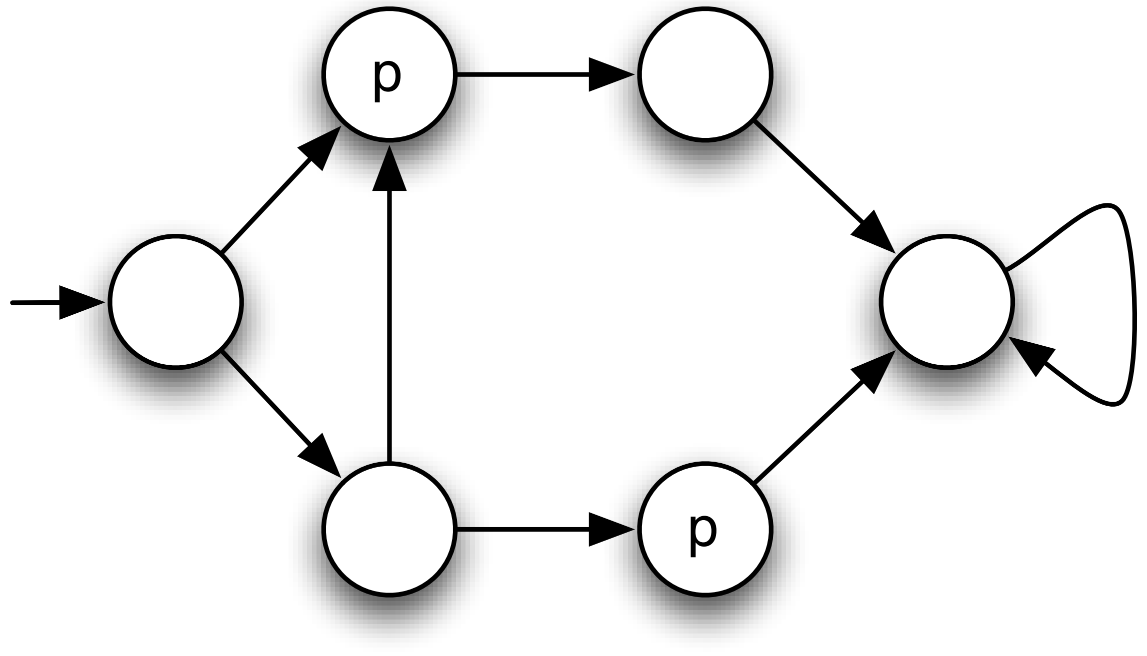
$$\mathrm{AG}\ \mathrm{EF}\ (i_0 \wedge i_1)$$

LTL vs CTL

- Most properties can be expressed both in LTL and CTL, but their expressiveness is incomparable
- For example, AG EF p cannot be expressed in LTL and F G p cannot be expressed in CTL



- In general, LTL formulas are not equivalent to the CTL formulas obtained by preceding each temporal operator with A. For example, AF AX p is not the same as F X p



SMV Input Language

```
CTLSPEC
  AG !(pc[0] = crit & pc[1] = crit)

CTLSPEC
  AG (pc[0] = wait -> AF pc[0] = crit) &
  AG (pc[1] = wait -> AF pc[1] = crit)

CTLSPEC
  AG EF (pc[0] = idle & pc[1] = idle)
```

Safety vs Liveness

- Safety properties
  - Nothing "bad" will happen
  - Counter-examples have a “bad” prefix, one where every possible continuation violates the property
  - Mutual exclusion is a safety property
- Liveness properties
  - Something "good" will happen
  - It is always possible to satisfy them after any finite prefix of events
  - Thus, counter-examples must be complete infinite traces
  - Lockout freedom is a liveness property

Fairness

- *Fairness* assumptions are necessary to verify most liveness properties

- A fairness assumption is a liveness property that forces the system to keep doing something (to progress) under certain conditions

    - *Unconditional fairness*: some action will recurrently occur

$$\mathsf{G}\ \mathsf{F}\ action$$

    - *Strong fairness*: some action that is recurrently enabled will (recurrently) occur

$$\mathsf{G}\ \mathsf{F}\ enabled \rightarrow \mathsf{G}\ \mathsf{F}\ action$$

    - *Weak fairness*: some action that is continuously enabled will (recurrently) occur

$$\mathsf{F}\ \mathsf{G}\ enabled \rightarrow \mathsf{G}\ \mathsf{F}\ action$$

SMV Input Language

```
LTLSPEC
  (G F (pc[0] = wait & sem = 0) -> G F wait0) &
  (G F (pc[1] = wait & sem = 0) -> G F wait1)
  ->
  G (pc[0] = wait -> F pc[0] = crit) &
  G (pc[1] = wait -> F pc[1] = crit)
```

Fairness in TLA

- If a is an *action* predicate (one with primes), then $\mathsf{WF}_t(a)$, and $\mathsf{SF}_t(a)$ are atomic TLA formulas

$$
\begin{aligned}
\mathsf{WF}_t(a) &\equiv \Diamond\Box\,\mathsf{ENABLED}(a) \Rightarrow \Box\Diamond\langle a\rangle_t \\
\mathsf{SF}_t(a) &\equiv \Box\Diamond\,\mathsf{ENABLED}(a) \Rightarrow \Box\Diamond\langle a\rangle_t \\
\Diamond\langle a\rangle_t &\equiv \Diamond(a \wedge t' \neq t) \\
&\equiv \Diamond\neg(\neg a \vee t' = t) \\
&\equiv \Diamond\neg[a]_t \\
&\equiv \neg\Box[a]_t
\end{aligned}
$$

TLA+

```
----------------------------- MODULE Semaphore -----------------------------
EXTENDS Naturals

CONSTANT N
ASSUME N > 0

VARIABLES sem, pc

Proc == 0..(N-1)

Init == …

idle(p) == …
wait(p) == …
crit(p) == …
exit(p) == …

Next == \E p \in Proc : idle(p) \/ wait(p) \/ crit(p) \/ exit(p)

state == <<sem,pc>>

Fairness == WF_state(Next) /\ \A p \in Proc: SF_state(wait(p))

Spec == Init /\ [][Next]_state /\ Fairness
=============================================================================
```

CTL Model Checking

- Given a Kripke structure $M = (S, I, R, L)$ and a CTL formula f, the goal of a model checker is to find the set of states that satisfy f

$$[\![f]\!]_M \equiv \{s \in M \mid M, s \models f\}$$

Explicit vs Symbolic

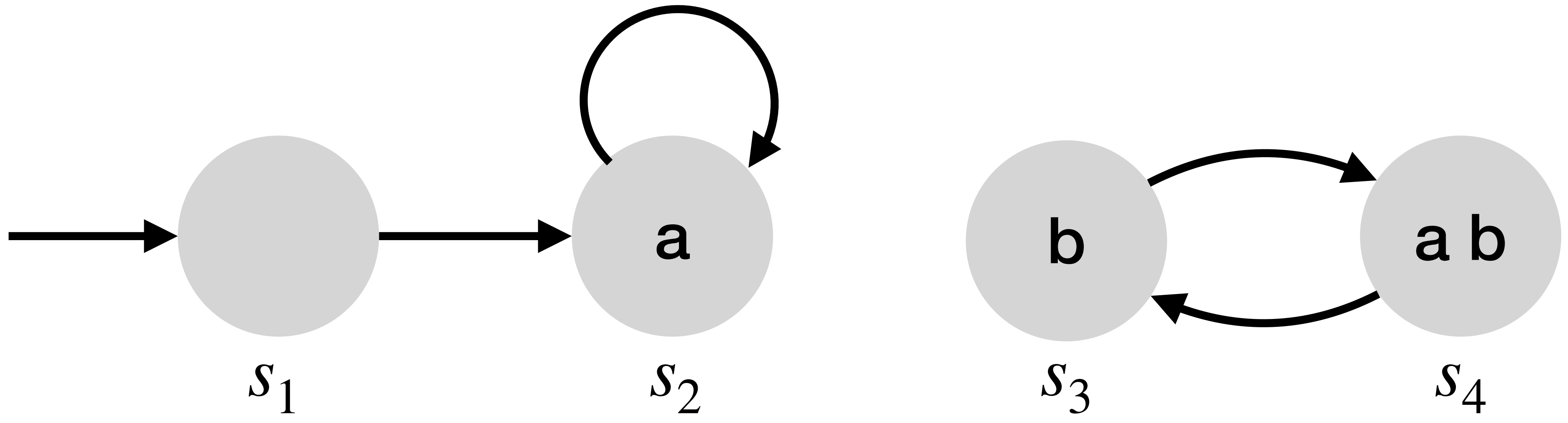
- *Explicit* model checking
  - Sets and transitions are encoded extensionally
  - Semantics of temporal operators is implemented by graph traversals

$$M \vDash f \quad \text{iff} \quad I \subseteq [\![f]\!]_M$$

- *Symbolic* model checking
  - Sets and transitions are encoded intentionally by propositional formulas
  - Semantics of temporal operators is implemented by fixpoint computations

$$M \vDash f \quad \text{iff} \quad I \to [\![f]\!]_M$$

Explicit vs Symbolic



$$I = \{s_1\}$$

$$R = \{(s_1, s_2), (s_2, s_2), (s_3, s_4), (s_4, s_3)\}$$

$$I = \neg a \wedge \neg b$$

$$R = (\neg b \wedge a' \wedge \neg b') \vee (b \wedge b' \wedge a' = \neg a)$$

Explicit CTL Model Checking

$$
\begin{aligned}
[\![p]\!] &= \{s \in S \mid p \in L(s)\} \\
[\![\top]\!] &= S \\
[\![\neg f]\!] &= S - [\![f]\!] \\
[\![f \vee g]\!] &= [\![f]\!] \cup [\![g]\!] \\
[\![\mathsf{EX}\ f]\!] &= \{s \in S \mid \exists s' \in [\![f]\!] \cdot (s, s') \in R\} = \bigcup_{s \in [\![f]\!]} R^{-1}(s)
\end{aligned}
$$

Explicit CTL Model Checking

$[\![f \ \mathsf{EU}\ g]\!] =$

$\quad U \leftarrow [\![g]\!]$

$\quad O \leftarrow A$

$\quad$ **while** $O \neq \emptyset$

$\quad\quad$ **choose** $s \in O$

$\quad\quad O \leftarrow O - \{s\}$

$\quad\quad$ **for** $s' \in R^{-1}(s)$

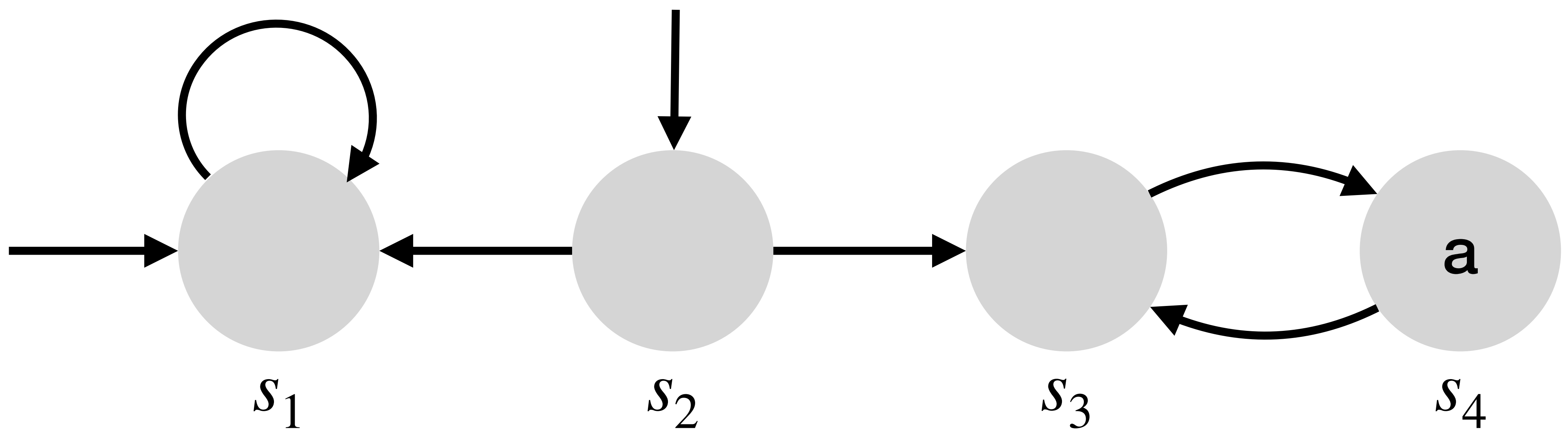
$\quad\quad\quad$ **if** $s' \notin U \wedge s' \in [\![f]\!]$

$\quad\quad\quad\quad U \leftarrow U \cup \{s'\}$

$\quad\quad\quad\quad O \leftarrow O \cup \{s'\}$

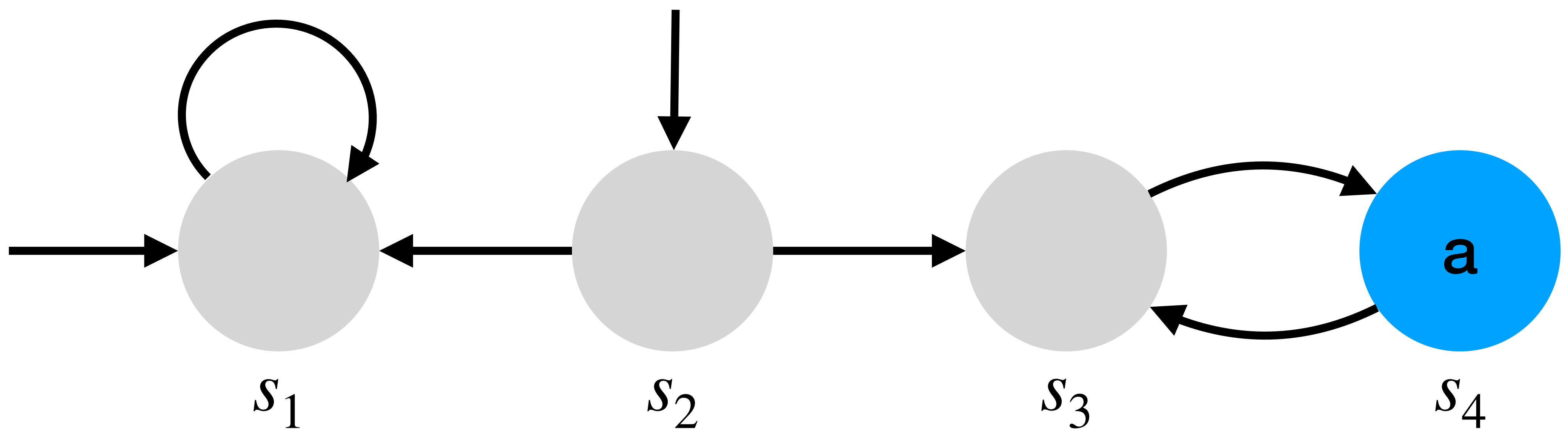
$\quad$ **return** U

Explicit CTL Model Checking



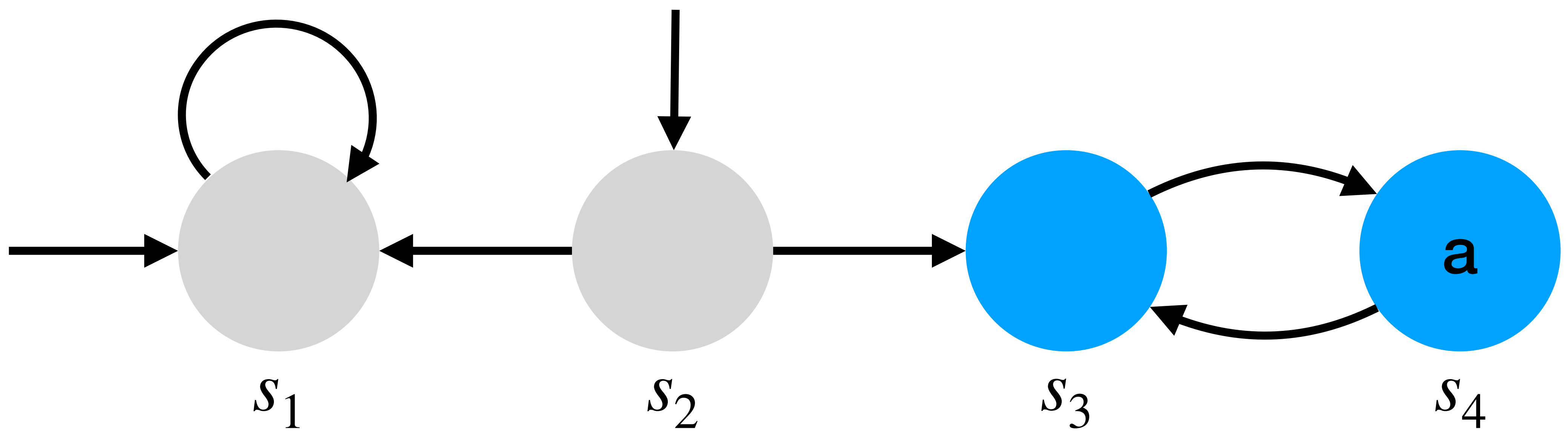
$$[\![\mathsf{AG}\ \neg a]\!] = [\![\neg\ (\top\ \mathsf{EU}\ a)]\!] = S - [\![\top\ \mathsf{EU}\ a]\!] =$$

Explicit CTL Model Checking



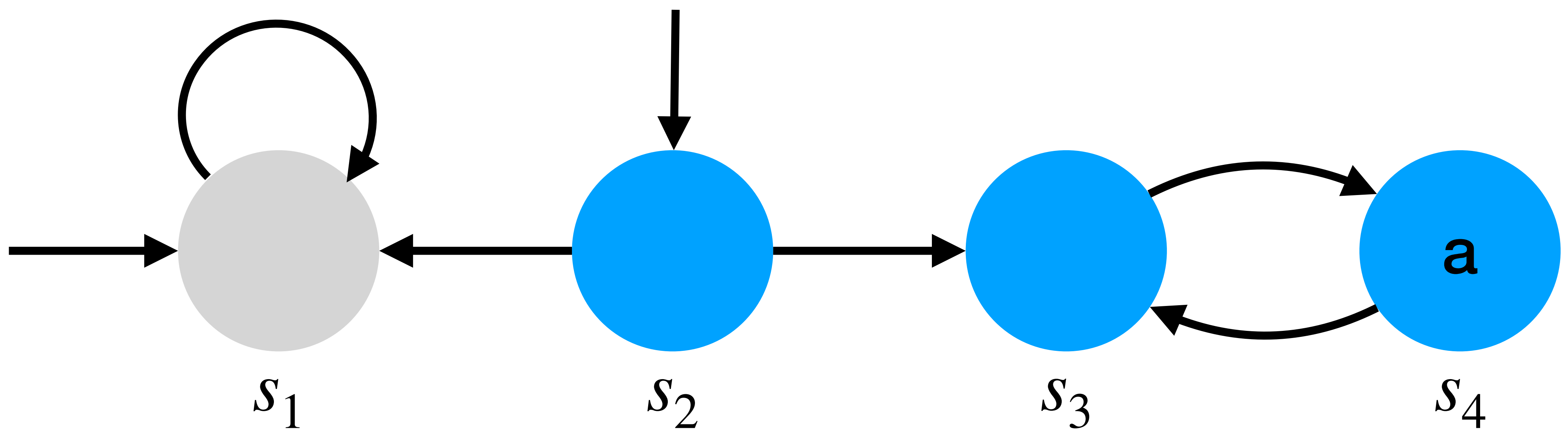
$$[\![\mathsf{AG}\ \neg a]\!] = [\![\neg\ (\top\ \mathsf{EU}\ a)]\!] = S - [\![\top\ \mathsf{EU}\ a]\!] =$$

Explicit CTL Model Checking



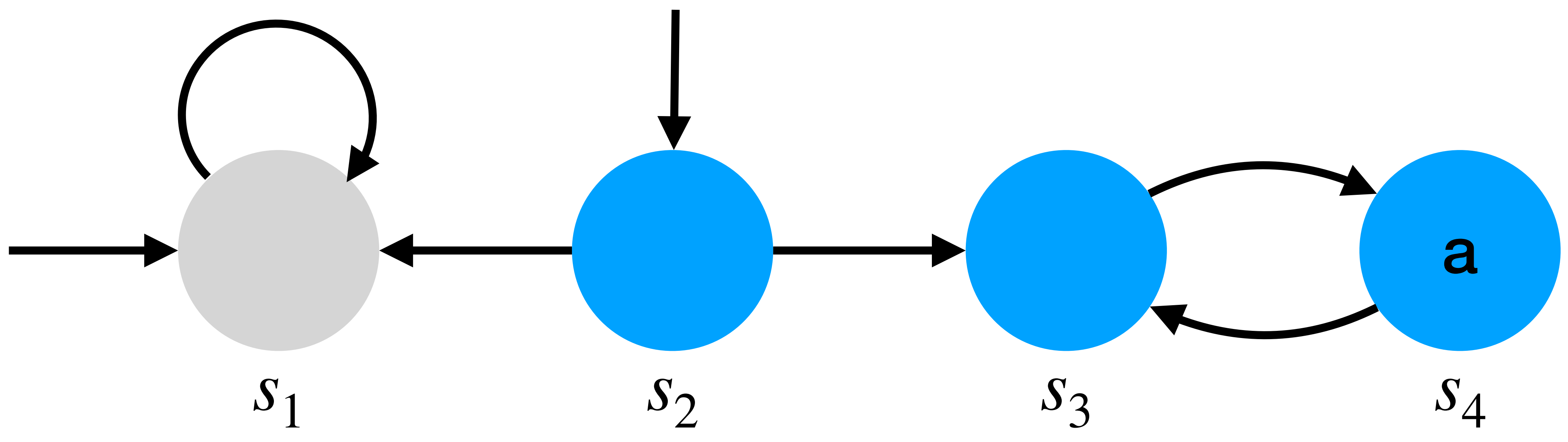
$$[\![\mathsf{AG}\ \neg a]\!] = [\![\neg\ (\top\ \mathsf{EU}\ a)]\!] = S - [\![\top\ \mathsf{EU}\ a]\!] =$$

Explicit CTL Model Checking



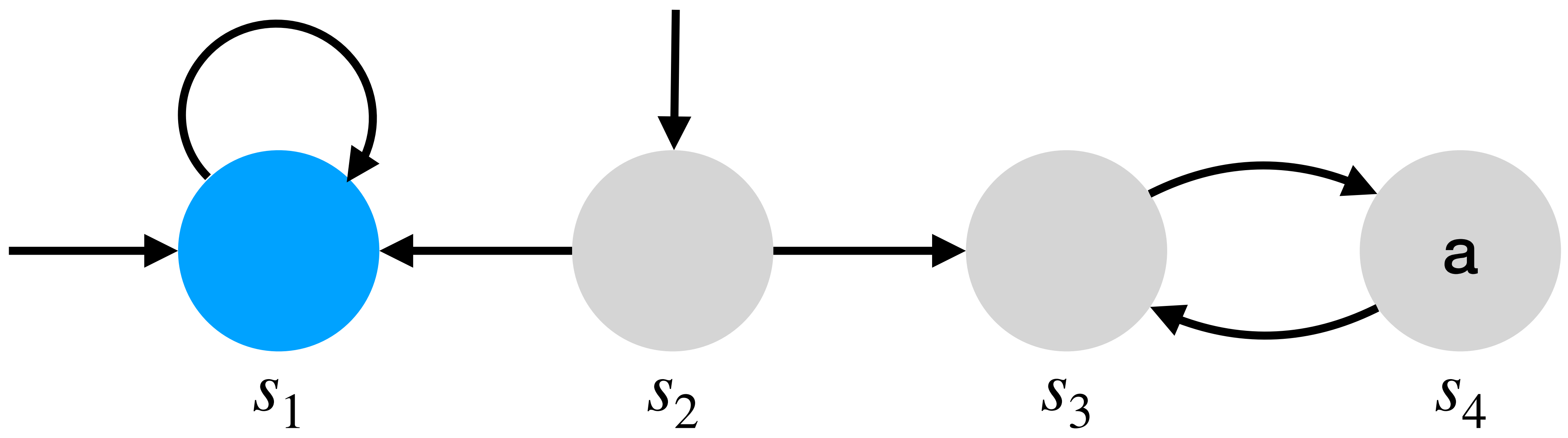
$$[\![\mathsf{AG}\ \neg a]\!] = [\![\neg\ (\top\ \mathsf{EU}\ a)]\!] = S - [\![\top\ \mathsf{EU}\ a]\!] =$$

Explicit CTL Model Checking



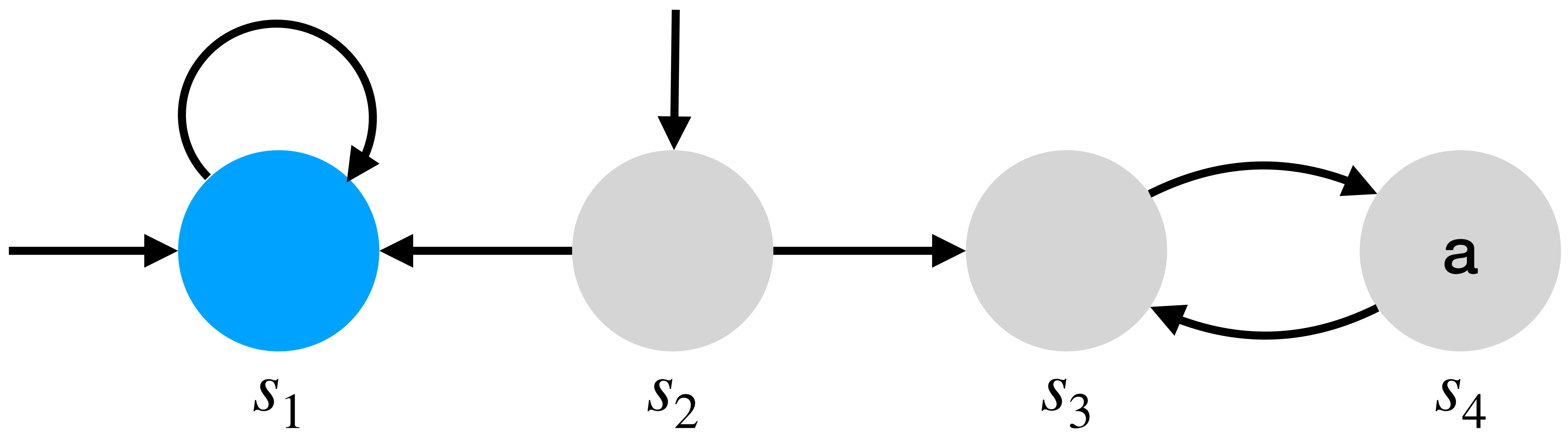
$$[\![\mathsf{AG}\ \neg a]\!] = [\![\neg\ (\top\ \mathsf{EU}\ a)]\!] = S - [\![\top\ \mathsf{EU}\ a]\!] = S - \{s_2, s_3, s_4\} =$$

Explicit CTL Model Checking



$$[\![\text{AG } \neg a]\!] = [\![\neg\ (\top \text{ EU } a)]\!] = S - [\![\top \text{ EU } a]\!] = S - \{s_2, s_3, s_4\} = \{s_1\}$$

Explicit CTL Model Checking



$$[\![\mathsf{AG}\ \neg a]\!] = [\![\neg\ (\top\ \mathsf{EU}\ a)]\!] = S - [\![\top\ \mathsf{EU}\ a]\!] = S - \{s_2, s_3, s_4\} = \{s_1\}$$

$$I \nsubseteq [\![\mathsf{AG}\ \neg a]\!] \quad \Rightarrow \quad M \nvDash \mathsf{AG}\ \neg a$$

Explicit CTL Model Checking

- To determine $[\![\text{EG } f]\!]$ it suffices to restrict M to the states that satisfy f

$$M_f = ([\![f]\!], I \cap [\![f]\!], R \cap [\![f]\!] \times [\![f]\!], L \cap [\![f]\!] \times A)$$

- $M, s \models \text{EG } f$ iff $s \in [\![f]\!]$ and there exists a path in M_f from s to some node in a *nontrivial strongly connected component* of M_f

- A *strongly connected component* (SCC) is a maximal subgraph where every node is reachable from every other

- A SCC is *nontrivial* if it has at least one path (more than one node or one node with a self loop)

- The nontrivial SCCs of M_f can be computed efficiently with Tarjan's algorithm

$$\mathbf{scc}(M_f) \subseteq 2^{[\![f]\!]}$$

Explicit CTL Model Checking

$[\![\mathsf{EG}\ f]\!] =$

$G \leftarrow \bigcup \mathbf{scc}(M_f)$

$O \leftarrow U$

while $O \neq \emptyset$

  choose $s \in O$

  $O \leftarrow O - \{s\}$

  for $s' \in R^{-1}(s)$

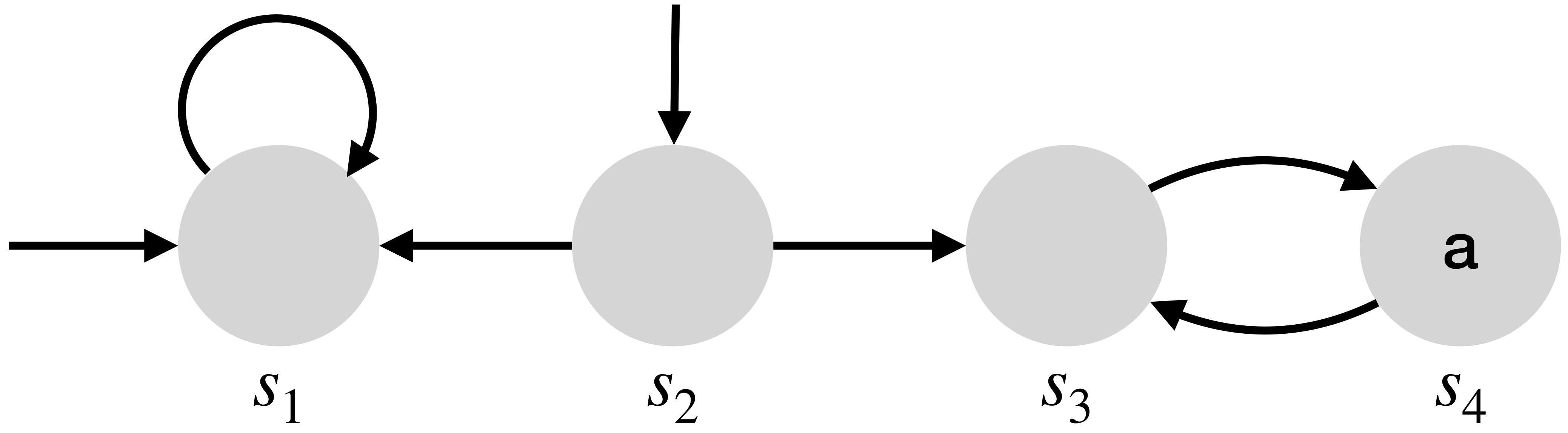
    if $s' \notin G$

      $G \leftarrow G \cup \{s'\}$

      $O \leftarrow O \cup \{s'\}$

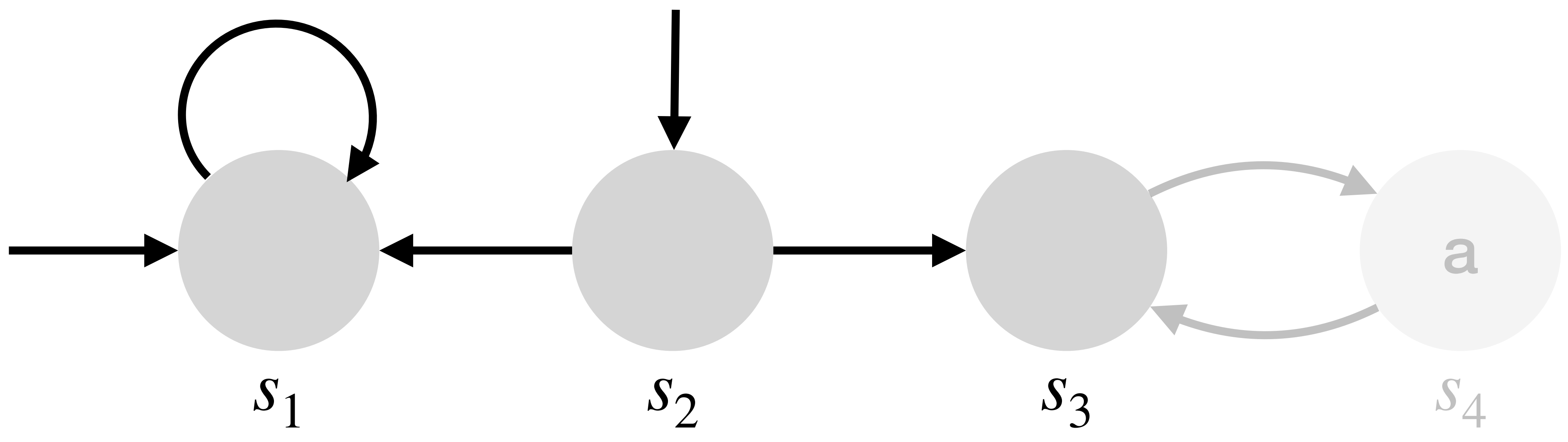
return G

Explicit CTL Model Checking



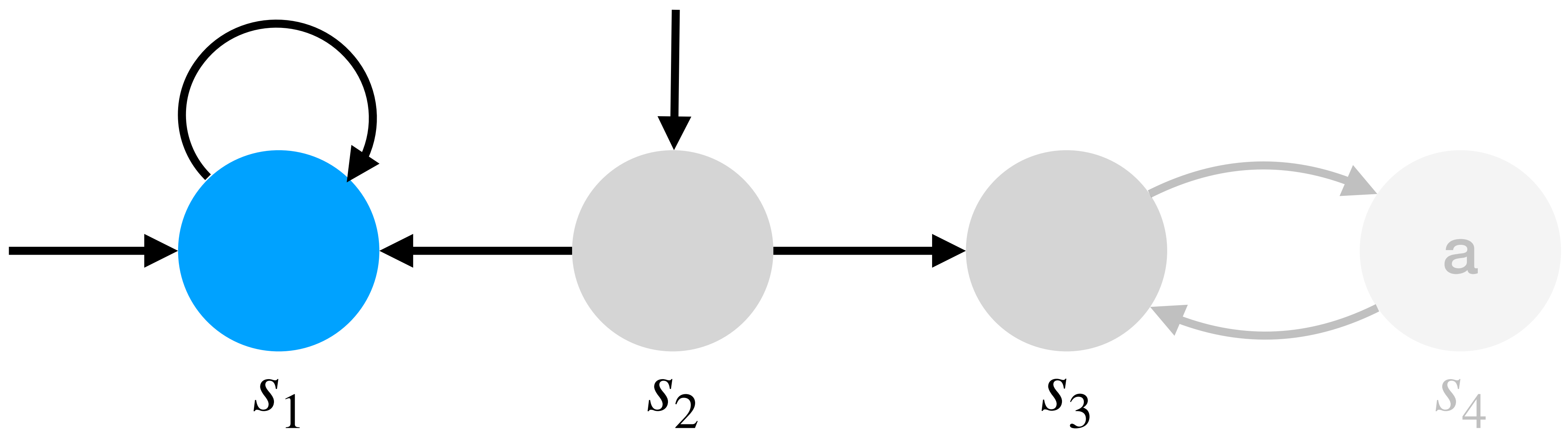
$$[\![\mathsf{AF}\ a]\!] = [\![\neg\ \mathsf{EG}\ \neg a]\!] = S - [\![\mathsf{EG}\ \neg a]\!] =$$

Explicit CTL Model Checking



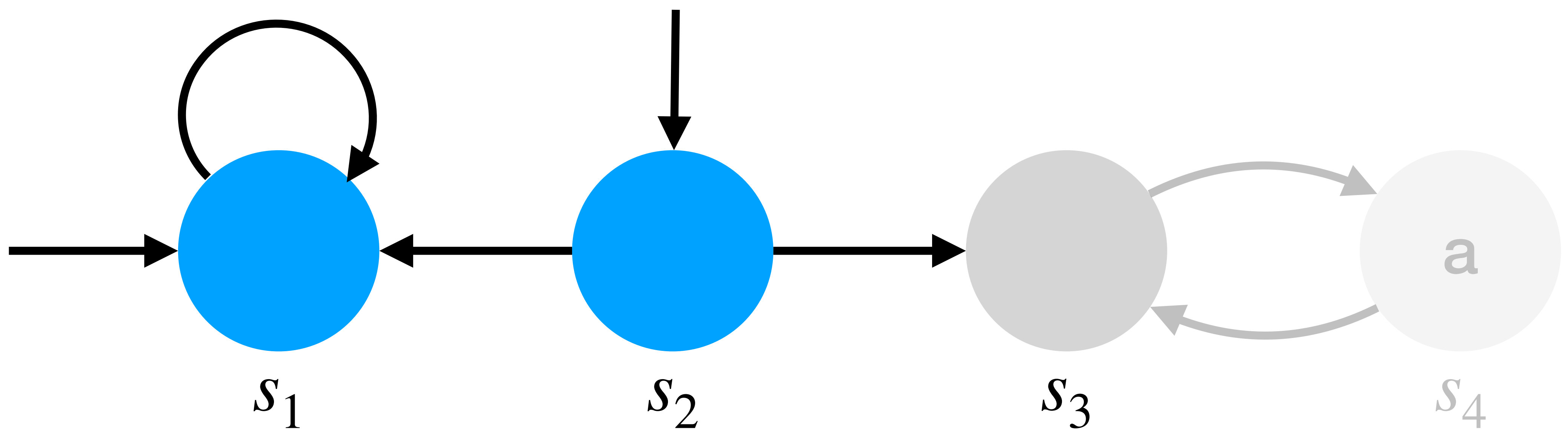
$[\![\text{AF } a]\!] = [\![\neg \text{ EG } \neg a]\!] = S - [\![\text{EG } \neg a]\!] =$

Explicit CTL Model Checking



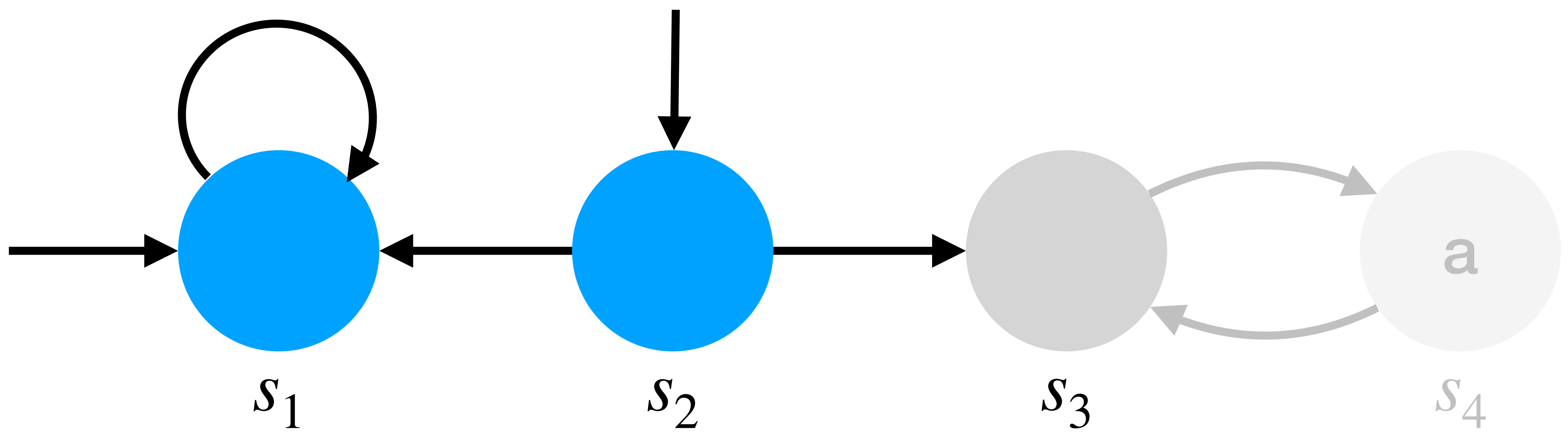
$[\![\mathsf{AF}\ a]\!] = [\![\neg\ \mathsf{EG}\ \neg a]\!] = S - [\![\mathsf{EG}\ \neg a]\!] =$

Explicit CTL Model Checking



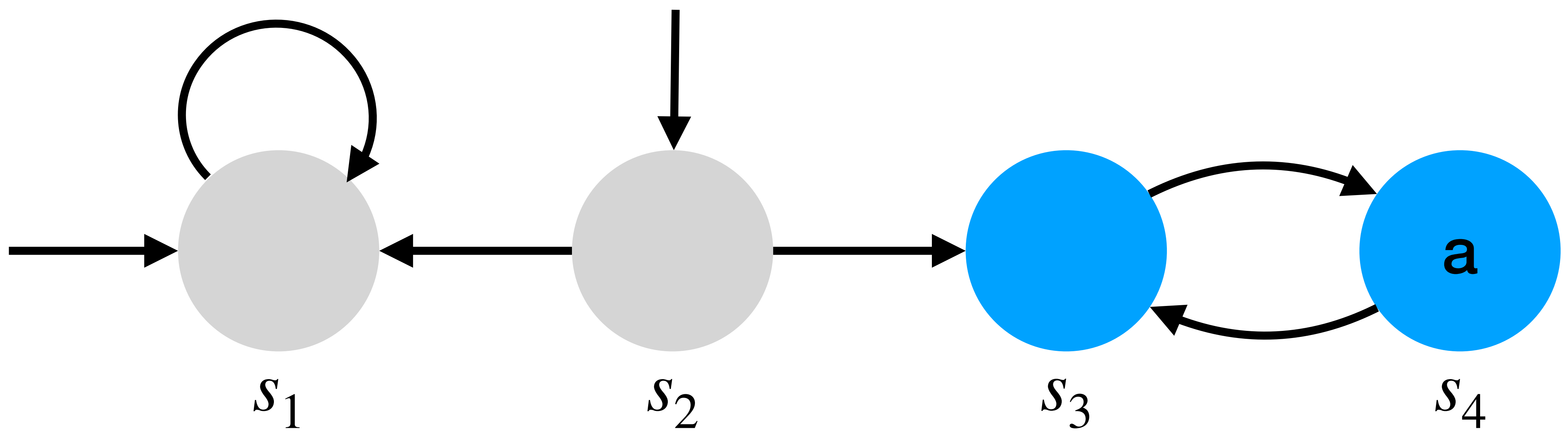
$$[\![\mathsf{AF}\ a]\!] = [\![\neg\ \mathsf{EG}\ \neg a]\!] = S - [\![\mathsf{EG}\ \neg a]\!] =$$

Explicit CTL Model Checking



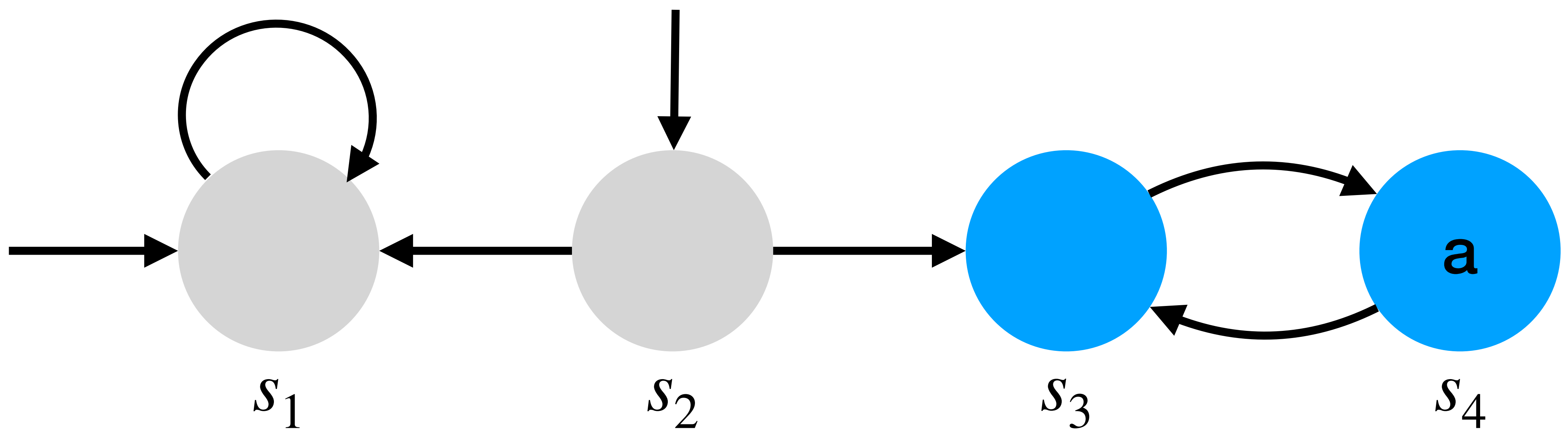
$$[\![\mathsf{AF}\ a]\!] = [\![\neg\ \mathsf{EG}\ \neg a]\!] = S - [\![\mathsf{EG}\ \neg a]\!] = S - \{s_1, s_2\} =$$

Explicit CTL Model Checking



$$[\![\mathsf{AF}\ a]\!] = [\![\neg\ \mathsf{EG}\ \neg a]\!] = S - [\![\mathsf{EG}\ \neg a]\!] = S - \{s_1, s_2\} = \{s_3, s_4\}$$

Explicit CTL Model Checking



$$[\![\mathsf{AF}\ a]\!] = [\![\neg\ \mathsf{EG}\ \neg a]\!] = S - [\![\mathsf{EG}\ \neg a]\!] = S - \{s_1, s_2\} = \{s_3, s_4\}$$

$$I \nsubseteq [\![\mathsf{AF}\ a]\!] \quad \Rightarrow \quad M \nvDash \mathsf{AF}\ a$$

Explicit CTL Model Checking

- Fairness cannot be expressed in CTL
- Semantics and model checking must be adapted to consider fairness
- $M \vDash [f]_p$ iff $M \vDash f$ and p is recurrently true in M (unconditional fairness)
- To model check $M \vDash [\mathsf{EG}\ f]_p$ it suffices to compute the reachability from *fair* SCCs, those where at least one state satisfies p
- Since a path is fair iff any of its suffixes is fair and since $[\ \mathsf{EG}\ \top\ ]_p$ holds in a state iff there is a fair path starting in that state we have

$$[\mathsf{EX}\ f]_p \equiv \mathsf{EX}\ ([f]_p \wedge [\mathsf{EG}\ \top\ ]_p)$$

$$[f\ \mathsf{EU}\ g]_p \equiv [f]_p\ \mathsf{EU}\ ([g]_p \wedge [\mathsf{EG}\ \top\ ]_p)$$

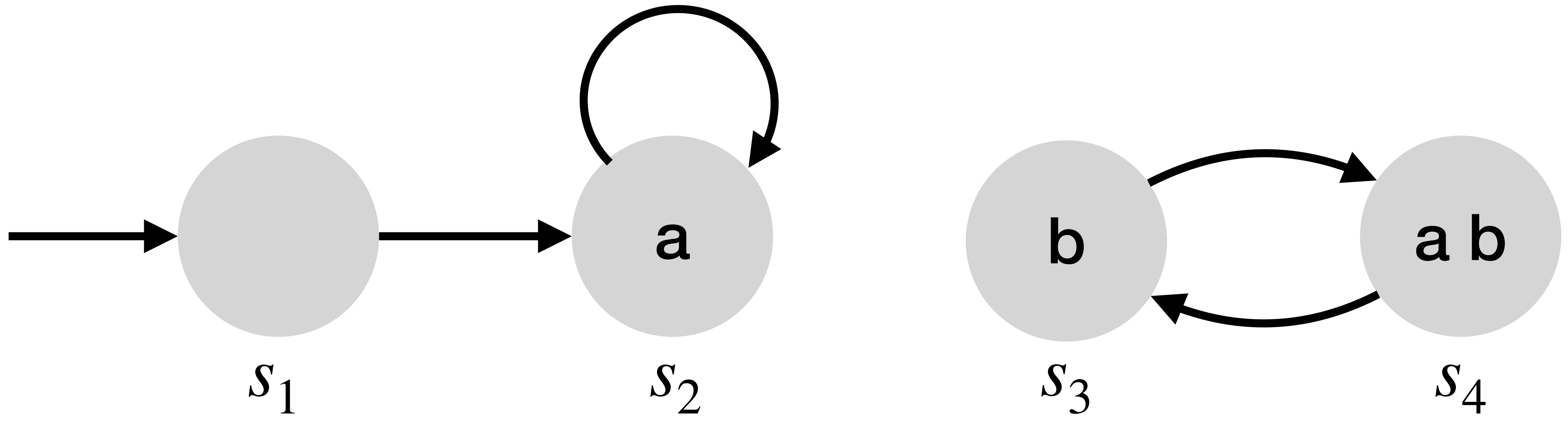
Symbolic CTL Model Checking

$$
\begin{aligned}
[\![p]\!] &= p \\
[\![\top]\!] &= \top \\
[\![\neg f]\!] &= \neg [\![f]\!] \\
[\![f \vee g]\!] &= [\![f]\!] \vee [\![g]\!] \\
[\![\mathsf{EX}\, f]\!] &= \exists \bar{x}' \cdot R \wedge [\![f]\!]'
\end{aligned}
$$

- $[\![f]\!]'$ is the formula obtained from $[\![f]\!]$ by replacing all variables by the respective primed version
- The existential quantifier can be eliminated by expansion

$$\exists x \cdot f \equiv f[x \leftarrow \top] \vee f[x \leftarrow \bot]$$

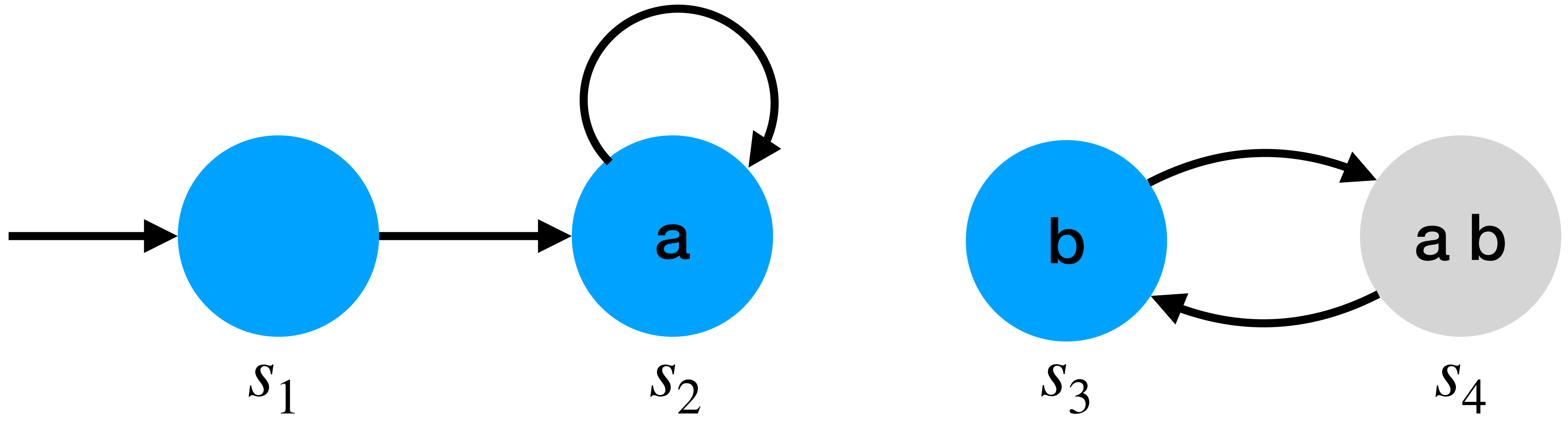
Symbolic CTL Model Checking



$$R = (\neg b \wedge a' \wedge \neg b') \vee (b \wedge b' \wedge a' = \neg a)$$

$$\begin{aligned}
[\![\mathsf{EX}\ a]\!] &= \exists a', b' \cdot R \wedge [\![a]\!]' \\
&= \exists a', b' \cdot R \wedge a' \\
&= \exists a', b' \cdot (\neg b \wedge a' \wedge \neg b') \vee (b \wedge \neg a \wedge b' \wedge a') \\
&= \exists a' \cdot (\neg b \wedge a' \wedge \neg \top) \vee (b \wedge \neg a \wedge \top \wedge a') \vee (\neg b \wedge a' \wedge \neg \bot) \vee (b \wedge \neg a \wedge \bot \wedge a') \\
&= \exists a' \cdot (b \wedge \neg a \wedge a') \vee (\neg b \wedge a') \\
&= (b \wedge \neg a \wedge \top) \vee (\neg b \wedge \top) \vee (b \wedge \neg a \wedge \bot) \vee (\neg b \wedge \bot) \\
&= (b \wedge \neg a) \vee \neg b
\end{aligned}$$

Symbolic CTL Model Checking



$$R = (\neg b \wedge a' \wedge \neg b') \vee (b \wedge b' \wedge a' = \neg a)$$

$$
\begin{aligned}
[\![\mathsf{EX}\ a]\!] &= \exists a', b' \cdot R \wedge [\![a]\!]' \\
&= \exists a', b' \cdot R \wedge a' \\
&= \exists a', b' \cdot (\neg b \wedge a' \wedge \neg b') \vee (b \wedge \neg a \wedge b' \wedge a') \\
&= \exists a' \cdot (\neg b \wedge a' \wedge \neg \top) \vee (b \wedge \neg a \wedge \top \wedge a') \vee (\neg b \wedge a' \wedge \neg \bot) \vee (b \wedge \neg a \wedge \bot \wedge a') \\
&= \exists a' \cdot (b \wedge \neg a \wedge a') \vee (\neg b \wedge a') \\
&= (b \wedge \neg a \wedge \top) \vee (\neg b \wedge \top) \vee (b \wedge \neg a \wedge \bot) \vee (\neg b \wedge \bot) \\
&= (b \wedge \neg a) \vee \neg b
\end{aligned}
$$

Symbolic CTL Model Checking

$$\mathsf{EG}\ f \equiv f \wedge \mathsf{EX}\ (\mathsf{EG}\ f)$$

$[\![\mathsf{EG}\ f]\!] =$

$G \leftarrow \top$

repeat

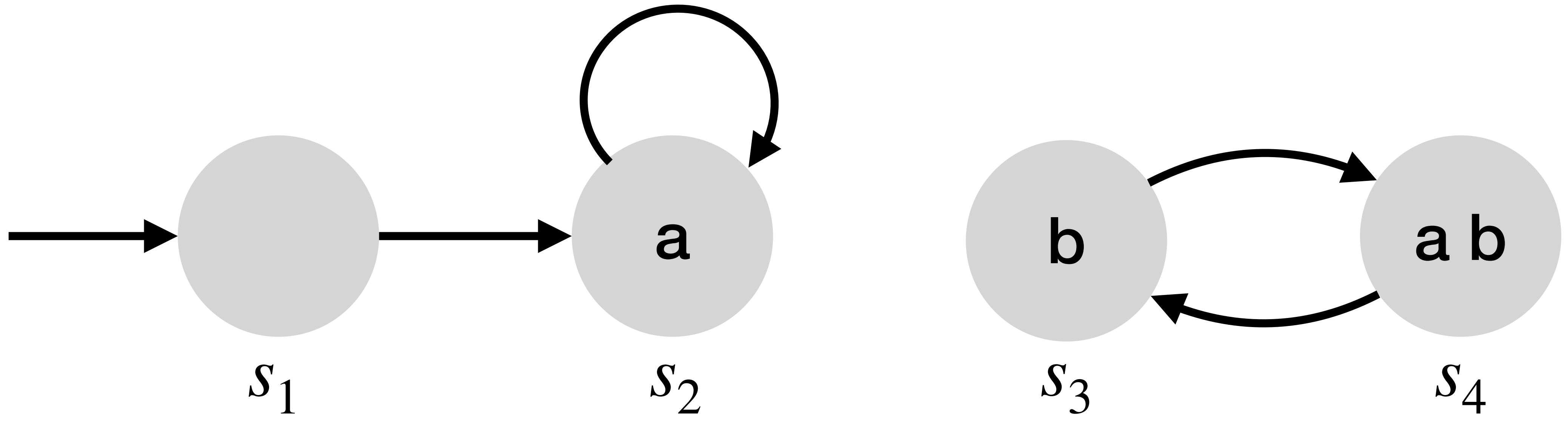
$G' \leftarrow G$

$G \leftarrow [\![f]\!] \wedge [\![\mathsf{EX}\ G]\!]$

until $G \equiv G'$

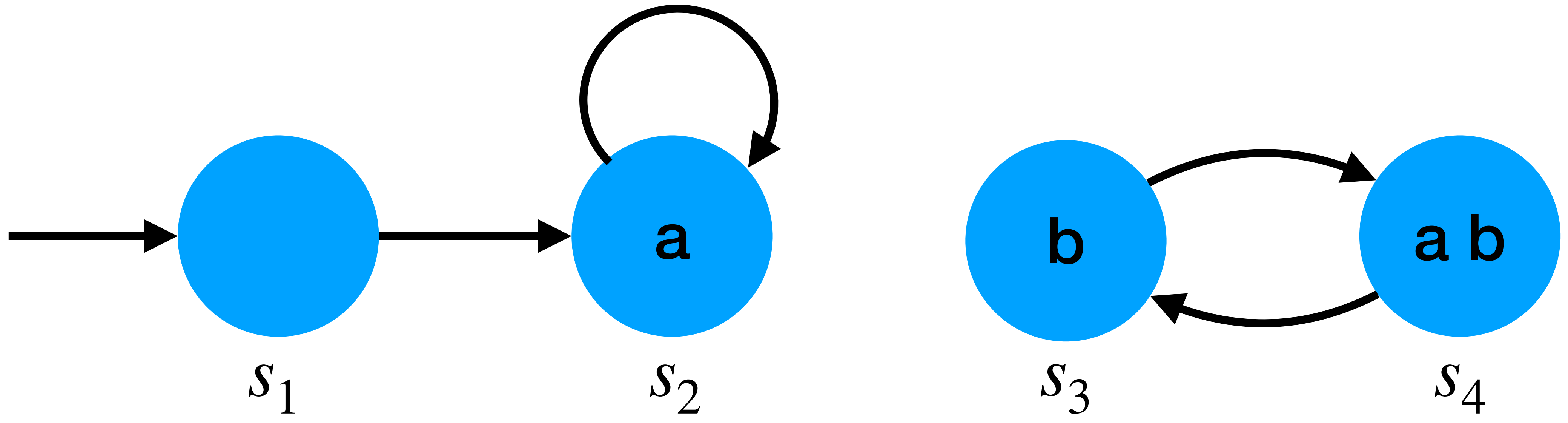
return G

Symbolic CTL Model Checking



$$[\![\mathsf{EG}\ a]\!] =$$

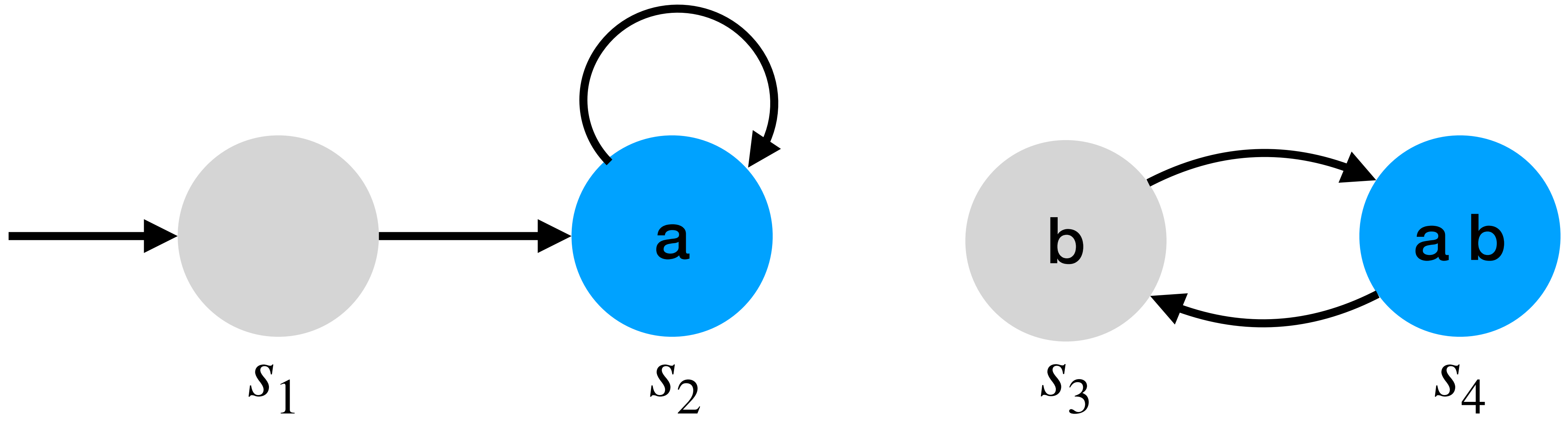
Symbolic CTL Model Checking



$$[\![\mathsf{EG}\ a]\!] =$$

$$G_0 = \top$$

Symbolic CTL Model Checking

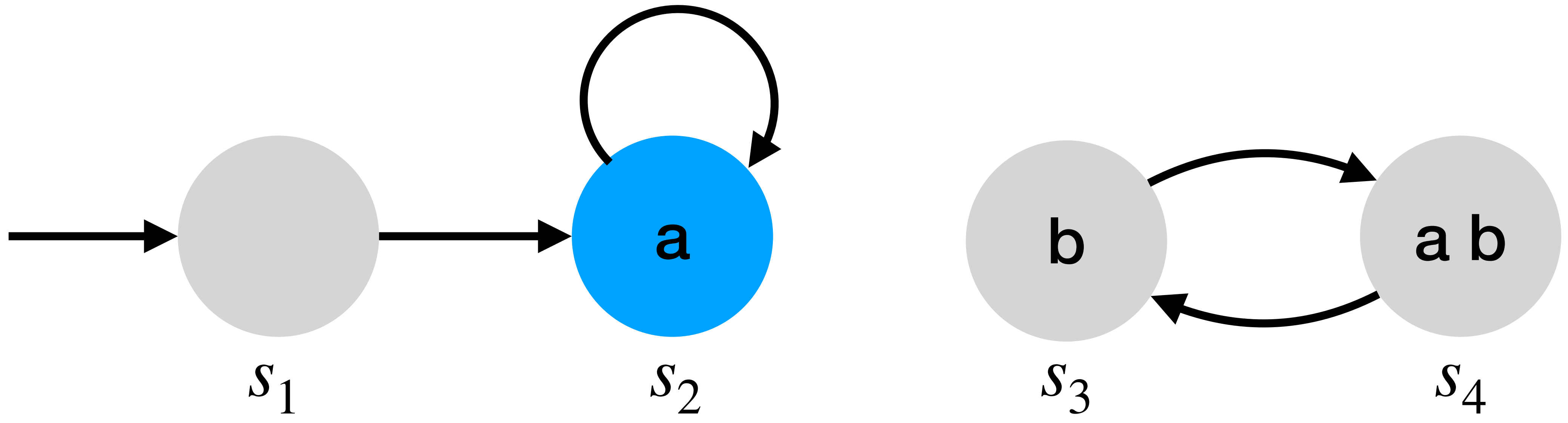


$$[\![\mathsf{EG}\ a]\!] =$$

$$G_0 = \top$$

$$G_1 = [\![a]\!] \land [\![\mathsf{EX}\ G_0]\!] = a$$

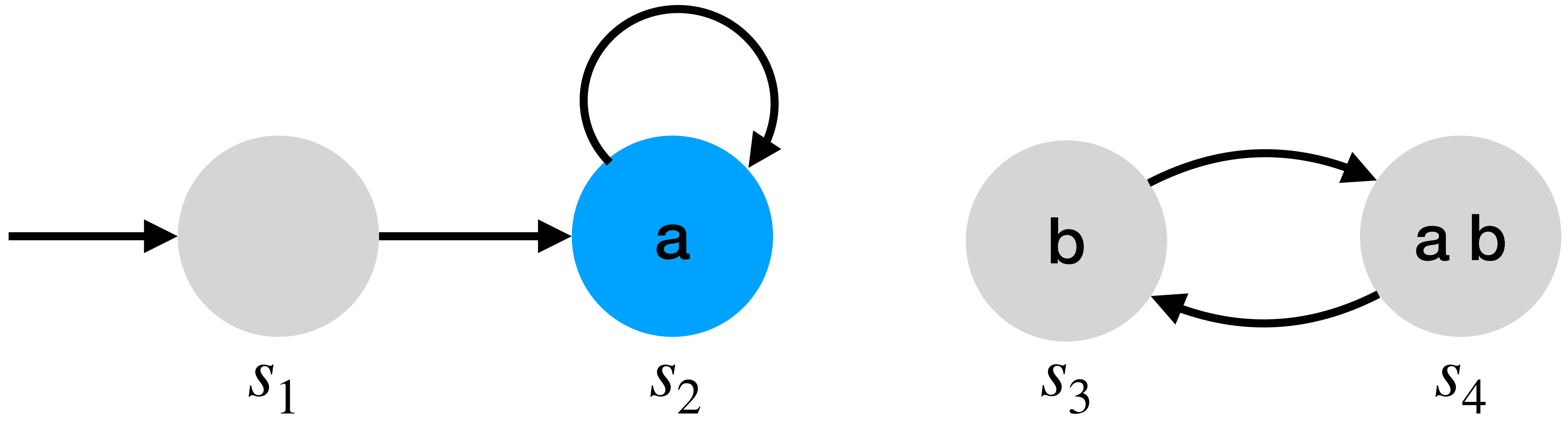
Symbolic CTL Model Checking



$[\![\text{EG } a]\!] =$

$$G_0 = \top$$
$$G_1 = [\![a]\!] \wedge [\![\text{EX } G_0]\!] = a$$
$$G_2 = [\![a]\!] \wedge [\![\text{EX } G_1]\!] = a \wedge ((b \wedge \neg a) \vee \neg b) = a \wedge \neg b$$

Symbolic CTL Model Checking



$$[\![\mathsf{EG}\ a]\!] = a \wedge \neg b$$

$$G_0 = \top$$
$$G_1 = [\![a]\!] \wedge [\![\mathsf{EX}\ G_0]\!] = a$$
$$G_2 = [\![a]\!] \wedge [\![\mathsf{EX}\ G_1]\!] = a \wedge ((b \wedge \neg a) \vee \neg b) = a \wedge \neg b$$
$$G_3 = [\![a]\!] \wedge [\![\mathsf{EX}\ G_2]\!] = a \wedge \neg b$$

Symbolic CTL Model Checking

$$f \ \mathsf{EU}\ g \equiv g \lor (f \land \mathsf{EX}\ (f\ \mathsf{EU}\ g))$$

$$
\begin{array}{l}
[\![f\ \mathsf{EU}\ g]\!] = \\
\quad U \leftarrow \bot \\
\quad \textbf{repeat} \\
\quad\quad U' \leftarrow U \\
\quad\quad U \leftarrow [\![g]\!] \lor ([\![f]\!] \land [\![\mathsf{EX}\ U]\!]) \\
\quad \textbf{until}\ U \equiv U' \\
\quad \textbf{return}\ U
\end{array}
$$

Symbolic CTL Model Checking

$$[\mathsf{EG}\ f]_p \equiv [f]_p \land \mathsf{EX}\ ([f]_p\ \mathsf{EU}\ (p \land [\mathsf{EG}\ f]_p))$$

$$[\mathsf{EX}\ f]_p \equiv \mathsf{EX}\ ([f]_p \land [\mathsf{EG}\ \top]_p)$$

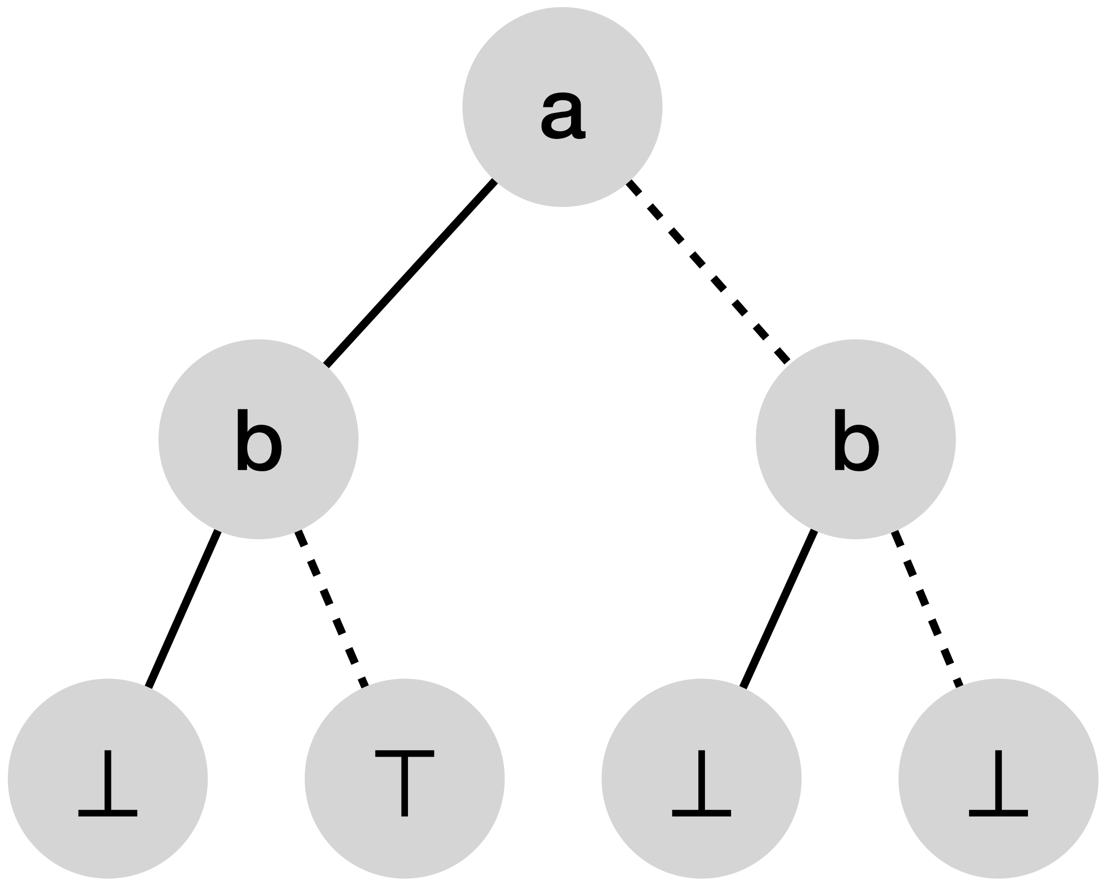
$$[f\ \mathsf{EU}\ g]_p \equiv [f]_p\ \mathsf{EU}\ ([g]_p \land [\mathsf{EG}\ \top]_p)$$

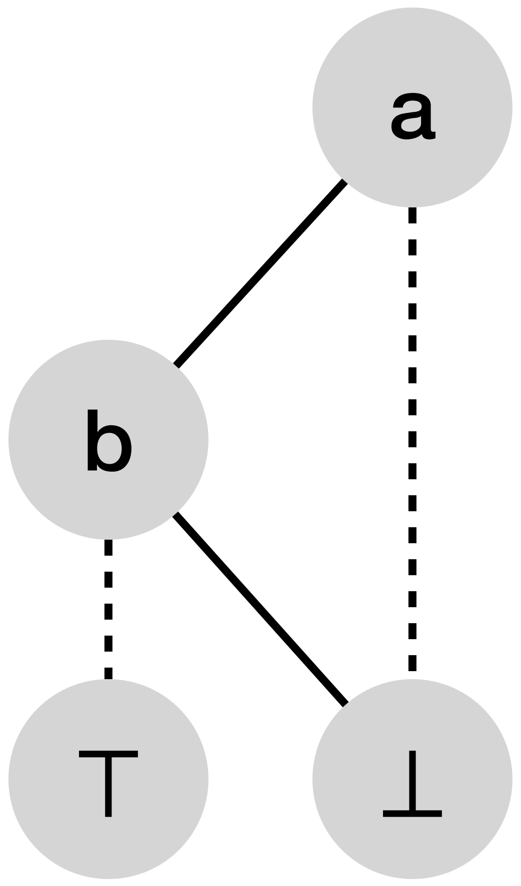
Symbolic CTL Model Checking

- Requires procedures to check the validity and equivalence of propositional formulas

- Can be implemented efficiently with *Ordered Binary Decision Diagrams*

$$a \wedge \neg b$$





Explicit LTL Model Checking

- Given a Kripke structure $M = (S, I, R, L)$ and considering S as an alphabet, the language of M, denoted $\mathscr{L}(M)$ is the set of all paths starting in an initial state

$$\mathscr{L}(M) = \{\pi \mid \pi \in M \wedge \pi_0 \in I\}$$

- The language of an LTL formula f, denoted $\mathscr{L}(f)$ is the set of all possible paths that satisfy f

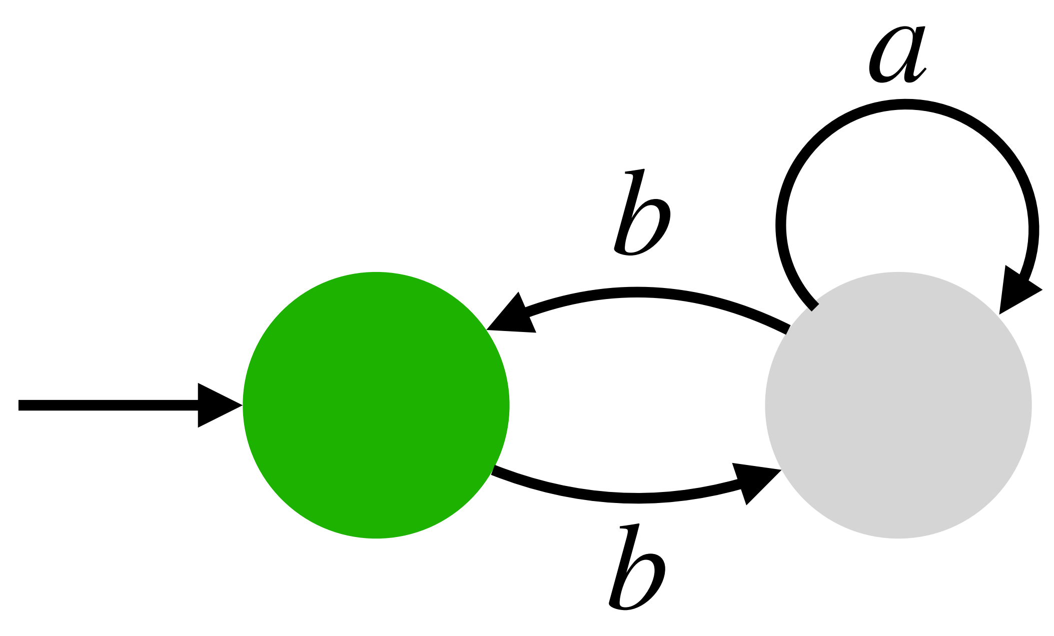
$$\mathscr{L}(f) = \{\pi \mid \pi \vDash f\}$$

- A formula is valid iff the language of the model is a subset of the language of the formula

$$M \vDash f \quad \text{iff} \quad \mathscr{L}(M) \subseteq \mathscr{L}(f) \quad \text{iff} \quad \mathscr{L}(M) \cap \mathscr{L}(\neg f) = \emptyset$$

Büchi Automata

- The model and formula languages can captured by a *non-deterministic Büchi automaton* (S, Σ, R, I, F) where
  - S is a set of states
  - Σ is an alphabet
  - $R \subseteq S \times \Sigma \times S$ is a transition relation
  - $I \subseteq S$ is a set of initial states
  - $F \subseteq S$ is a set of accepting (or final) states
- A valid path must visit an accepting state infinitely often
- The language of an automaton is the set of all valid paths

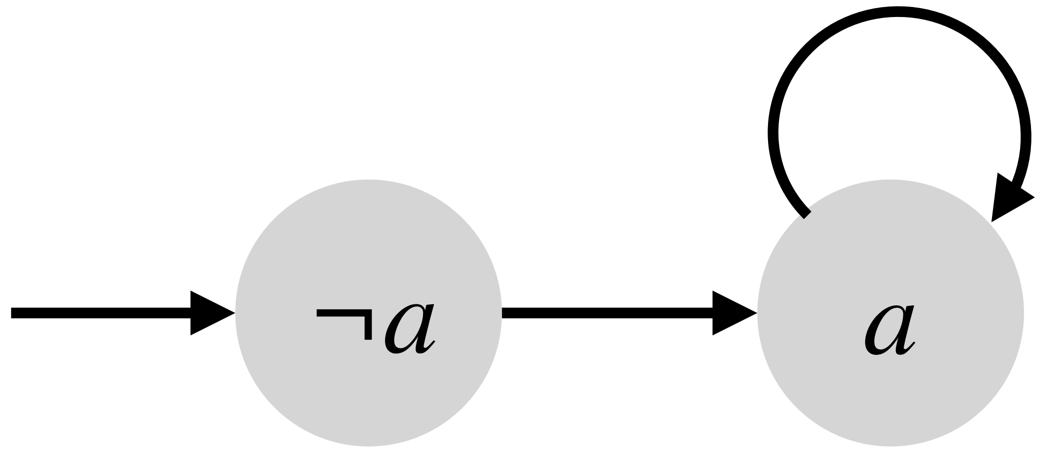


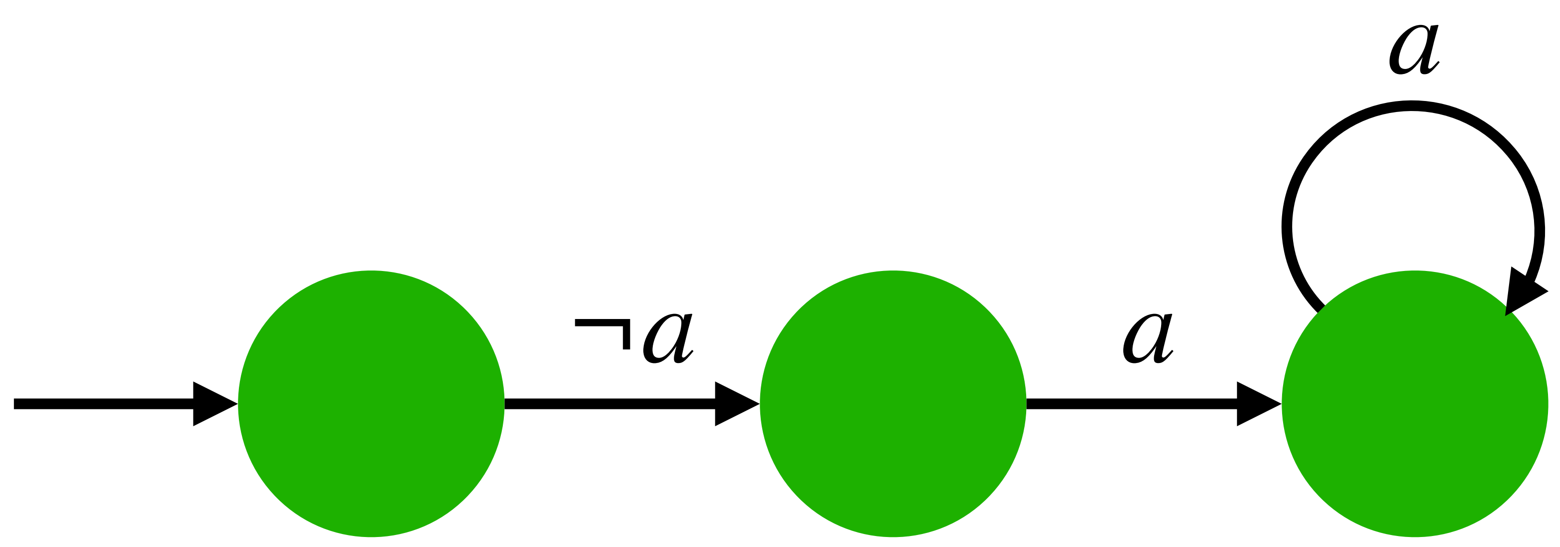
$$\mathscr{L} = \{bbbbb\ldots, babbb\ldots, bbabbb\ldots, baabb\ldots\}$$

From Kripke Structures to Automata

- Given a Kripke structure M it is possible to construct a Büchi automaton $\mathscr{A}_M$ with the same language
  - Using as alphabet conjunctions of atomic propositions, that is $\Sigma = 2^A$
  - Adding a new separate initial state
  - A transition in $\mathscr{A}_M$ is possible iff the transition label matches the next state label in M
  - All states are accepting

From Kripke Structures to Automata



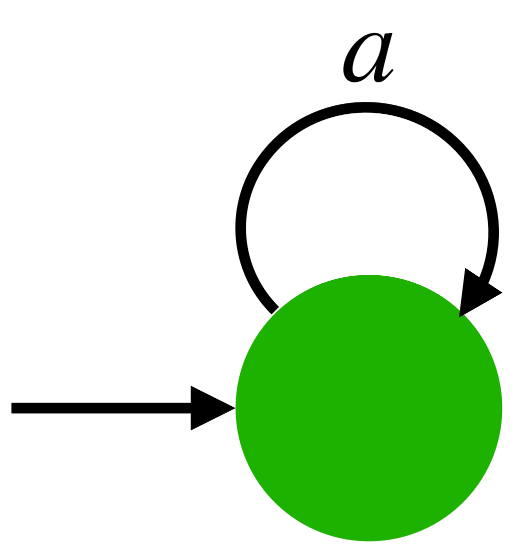


From LTL Formulas to Automata

- Given an LTL formula f it is possible to construct a Büchi automaton $\mathscr{A}_f$ with the same language
  - Again using as alphabet conjunctions of atomic propositions, that is $\Sigma = 2^A$
  - Try it at http://www.lsv.fr/~gastin/ltl2ba/index.php

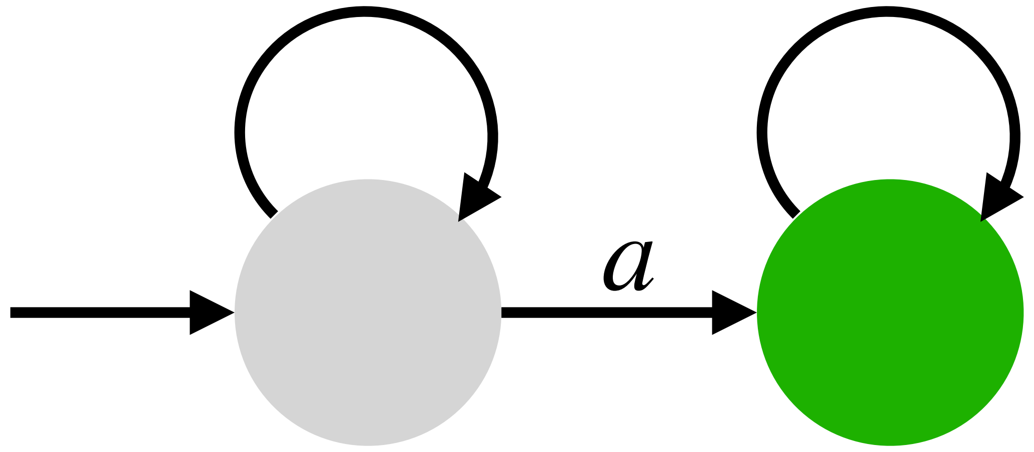
From LTL Formulas to Automata

G a



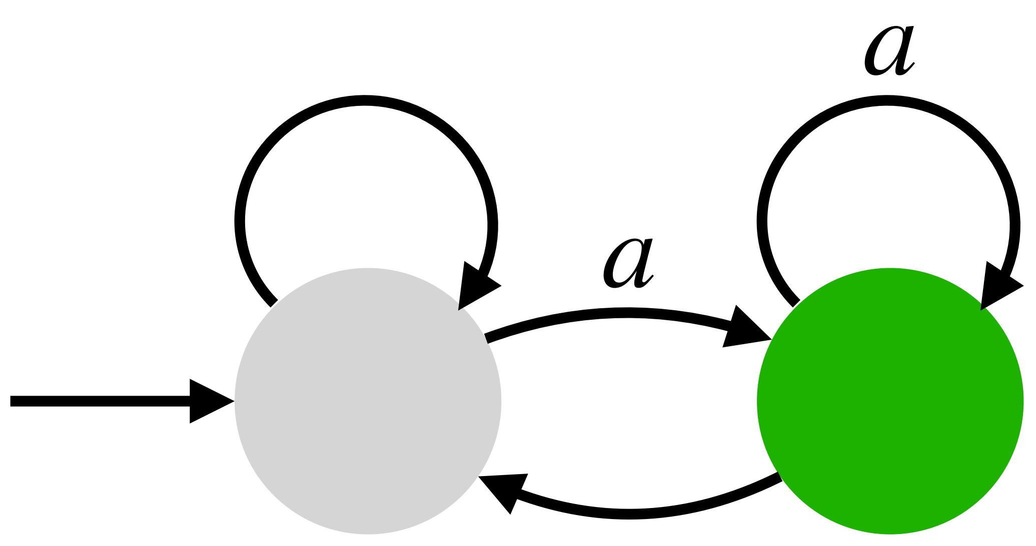
From LTL Formulas to Automata

F a



From LTL Formulas to Automata

G F a



Explicit LTL Model Checking

- Checking the emptiness of language intersection can be reduced to checking the emptiness of the product automaton

$$M \vDash f \quad \text{iff} \quad \mathscr{L}(M) \cap \mathscr{L}(\neg f) = \emptyset \quad \text{iff} \quad \mathscr{L}(\mathscr{A}_M \otimes \mathscr{A}_{\neg f})$$

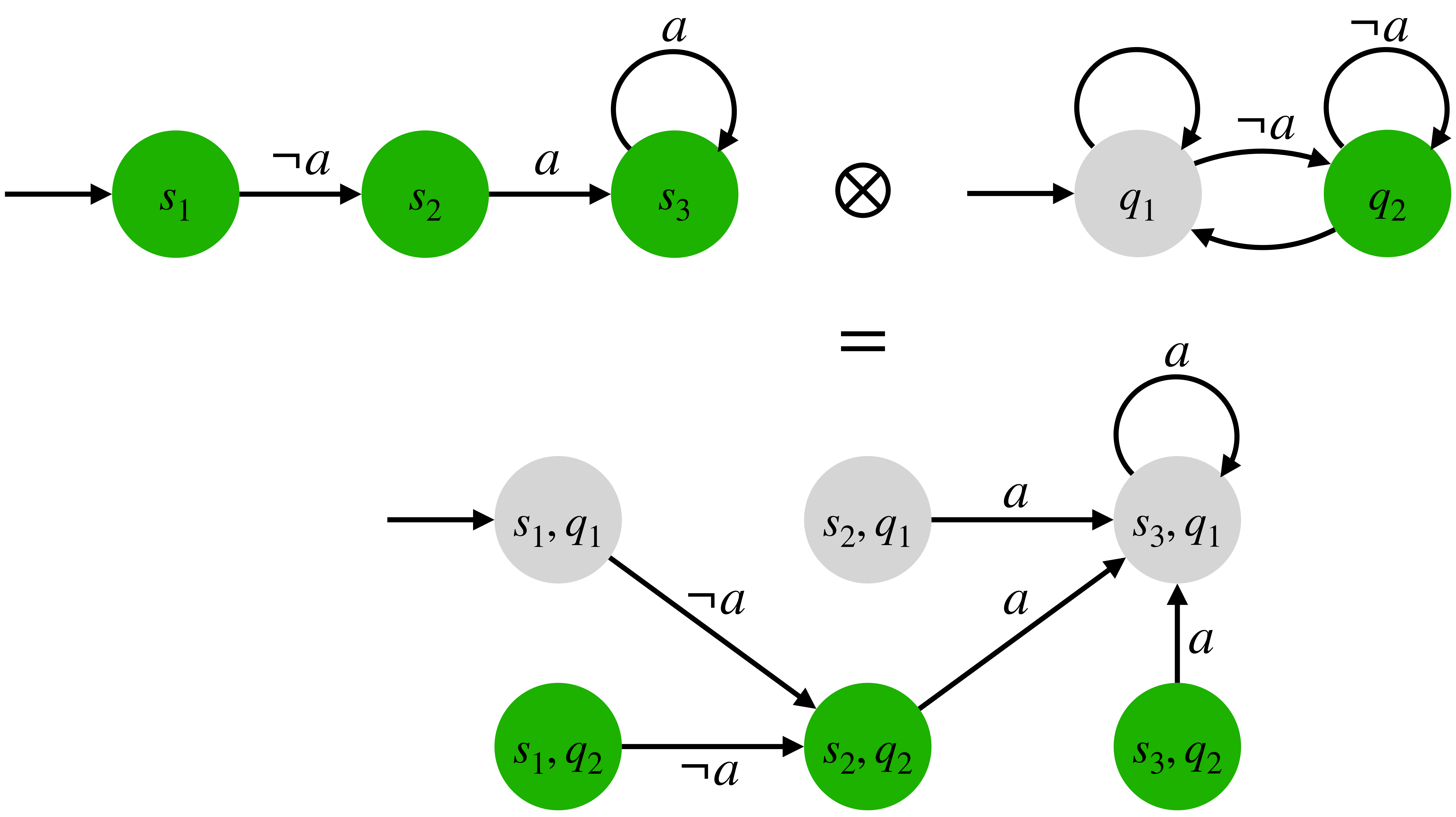
- Since all states of $\mathscr{A}_M$ are accepting, the product of $\mathscr{A}_M = (S_M, \Sigma, R_M, I_M, S_M)$ and $\mathscr{A}_{\neg f} = (S_{\neg f}, \Sigma, R_{\neg f}, I_{\neg f}, F_{\neg f})$ can be computed as follows

$$\mathscr{A}_M \otimes \mathscr{A}_{\neg f} = (S_M \times S_{\neg f}, \Sigma, R, I_M \times I_{\neg f}, S_M \times F_{\neg f})$$

$$((s_0, s_1), a, (s_0', s_1')) \in R \quad \text{iff} \quad (s_0, a, s_0') \in R_M \wedge (s_1, a, s_1') \in R_{\neg f}$$

Explicit LTL Model Checking

$$M \models \mathsf{F}\,\mathsf{G}\,a \quad \text{iff} \quad \mathscr{L}(\mathscr{A}_M \otimes \mathscr{A}_{\mathsf{G}\,\mathsf{F}\,\neg a})$$



Checking Automata Emptiness

- Check if a nontrivial SCC containing an accepting state is reachable from the initial state
  - Typically requires storing the entire automaton in memory
- Determine the reachable states using DFS and if an accepting state is reachable run a nested DFS to determine if there is a cycle
  - Better for on-the-fly model checking
- Use a CTL model checker to verify if $\mathrm{EG}\ \top$ is valid assuming the system is fair to the accepting states
  - Enables symbolic model checking for LTL