Computer-Aided Reasoning for Software

CSE507

Model Checking I

courses.cs.washington.edu/courses/cse507/14au/

Emina Torlak
emina@cs.washington.edu

Today

Today

Last lecture

- Symbolic execution and concolic testing

Today

Last lecture

- Symbolic execution and concolic testing

Today

- Introduction to model checking

Today

Last lecture

- Symbolic execution and concolic testing

Today

- Introduction to model checking

Reminders

- Homework 3 is due on Tuesday, November 18, at 11pm

Today

Last lecture

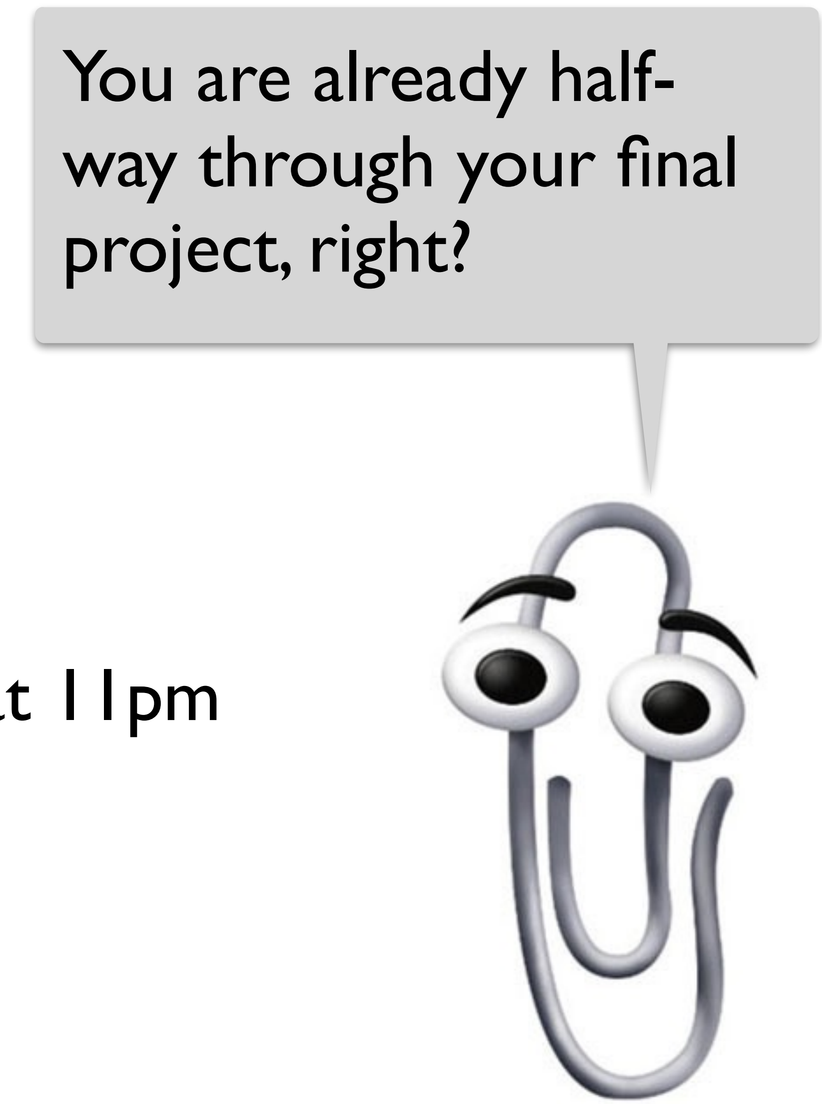
- Symbolic execution and concolic testing

Today

- Introduction to model checking

Reminders

- Homework 3 is due on Tuesday, November 18, at 11pm



What is model checking?

An automated technique for verifying that a concurrent finite state system satisfies a given temporal property.

$$M, s \models P$$

What is model checking?

An automated technique for verifying that a concurrent finite state system satisfies a given temporal property.

$$M, s \models P$$

A mathematical model of the system, given as a **Kripke structure** (a finite state machine).

What is model checking?

An automated technique for verifying that a concurrent finite state system satisfies a given temporal property.

A state of the system (e.g., an initial state).

$$M, s \models P$$

A mathematical model of the system, given as a **Kripke structure** (a finite state machine).

What is model checking?

An automated technique for verifying that a concurrent finite state system satisfies a given temporal property.

A state of the system (e.g., an initial state).

A **temporal logic** formula (e.g., a request is *eventually* acknowledged).

$$M, s \models P$$

A mathematical model of the system, given as a **Kripke structure** (a finite state machine).

Why model checking?

Model checking

Classic & bounded verification

Why model checking?

Model checking

Classic & bounded verification

- Deterministic, single-threaded, possibly infinite-state, terminating programs.
- Fully described by their input/output behavior.
- Semi-automatic or bounded-automatic checking of properties in expressive logics (e.g., FOL).

Why model checking?

Model checking

- *Reactive systems*: concurrent finite-state programs with *ongoing* input/output behavior.
- *Control-intensive* but without a lot of data manipulation.
- Fully automatic checking of properties in less expressive (temporal) logics.

Classic & bounded verification

- Deterministic, single-threaded, possibly infinite-state, terminating programs.
- Fully described by their input/output behavior.
- Semi-automatic or bounded-automatic checking of properties in expressive logics (e.g., FOL).

Why model checking?

Model checking

- *Reactive systems*: concurrent finite-state programs with *ongoing* input/output behavior.
- *Control-intensive* but without a lot of data manipulation.
- Fully automatic checking of properties in less expressive (temporal) logics.

Classic & bounded verification

- Deterministic, single-threaded, possibly infinite-state, terminating programs.
- Fully described by their input/output behavior.
- Semi-automatic or bounded-automatic checking of properties in expressive logics (e.g., FOL).

- Microprocessors and device drivers
- Embedded controllers (e.g., cars, planes)
- Protocols (e.g., cache coherence)

Why model checking?

Model checking

- *Reactive systems*: concurrent finite-state programs with *ongoing* input/output behavior.
- *Control-intensive* but without a lot of data manipulation.
- Fully automatic checking of properties in less expressive (temporal) logics.

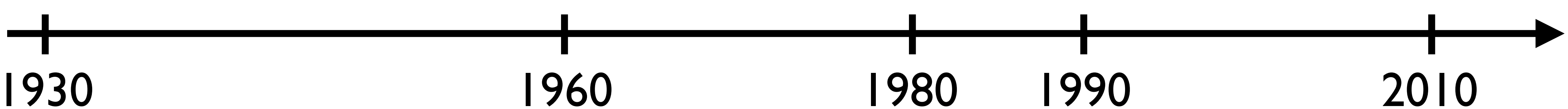
- Microprocessors and device drivers
- Embedded controllers (e.g., cars, planes)
- Protocols (e.g., cache coherence)

Classic & bounded verification

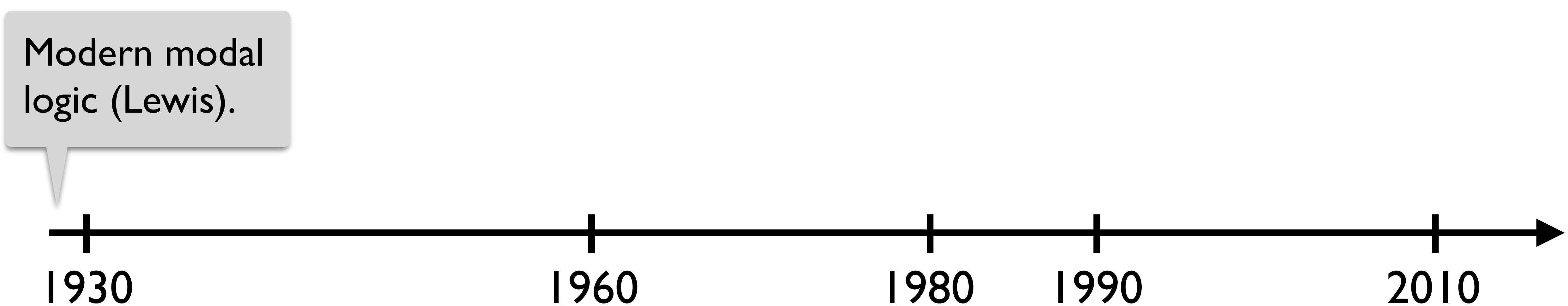
- Deterministic, single-threaded, possibly infinite-state, terminating programs.
- Fully described by their input/output behavior.
- Semi-automatic or bounded-automatic checking of properties in expressive logics (e.g., FOL).

- Libraries and ADT implementations
- Heap-manipulating programs (e.g., OO)
- Tricky deterministic algorithms

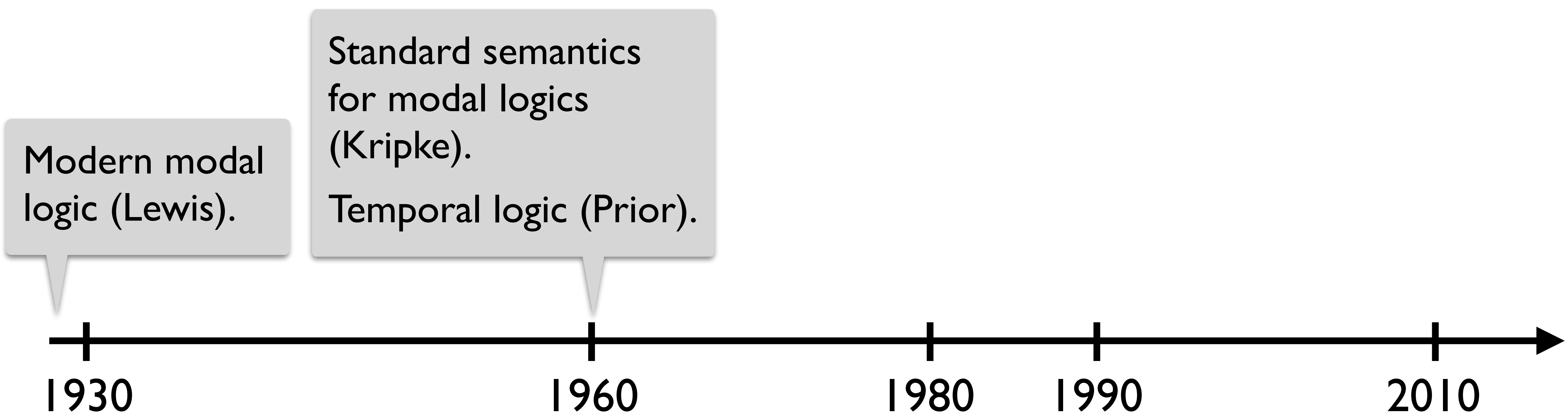
A brief history of model checking



A brief history of model checking



A brief history of model checking



A brief history of model checking

Modern modal logic (Lewis).

Standard semantics for modal logics (Kripke).

Temporal logic (Prior).

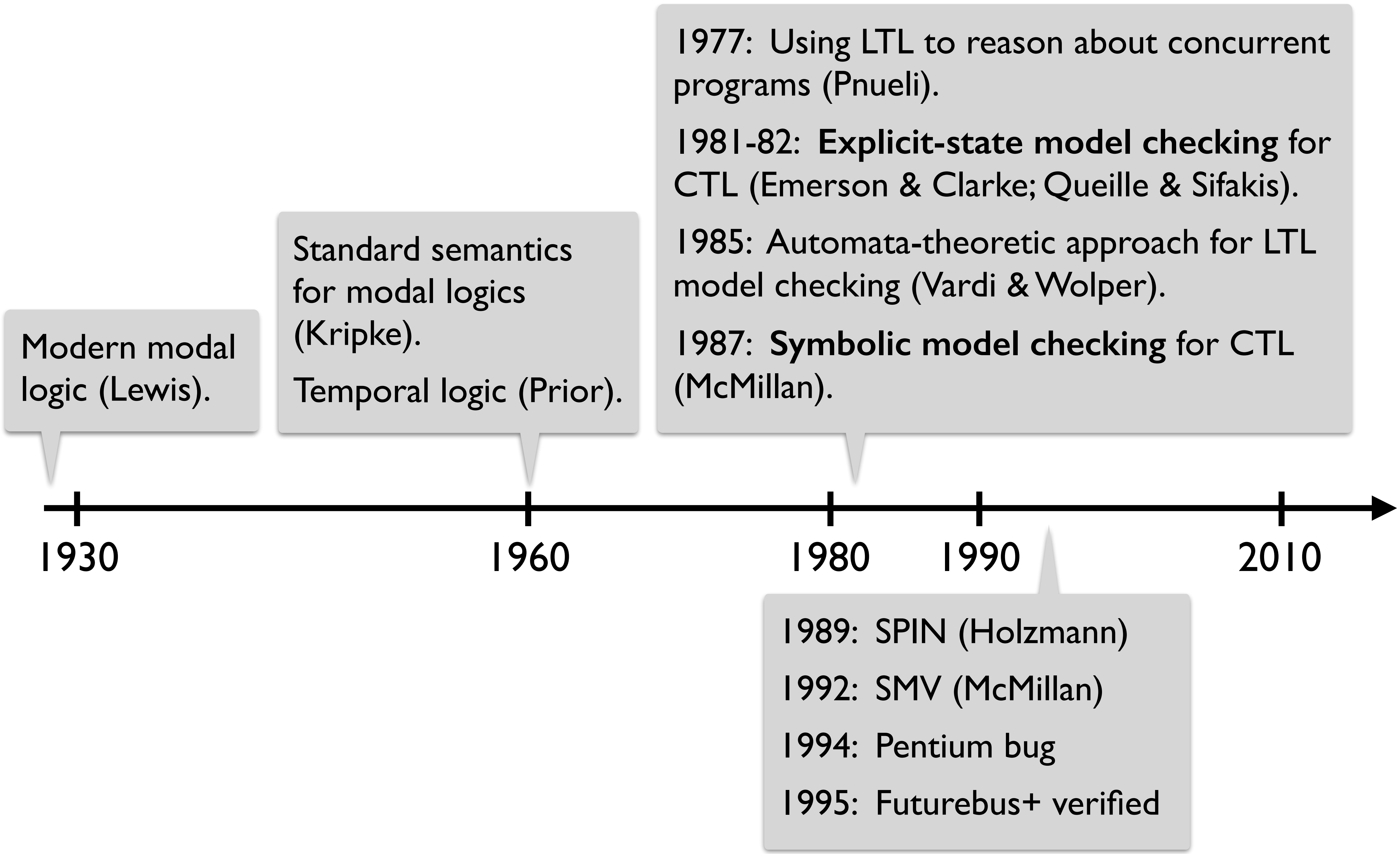
1977: Using LTL to reason about concurrent programs (Pnueli).

1981-82: **Explicit-state model checking** for CTL (Emerson & Clarke; Queille & Sifakis).

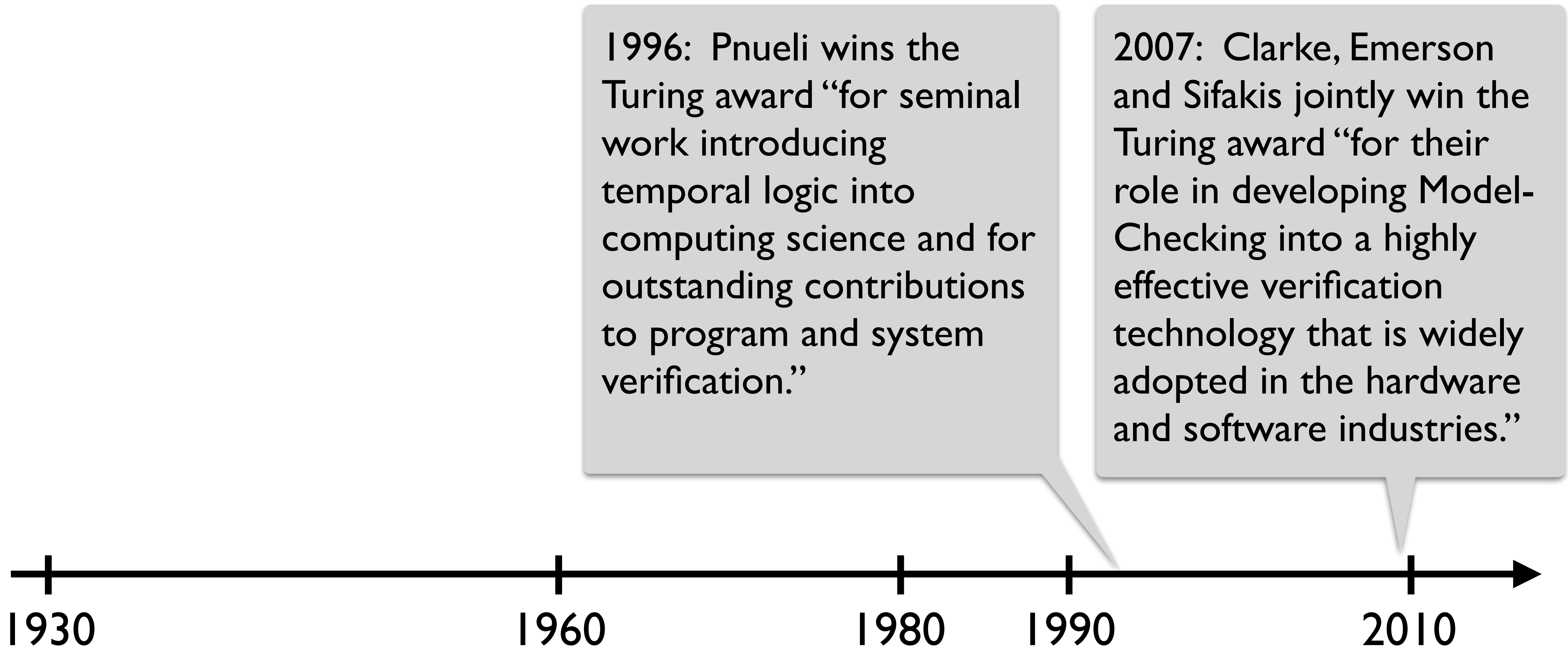
1985: Automata-theoretic approach for LTL model checking (Vardi & Wolper).

1987: **Symbolic model checking** for CTL (McMillan).

A brief history of model checking



A brief history of model checking



Kripke structures

Kripke structures

A Kripke structure is a tuple $M = \langle S, S_0, R, L \rangle$

Kripke structures

A Kripke structure is a tuple $M = \langle S, S_0, R, L \rangle$

- S is a finite set of states.

Kripke structures

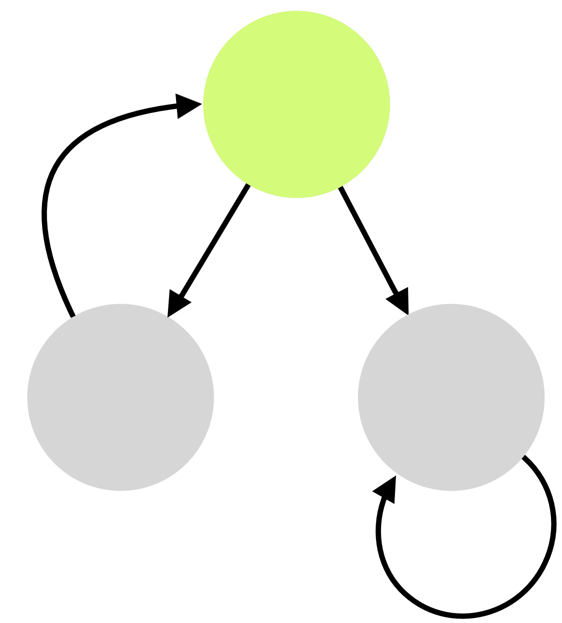
A Kripke structure is a tuple $M = \langle S, S_0, R, L\rangle$

- S is a finite set of states.
- $S_0 \subseteq S$ is the set of initial states.

Kripke structures

A Kripke structure is a tuple $M = \langle S, S_0, R, L \rangle$

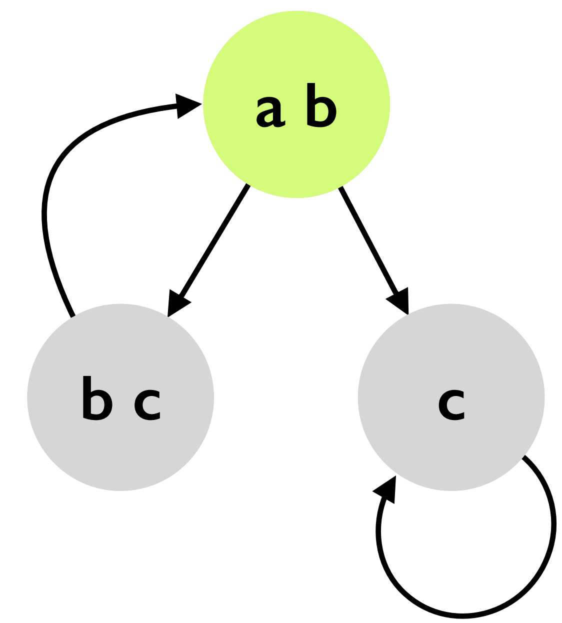
- S is a finite set of states.
- $S_0 \subseteq S$ is the set of initial states.
- $R \subseteq S \times S$ is the transition relation, which must be *total*.



Kripke structures

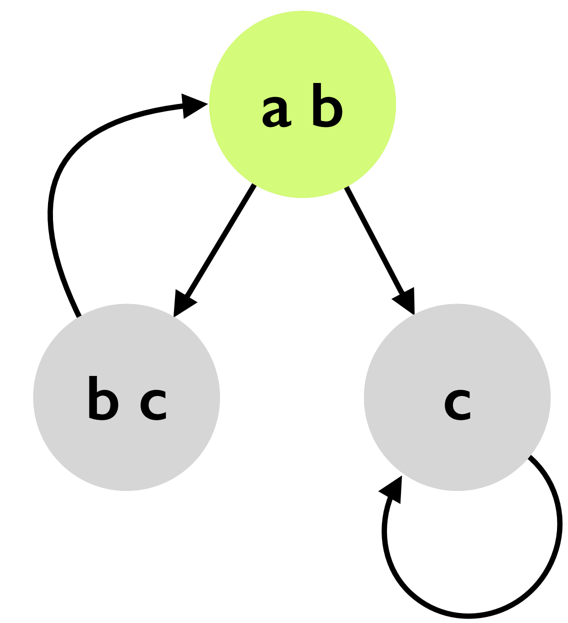
A Kripke structure is a tuple $M = \langle S, S_0, R, L\rangle$

- S is a finite set of states.
- $S_0 \subseteq S$ is the set of initial states.
- $R \subseteq S \times S$ is the transition relation, which must be *total.*
- $L : S \rightarrow 2^{AP}$ is a function that *labels* each state with a set of *atomic propositions* true in that state.



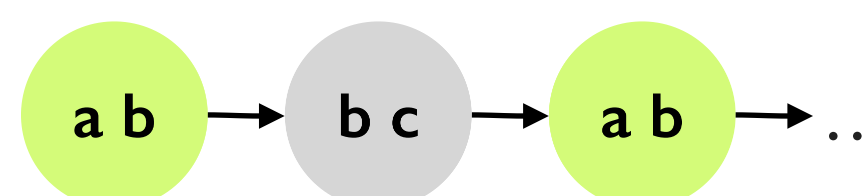
Kripke structures

A Kripke structure is a tuple $M = \langle S, S_0, R, L\rangle$



- S is a finite set of states.
- $S_0 \subseteq S$ is the set of initial states.
- $R \subseteq S \times S$ is the transition relation, which must be *total*.
- $L : S \to 2^{AP}$ is a function that *labels* each state with a set of *atomic propositions* true in that state.

A *path* in M **is an infinite sequence of states** $\pi = s_0 s_1 \ldots$ **such that for all** $i \geq 0$, $(s_i, s_{i+1}) \in R$**.**



Modeling systems with Kripke structures

```
// x==1, y==1
x := (x + y) % 2
```

- In a finite-state program, *system variables* V range over a *finite domain* D: V = {x, y} and D = {0, 1}.
- A *state* of the system is a *valuation* $s : V \to D$.

Modeling systems with Kripke structures

```
// x==1, y==1
x := (x + y) % 2
```

$S \equiv (x = 0 \lor x = 1) \land (y = 0 \lor y = 1)$
$S_0 \equiv (x = 1) \land (y = 1)$
$R(x, y, x', y') \equiv (x' = (x + y) \% 2) \land (y' = y)$

- In a finite-state program, *system variables* V range over a *finite domain* D: V = {x, y} and D = {0, 1}.
- A *state* of the system is a *valuation* $s : V \rightarrow D$.
- Use FOL to describe the (initial) states and the transition relation.

Modeling systems with Kripke structures

```
// x==1, y==1
x := (x + y) % 2
```

$S \equiv (x = 0 \lor x = 1) \land (y = 0 \lor y = 1)$
$S_0 \equiv (x = 1) \land (y = 1)$
$R(x, y, x', y') \equiv (x' = (x + y) \% 2) \land (y' = y)$

- In a finite-state program, *system variables* V range over a *finite domain* D: V = {x, y} and D = {0, 1}.
- A *state* of the system is a *valuation* $s : V \rightarrow D$.
- Use FOL to describe the (initial) states and the transition relation.
- Extract a Kripke structure from the FOL description.

Modeling systems with Kripke structures

```
// x==1, y==1
x := (x + y) % 2
```

$S \equiv (x = 0 \lor x = 1) \land (y = 0 \lor y = 1)$

$S_0 \equiv (x = 1) \land (y = 1)$

$R(x, y, x', y') \equiv (x' = (x + y) \% 2) \land (y' = y)$

x=1, y=1 x=0, y=1 x=0, y=0 x=1, y=0

- In a finite-state program, *system variables* V range over a *finite domain* D: V = {x, y} and D = {0, 1}.
- A *state* of the system is a *valuation* $s : V \rightarrow D$.
- Use FOL to describe the (initial) states and the transition relation.
- Extract a Kripke structure from the FOL description.

Modeling systems with Kripke structures

```
// x==1, y==1
x := (x + y) % 2
```

$S \equiv (x = 0 \lor x = 1) \land (y = 0 \lor y = 1)$

$S_0 \equiv (x = 1) \land (y = 1)$

$R(x, y, x', y') \equiv (x' = (x + y)\ \%\ 2) \land (y' = y)$

x=1, y=1 | x=0, y=1 | x=0, y=0 | x=1, y=0

- In a finite-state program, *system variables* V range over a *finite domain* D: V = {x, y} and D = {0, 1}.
- A *state* of the system is a *valuation* $s : V \rightarrow D$.
- Use FOL to describe the (initial) states and the transition relation.
- Extract a Kripke structure from the FOL description.

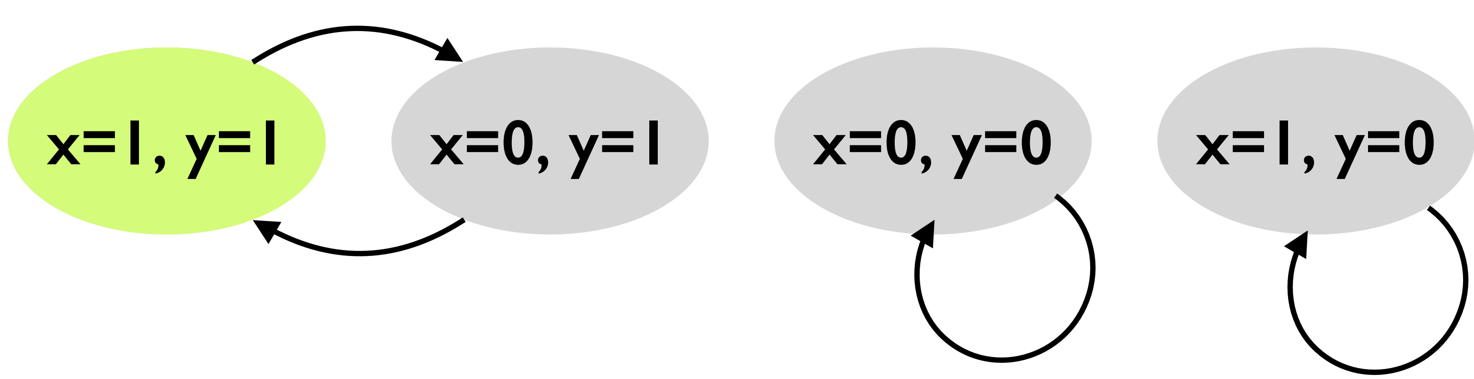
Modeling systems with Kripke structures

```
// x==1, y==1
x := (x + y) % 2
```

$S \equiv (x = 0 \lor x = 1) \land (y = 0 \lor y = 1)$

$S_0 \equiv (x = 1) \land (y = 1)$

$R(x, y, x', y') \equiv (x' = (x + y) \% 2) \land (y' = y)$



- In a finite-state program, *system variables* V range over a *finite domain* D: V = {x, y} and D = {0, 1}.
- A *state* of the system is a *valuation* $s : V \to D$.
- Use FOL to describe the (initial) states and the transition relation.
- Extract a Kripke structure from the FOL description.

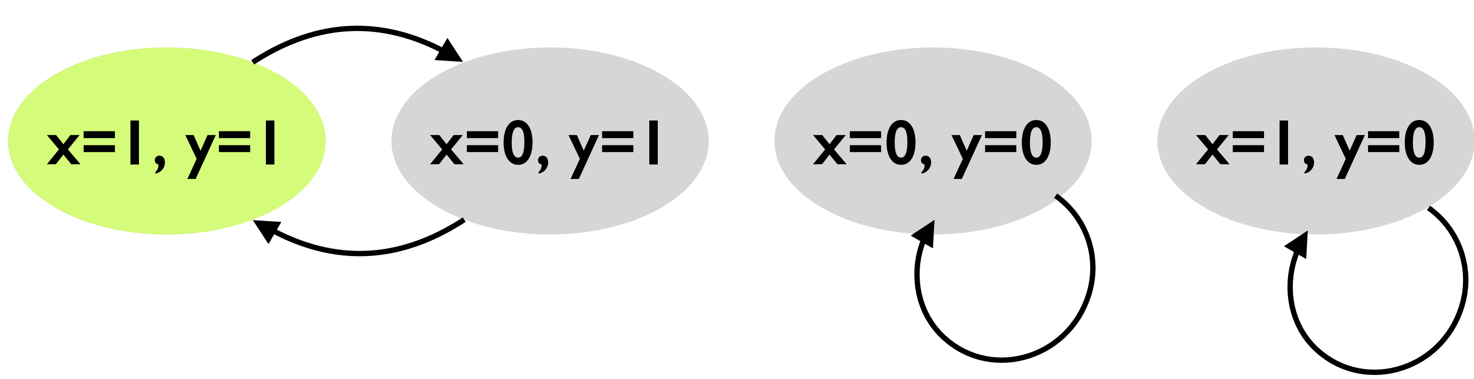
Modeling systems with Kripke structures

```
// x==1, y==1
x := (x + y) % 2
```

$S \equiv (x = 0 \vee x = 1) \wedge (y = 0 \vee y = 1)$

$S_0 \equiv (x = 1) \wedge (y = 1)$

$R(x, y, x', y') \equiv (x' = (x + y) \% 2) \wedge (y' = y)$



- In a finite-state program, *system variables* V range over a *finite domain* D: V = {x, y} and D = {0, 1}.
- A *state* of the system is a *valuation* $s : V \to D$.
- Use FOL to describe the (initial) states and the transition relation.
- Extract a Kripke structure from the FOL description.

State explosion: Kripke structure usually exponential in the size of the program.

A Kripke structure for a concurrent program

P_1

```
10  while (true) {
11    wait(turn == 0);
      // critical section
12    turn := 1;
13  }
```

P_2

```
20  while (true) {
21    wait(turn == 1);
      // critical section
22    turn := 0;
23  }
```

Two processes executing concurrently and asynchronously, using the shared variable `turn` to ensure ***mutual exclusion***:

They are never in the critical section at the same time.

A Kripke structure for a concurrent program

P_1

```
10  while (true) {
11    wait(turn == 0);
      // critical section
12    turn := 1;
13  }
```

P_2

```
20  while (true) {
21    wait(turn == 1);
      // critical section
22    turn := 0;
23  }
```

Two processes executing concurrently and asynchronously, using the shared variable `turn` to ensure ***mutual exclusion***:

They are never in the critical section at the same time.

State of the program described by the variable `turn` and the *program counters* for the two processes.

A Kripke structure for a concurrent program

P_1

```
10  while (true) {
11    wait(turn == 0);
      // critical section
12    turn := 1;
13  }
```

P_2

```
20  while (true) {
21    wait(turn == 1);
      // critical section
22    turn := 0;
23  }
```

A Kripke structure for a concurrent program

P_1

```
10  while (true) {
11    wait(turn == 0);
      // critical section
12    turn := 1;
13  }
```

P_2

```
20  while (true) {
21    wait(turn == 1);
      // critical section
22    turn := 0;
23  }
```

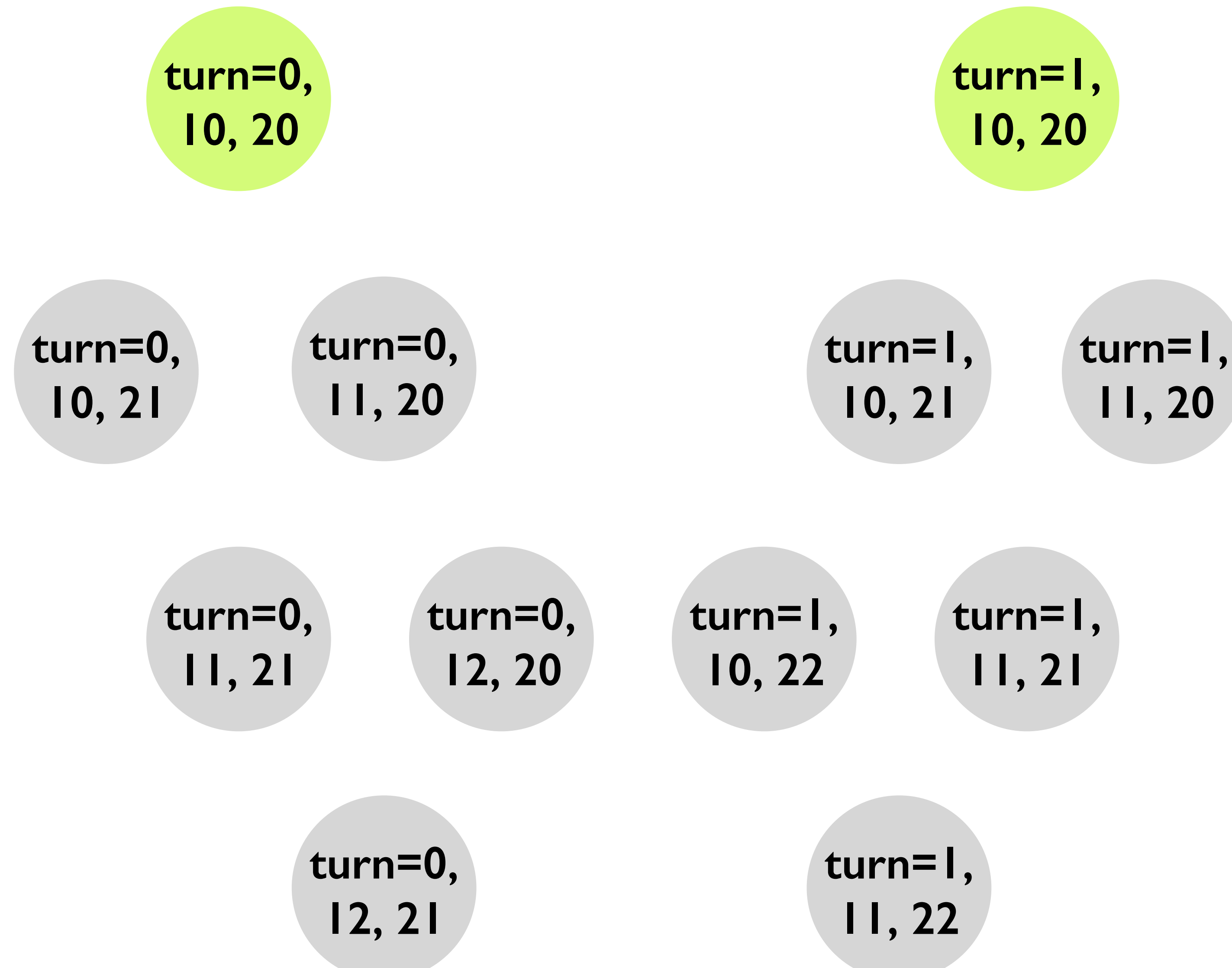
A Kripke structure for a concurrent program

P1

```
10  while (true) {
11    wait(turn == 0);
      // critical section
12    turn := 1;
13  }
```

P2

```
20  while (true) {
21    wait(turn == 1);
      // critical section
22    turn := 0;
23  }
```



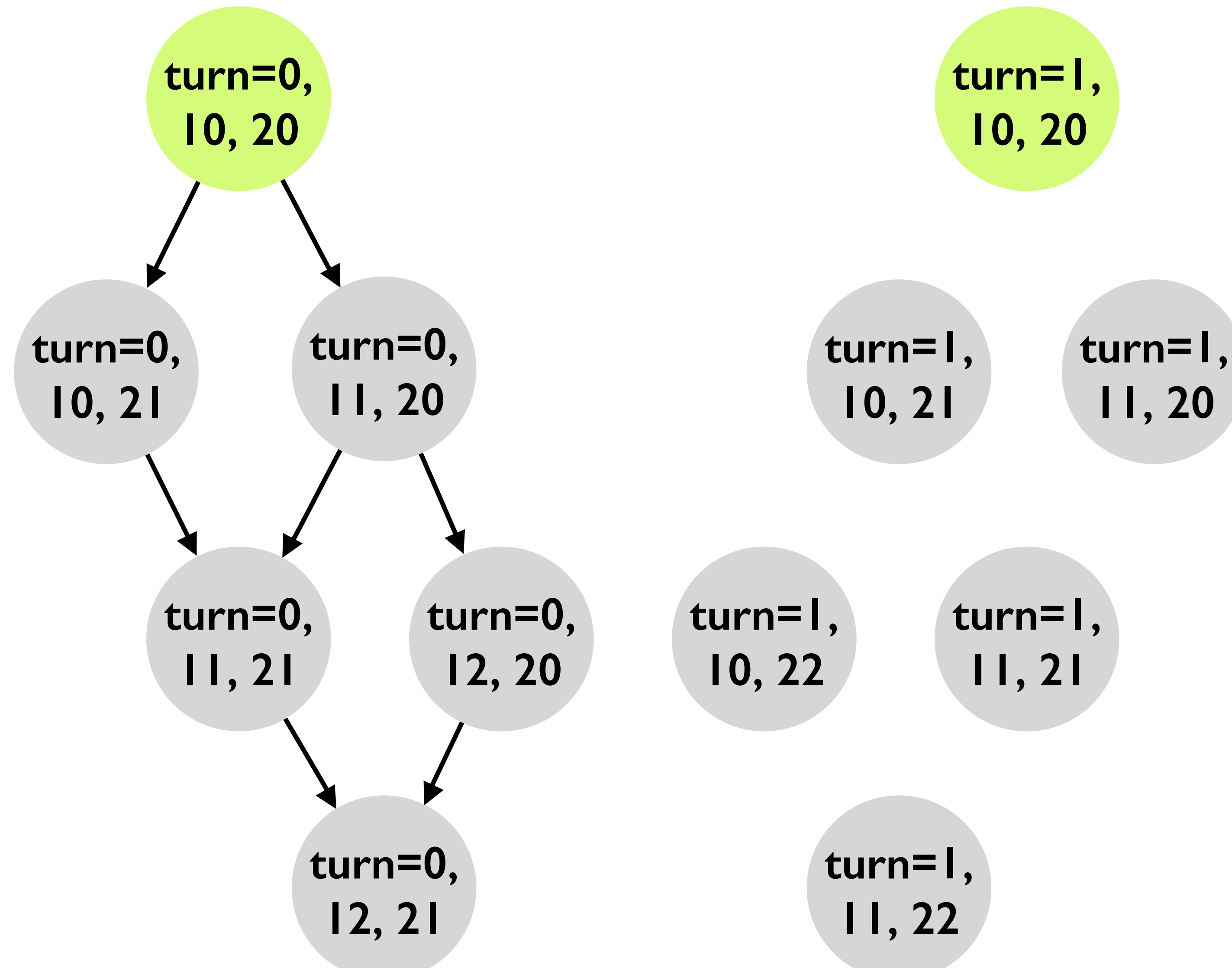
A Kripke structure for a concurrent program

P_1

```
10  while (true) {
11    wait(turn == 0);
      // critical section
12    turn := 1;
13  }
```

P_2

```
20  while (true) {
21    wait(turn == 1);
      // critical section
22    turn := 0;
23  }
```



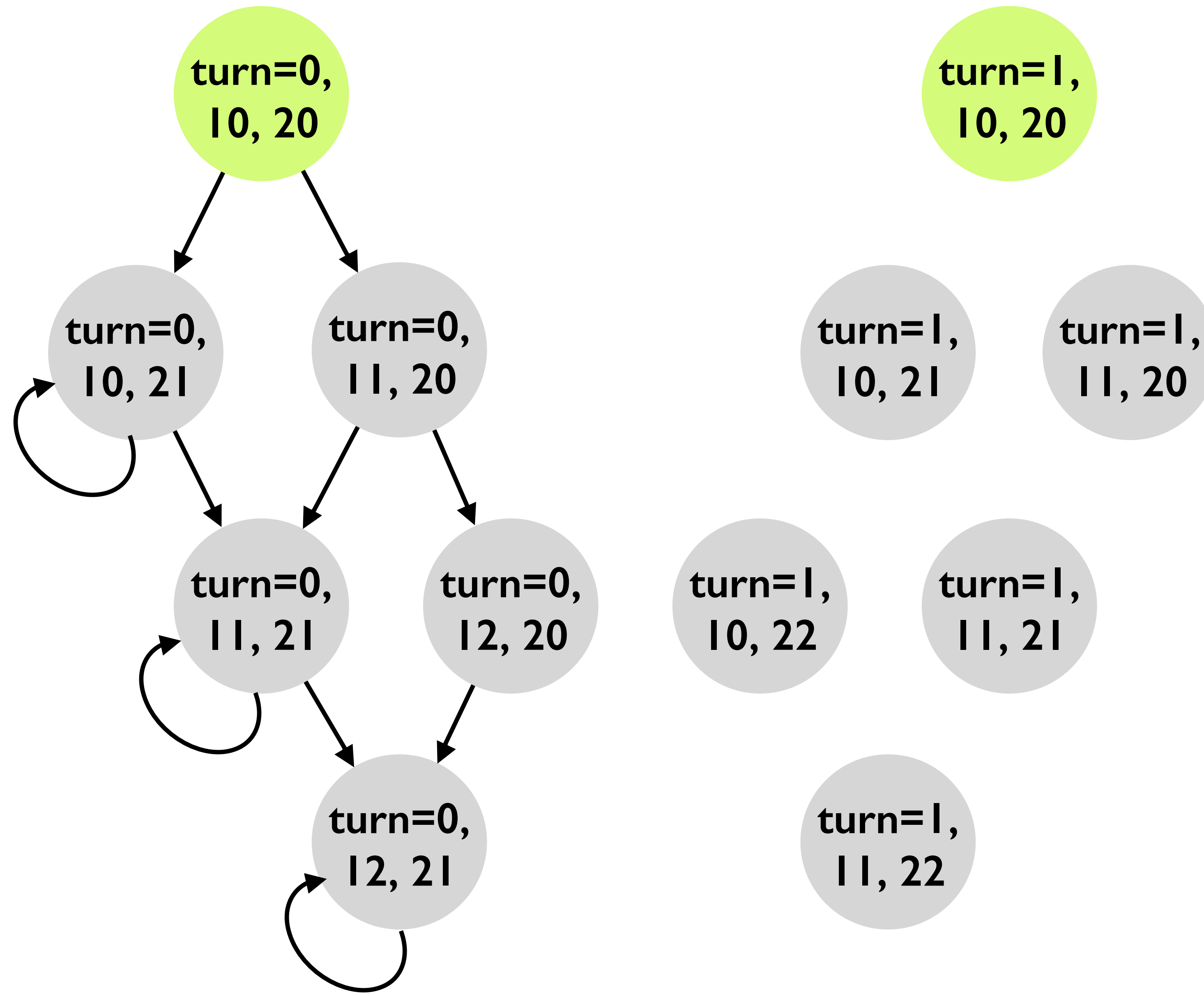
A Kripke structure for a concurrent program

P₁

```
10  while (true) {
11    wait(turn == 0);
      // critical section
12    turn := 1;
13  }
```

P₂

```
20  while (true) {
21    wait(turn == 1);
      // critical section
22    turn := 0;
23  }
```



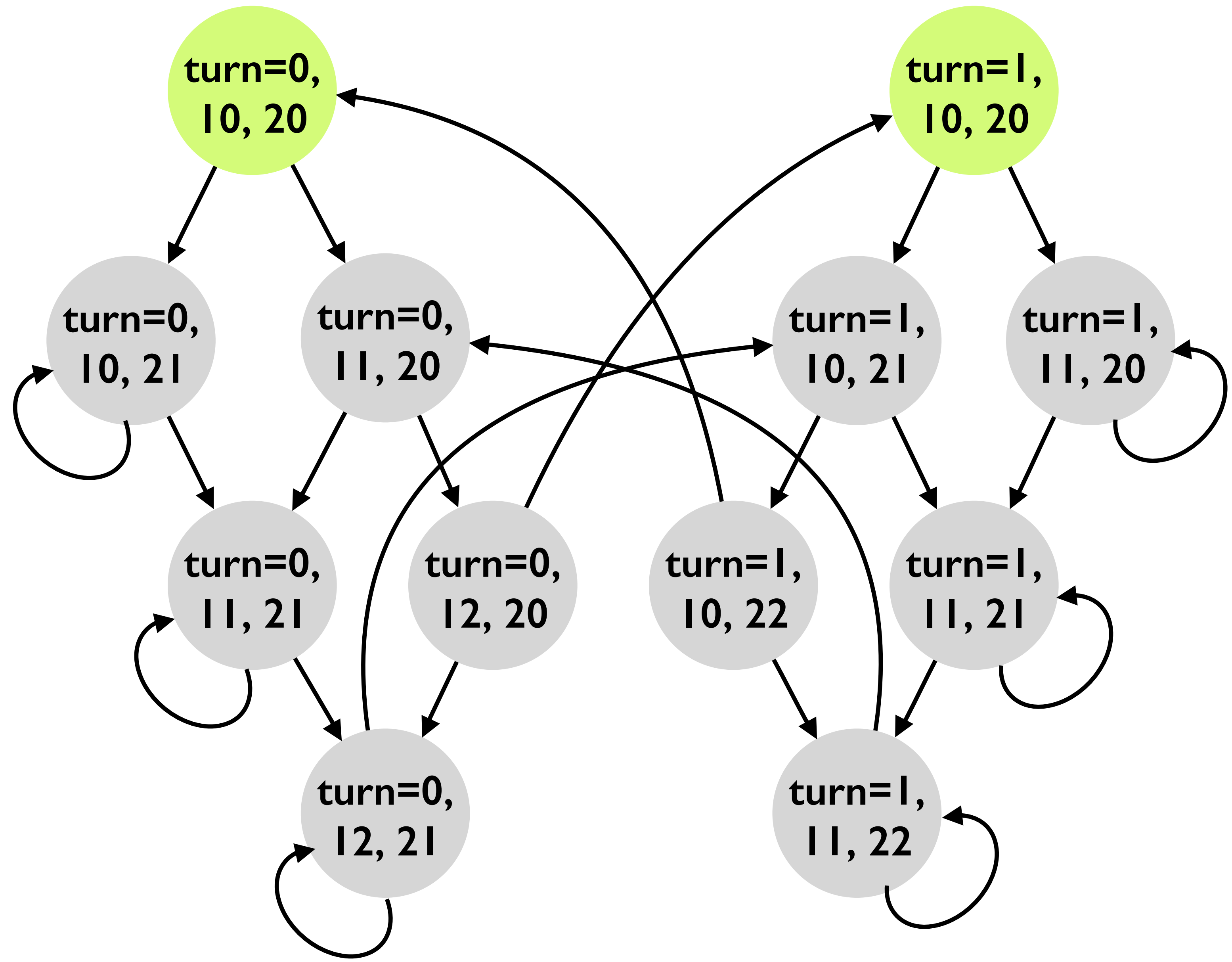
A Kripke structure for a concurrent program

P_1

```
10  while (true) {
11    wait(turn == 0);
      // critical section
12    turn := 1;
13  }
```

P_2

```
20  while (true) {
21    wait(turn == 1);
      // critical section
22    turn := 0;
23  }
```



Safety & liveness properties of reactive systems

Safety

- "Nothing bad will happen."
- φ is a safety property iff every infinite path π violating φ has a **finite prefix** π' such that every extension of π' violates φ.

Liveness

- "Something good will happen."
- ψ is a liveness property iff every finite path (prefix) π can be extended so that it satisfies ψ.

Safety & liveness properties of reactive systems

Safety

- "Nothing bad will happen."
- φ is a safety property iff every infinite path π violating φ has a **finite prefix** π' such that every extension of π' violates φ.

Finite witnesses (counterexamples).

Reducible to checking reachability in the state transition graph.

Liveness

- "Something good will happen."
- ψ is a liveness property iff every finite path (prefix) π can be extended so that it satisfies ψ.

Safety & liveness properties of reactive systems

Safety

- "Nothing bad will happen."
- φ is a safety property iff every infinite path π violating φ has a **finite prefix** π' such that every extension of π' violates φ.

Finite witnesses (counterexamples).

Reducible to checking reachability in the state transition graph.

Liveness

- "Something good will happen."
- ψ is a liveness property iff every finite path (prefix) π can be extended so that it satisfies ψ.

No finite witnesses (counterexamples).

Safety & liveness properties of reactive systems

Safety

- "Nothing bad will happen."
- φ is a safety property iff every infinite path π violating φ has a **finite prefix** π' such that every extension of π' violates φ.

Liveness

- "Something good will happen."
- ψ is a liveness property iff every finite path (prefix) π can be extended so that it satisfies ψ.

Mutual exclusion: P_1 and P_2 will never be in their critical regions simultaneously.

Safety & liveness properties of reactive systems

Safety

- "Nothing bad will happen."
- φ is a safety property iff every infinite path π violating φ has a **finite prefix** π' such that every extension of π' violates φ.

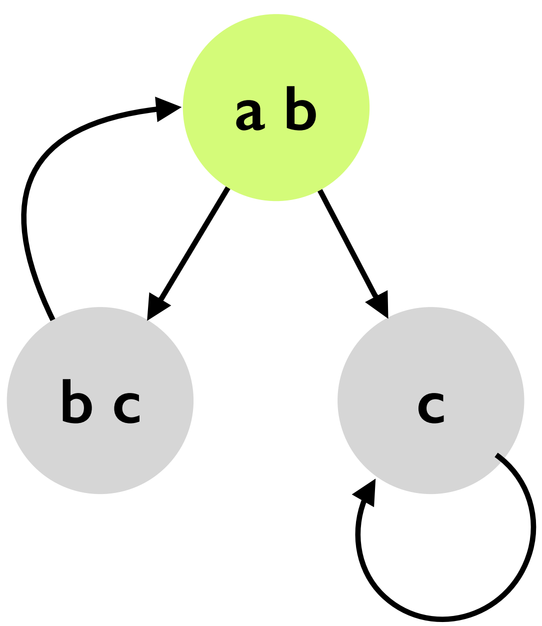
Mutual exclusion: P_1 and P_2 will never be in their critical regions simultaneously.

Liveness

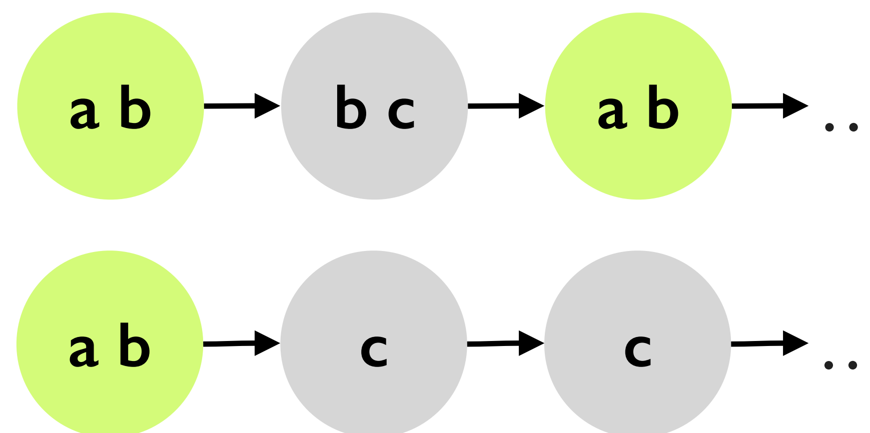
- "Something good will happen."
- ψ is a liveness property iff every finite path (prefix) π can be extended so that it satisfies ψ.

Starvation freedom: whenever P_1 is ready to enter its critical section, it will eventually succeed (provided that the scheduler is *fair* and does not let P_2 stay in its critical section forever).

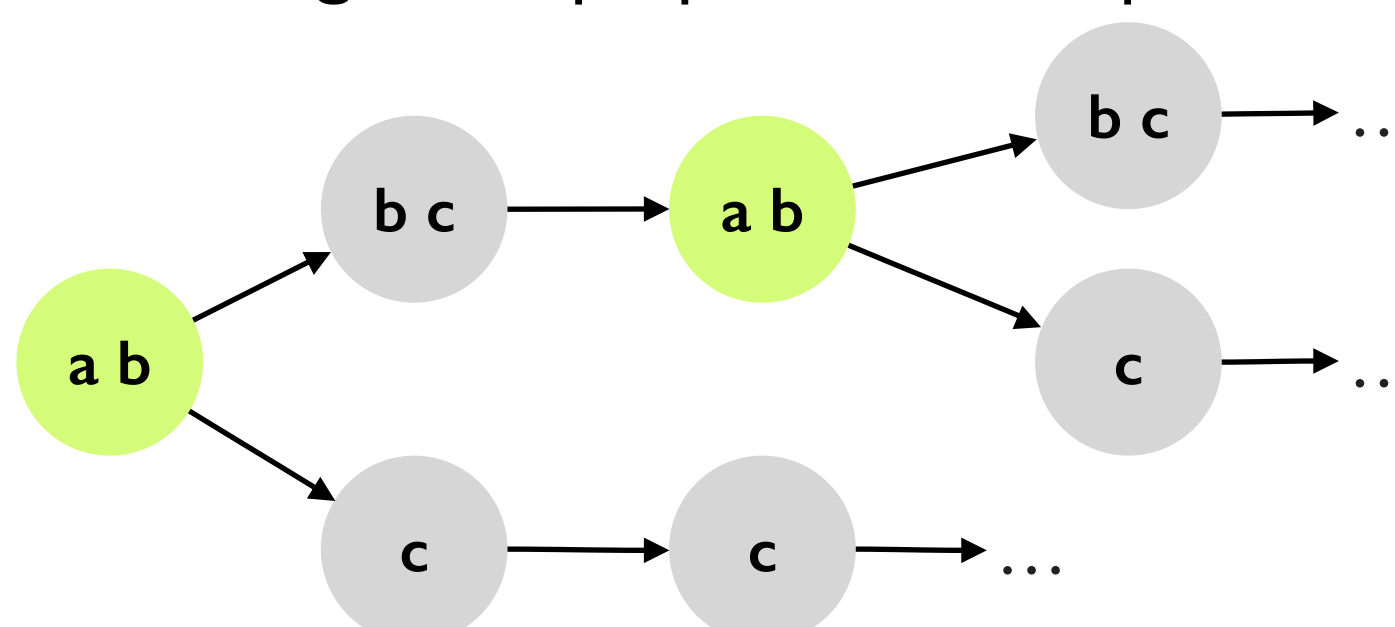
Expressing properties in temporal logics



Linear time: properties of computation paths



Branching time: properties of computation trees



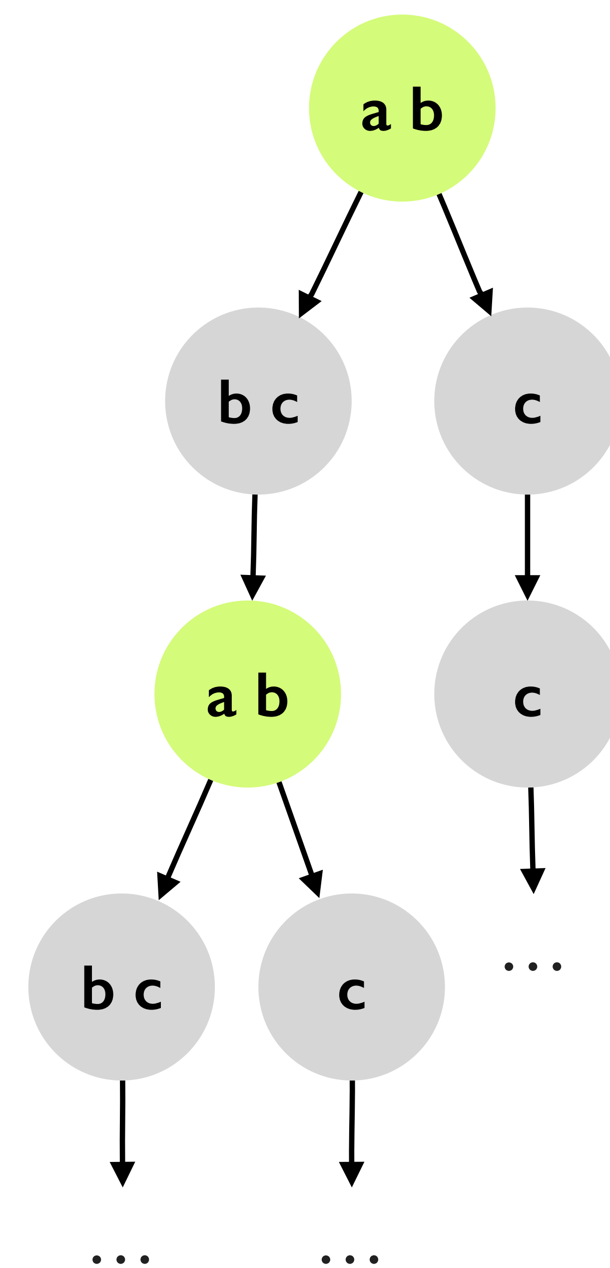
Computation tree logic CTL*

Path quantifiers describe the branching structure of the computation tree:

- **A** (for all paths)
- **E** (there exists a path)

Temporal operators describe properties of a path through a tree:

- **X**p (p holds "next time")
- **F**p (p holds "eventually" or "in the future")
- **G**p (p holds "always" or "globally")
- p **U** q (p holds "until" q holds)



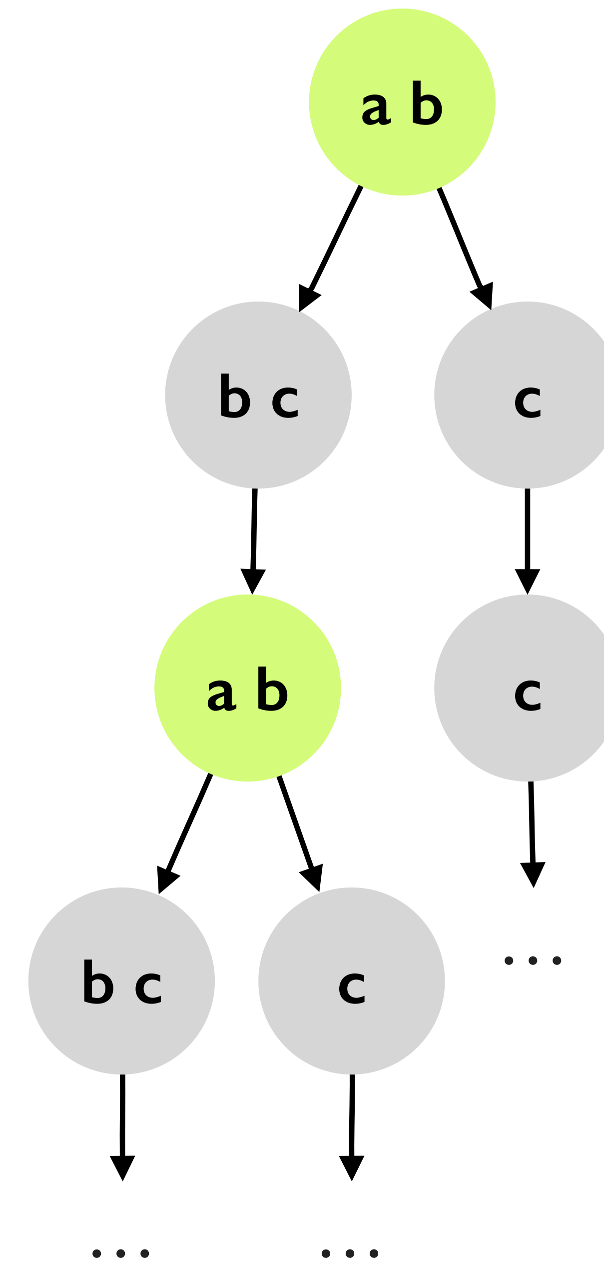
Syntax of CTL*

State formulas

- Atomic propositions: $a \in AP$
- $\neg f$, $f \wedge g$, $f \vee g$, where f and g are state formulas
- **A**p and **E**p, where p is a path formula

Path formulas

- f, where f is a state formula
- $\neg p$, $p \wedge p$, $p \vee q$, where p and q are path formulas
- **X**p, **F**p, **G**p, p **U** q, where p and q are path formulas



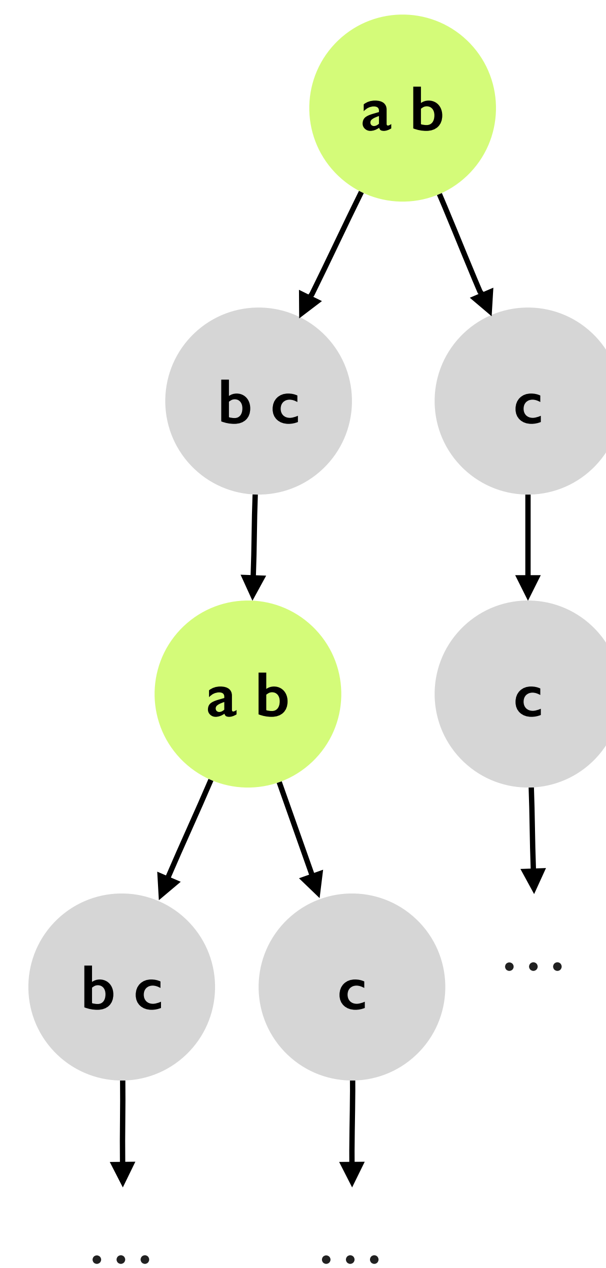
Semantics of CTL*

State formulas

- M, s $\vDash$ a iff a $\in$ L(s)
- M, s $\vDash$ **A**p iff M, $\pi \vDash$ p for all paths π that start at s
- M, s $\vDash$ **E**p iff M, $\pi \vDash$ p for some path π that starts at s

Path formulas (π^k is suffix of π starting at s_k)

- M, $\pi \vDash$ f iff M, s $\vDash$ f and s is the first state of π
- M, $\pi \vDash$ **X**p iff M, $\pi^1 \vDash$ p
- M, $\pi \vDash$ **F**p iff M, $\pi^k \vDash$ p for some k $\geq$ 0
- M, $\pi \vDash$ **G**p iff M, $\pi^k \vDash$ p for all k $\geq$ 0
- M, $\pi \vDash$ p **U** q iff M, $\pi^k \vDash$ q and M, $\pi^j \vDash$ q for some k $\geq$ 0 and for all 0 $\leq$ j < k



CTL and Linear Temporal Logic (LTL)

Computation Tree Logic (CTL)

Linear Temporal Logic (LTL)

CTL and Linear Temporal Logic (LTL)

Computation Tree Logic (CTL)

- Fragment of CTL* in which each temporal operator is prefixed with a path quantifier.
- **AG**(**EF** p): From any state, it is possible to get to a state where p holds.

Linear Temporal Logic (LTL)

CTL and Linear Temporal Logic (LTL)

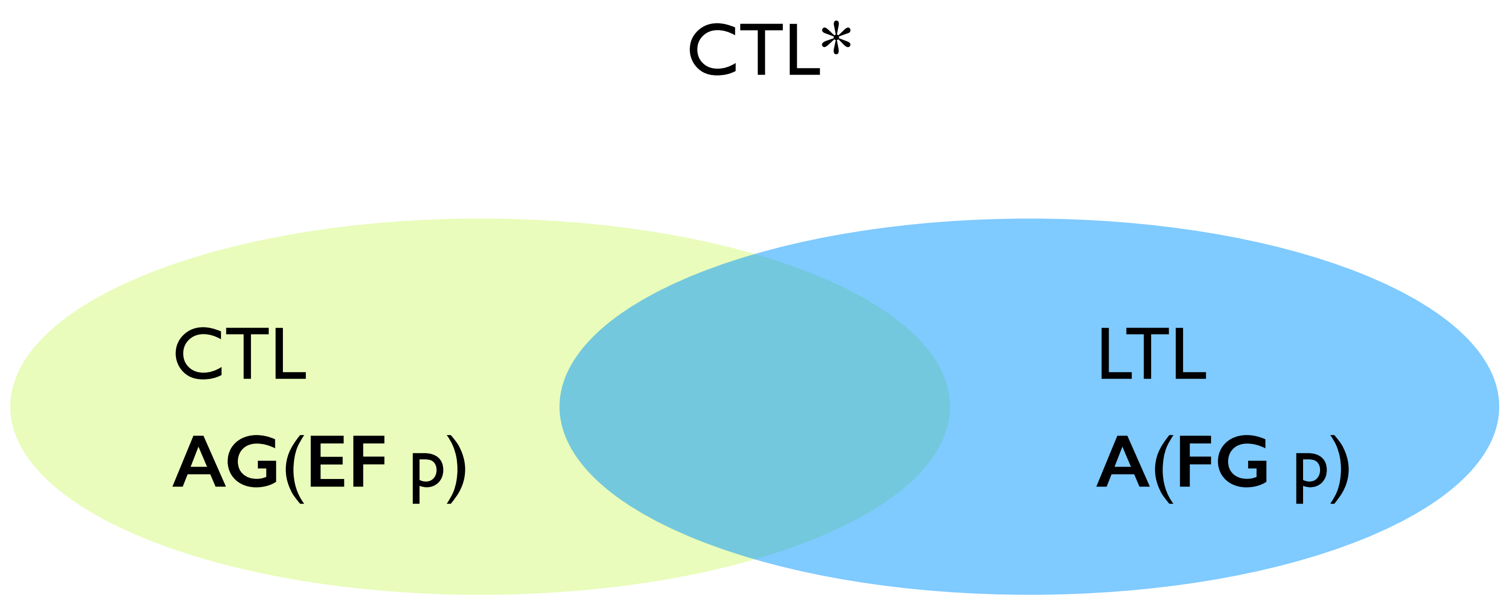
Computation Tree Logic (CTL)

- Fragment of CTL* in which each temporal operator is prefixed with a path quantifier.
- **AG**(**EF** p): From any state, it is possible to get to a state where p holds.

Linear Temporal Logic (LTL)

- Fragment of CTL* with formulas of the form **A**p, where p contains no path quantifiers.
- **A**(**FG** p): Along every path, there is some state from which p will hold forever.

Expressive power of CTL, LTL, and CTL*



Fairness

Cannot be expressed in CTL

Can be expressed in LTL

Fairness

Cannot be expressed in CTL

- Handled by changing the semantics to use fair Kripke structures.
- A *fair* Kripke structure $M = \langle S, S_0, R, L, F \rangle$ includes an additional set of sets of states $F \subseteq 2^S$.
- For each $P \in F$, a *fair path* π includes some states from P infinitely often.
- Path quantifiers interpreted only with respect to fair paths.

Can be expressed in LTL

Fairness

Cannot be expressed in CTL

- Handled by changing the semantics to use fair Kripke structures.
- A *fair* Kripke structure $M = \langle S, S_0, R, L, F\rangle$ includes an additional set of sets of states $F \subseteq 2^S$.
- For each $P \in F$, a *fair path* π includes some states from P infinitely often.
- Path quantifiers interpreted only with respect to fair paths.

Can be expressed in LTL

- Absolute fairness: $\mathbf{A}(\mathbf{GF}\ p_{exec})$
- Strong fairness:
  $\mathbf{A}((\mathbf{GF}\ p_{ready}) \Rightarrow (\mathbf{GF}\ p_{ready} \wedge p_{exec}))$
- Weak fairness:
  $\mathbf{A}((\mathbf{FG}\ p_{ready}) \Rightarrow (\mathbf{GF}\ p_{ready} \wedge p_{exec}))$

Model checking complexity for CTL, LTL, CTL*

Polynomial Time for CTL

- Best known algorithm: $O(|M| * |f|)$

$M, s \models f$

PSPACE-complete for LTL

- Best known algorithm: $O(|M| * 2^{|f|})$

PSPACE-complete for CTL*

- Best known algorithm: $O(|M| * 2^{|f|})$

Model checking techniques for CTL and LTL

CTL

- Graph-theoretic explicit-state model checking (EMC)
- Symbolic model checking with Ordered Binary Decision Diagrams (SMV, NuSMV)
- Bounded model checking based on SAT (NuSMV)

LTL

- Automata-theoretic model checking:
  - Explicit-state (SPIN) or
  - Symbolic (NuSMV)

Summary

Today

- Basics of model checking:
  - Kripke structures
  - Temporal logics (CTL, LTL,CTL*)
  - Model checking techniques

Next lecture

- Software model checking