Model Checking

Tom Henzinger

IST Austria

Exercise

1. Draw a state-transition graph that generates the Roman numerals
2. Define the property “there are no more than 3 adjacent I” using
    a. LTL
    b. a specification automaton
    c. a monitor automaton
3. Use one of the three specifications to model check the property (show all intermediate steps)

Model checking, narrowly interpreted:

Decision procedures for checking if a given Kripke structure is a model for a given formula of a modal logic.

Why is this of interest to us?

Because the dynamics of a discrete system can be captured by a Kripke structure.

Because some dynamic properties of a discrete system can be stated in modal logics.

$\Downarrow$

Model checking = System verification

Model checking, generously interpreted:

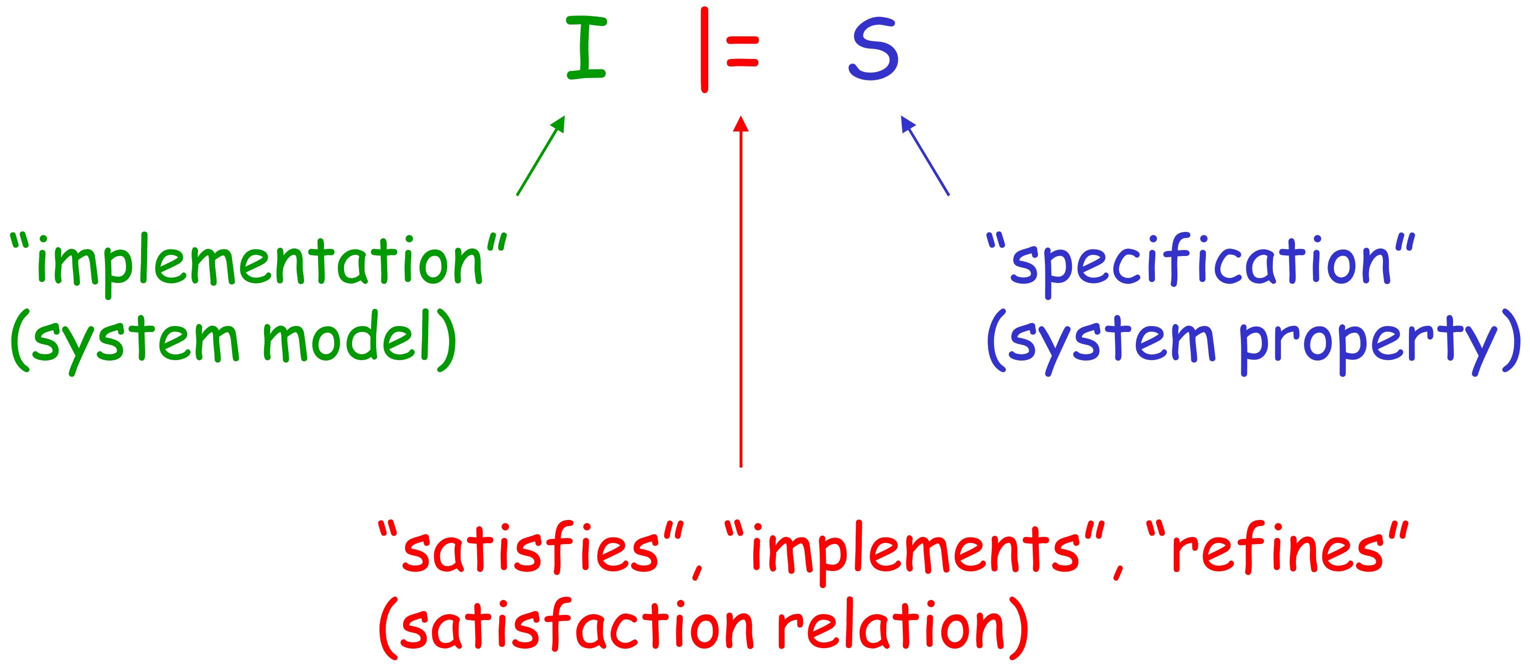
Algorithms, rather than proof calculi,
for system verification which operate on
a system model (semantics), rather than
a system description (syntax).

There are many different model-checking problems:

for different (classes of) system models

for different (classes of) system properties

A specific model-checking problem is defined by



Characteristics of system models which favor model checking over other verification techniques:

ongoing input/output behavior
(not: single input, single result)

concurrency
(not: single control flow)

control intensive
(not: lots of data manipulation)

Examples

-control logic of hardware designs

-communication protocols

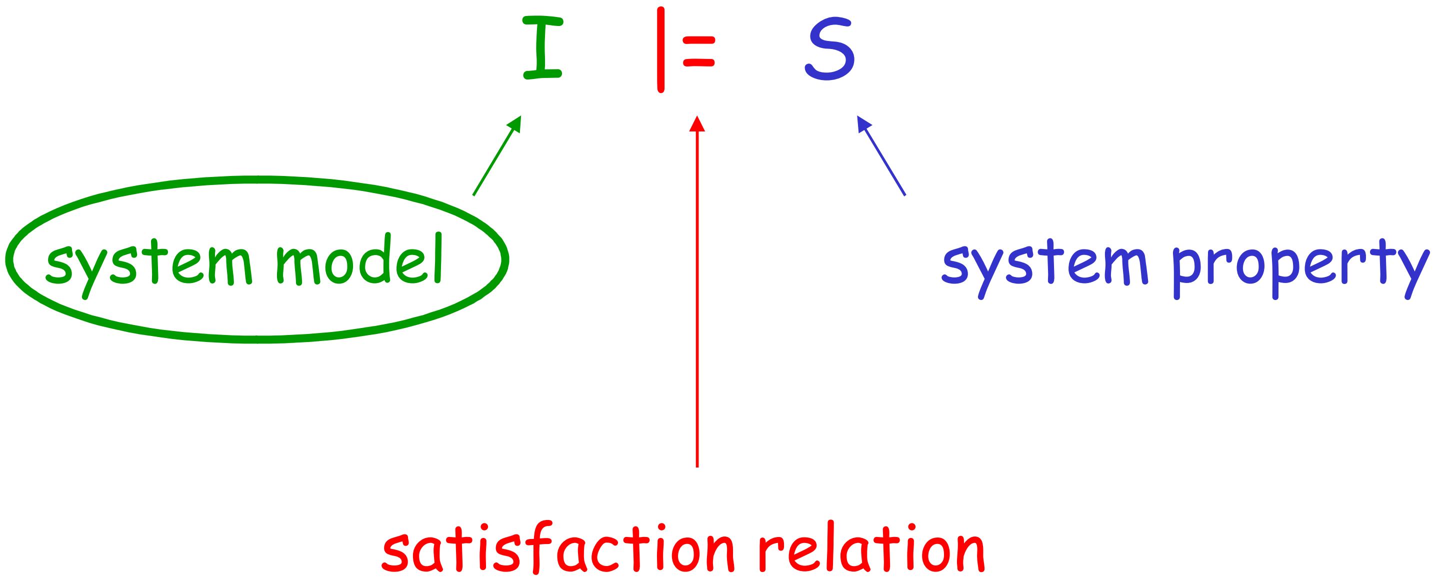
-device drivers

Paradigmatic example:
mutual-exclusion protocol

```
loop                                         ||  loop
   out:  x1 := 1; last := 1                        out:  x2 := 1; last := 2
   req:  await  x2 = 0  or  last = 2               req:  await  x1 = 0  or  last = 1
   in:     x1 := 0                                 in:     x2 := 0
end loop.                                        end loop.

             P1                                                P2
```

Model-checking problem



Important decisions when choosing a system model

-variable-based vs. event-based

-interleaving vs. true concurrency

-synchronous vs. asynchronous interaction

-clocked vs. speed-independent progress

-etc.

Particular combinations of choices yield

CSP

Petri nets

I/O automata

Reactive modules

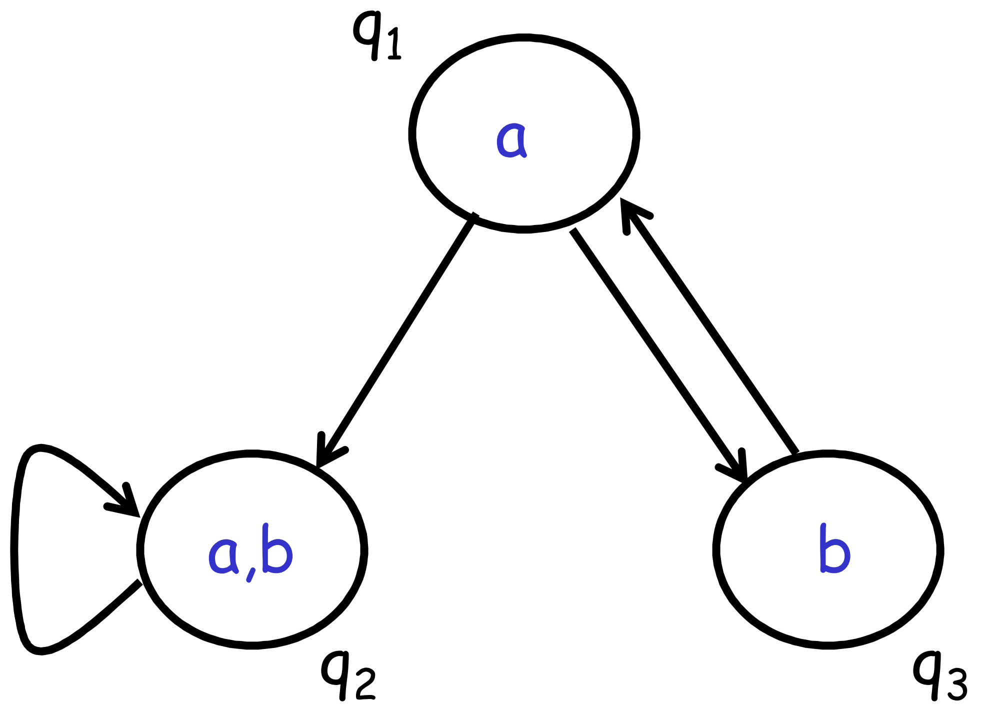
Verilog

C

etc.

While the choice of system model is important for the application,

the only thing that is important for model checking is that the system model can be translated into a state-transition graph.



q_1
a
q_2
a,b
q_3
b

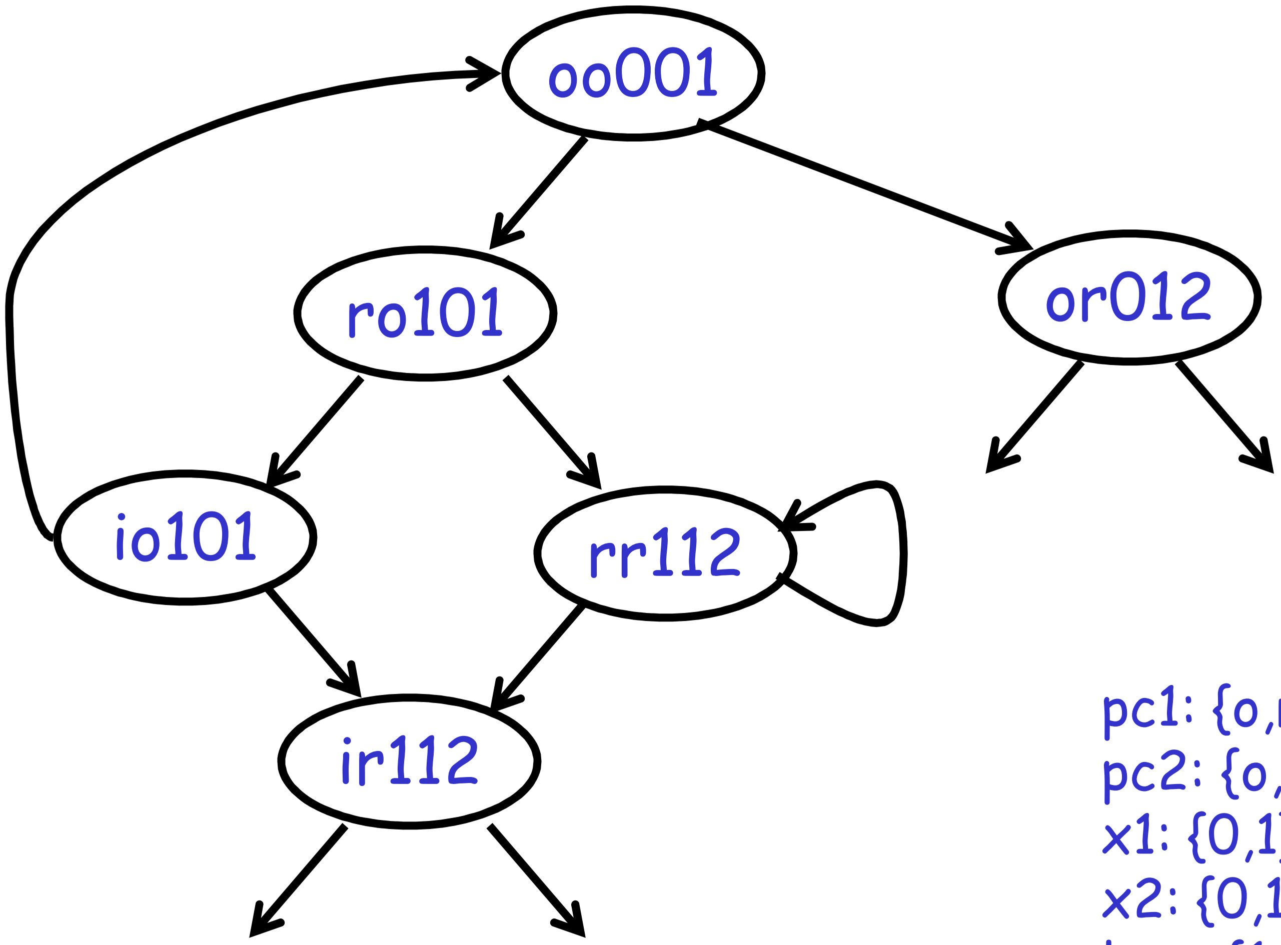
State-transition graph

Q	set of states	$\{q_1, q_2, q_3\}$
A	set of atomic observations	$\{a, b\}$
$\rightarrow \subseteq Q \times Q$	transition relation	$q_1 \rightarrow q_2$
$[\ ]: Q \rightarrow 2^A$	observation function	$[q_1] = \{a\}$

Mutual-exclusion protocol

```
loop                                              ||  loop
   out:  x1 := 1; last := 1                              out:  x2 := 1; last := 2
   req:  await  x2 = 0  or  last = 2                     req:  await  x1 = 0  or  last = 1
   in:     x1 := 0                                       in:     x2 := 0
end loop.                                             end loop.

             P1                                                    P2
```



pc1: {o,r,i}
pc2: {o,r,i}
x1: {0,1}
x2: {0,1}
last: {1,2}

$3 \cdot 3 \cdot 2 \cdot 2 \cdot 2 = 72$ states

The translation from a system description to a state-transition graph usually involves an exponential blow-up !!!

e.g., n boolean variables $\Rightarrow$ 2^n states

This is called the "state-explosion problem."

State-transition graphs are not necessarily finite-state, but they don't handle well:

-recursion (need pushdown models)

-environment interaction (need game models)

-process creation (need dynamic models)

-real time (need clock models)

-probabilistic choice (need stochastic models)

Model-checking problem

$$\begin{tikzcd}
I & \models & S \\
\text{system model} \arrow[u] & & \text{system property} \arrow[u] \\
& \text{satisfaction relation} \arrow[uu] &
\end{tikzcd}$$

Three important decisions when choosing system properties:

1 operational vs. declarative:
automata vs. logic

2 may vs. must:
branching vs. linear time

3 prohibiting bad vs. desiring good behavior:
safety vs. liveness

Three important decisions when choosing system properties:

1 operational vs. declarative:
automata vs. logic

2 may vs. must:
branching vs. linear time

3 prohibiting bad vs. desiring good behavior:
safety vs. liveness

The three decisions are orthogonal, and they lead to substantially different model-checking problems.

Safety vs. liveness

Safety:	something "bad" will never happen
Liveness:	something "good" will happen (but we don't know when)

Safety vs. liveness for sequential programs

Safety:	the program will never produce a wrong result ("partial correctness")
Liveness:	the program will produce a result ("termination")

Safety vs. liveness for sequential programs

induction on control flow

Safety: the program will never produce a wrong result ("partial correctness")

Liveness: the program will produce a result ("termination")

well-founded induction on data

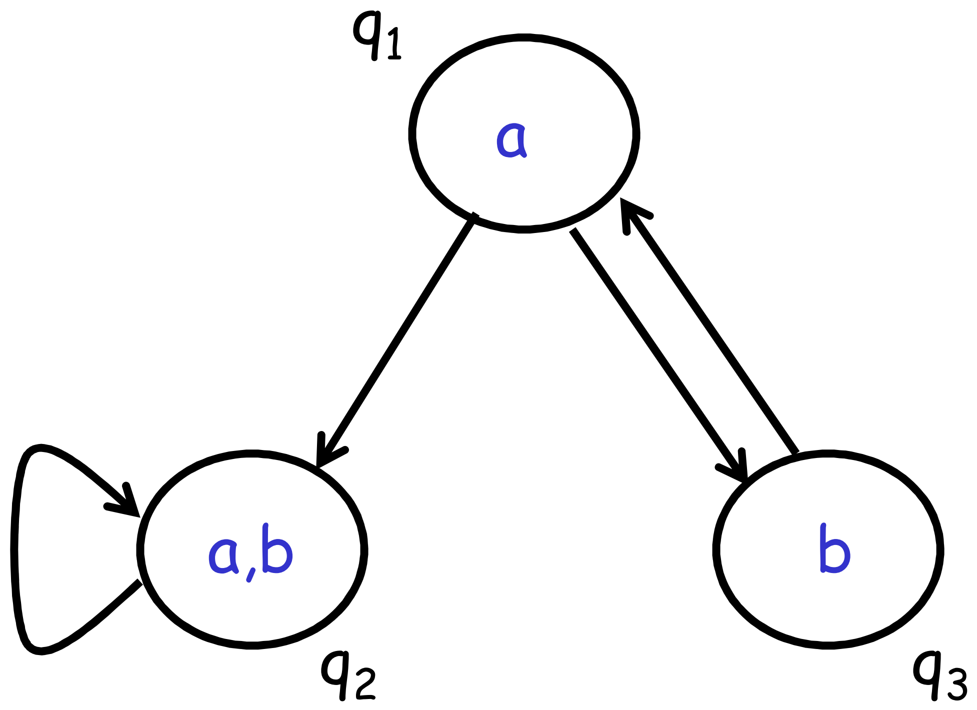
Safety vs. liveness for state-transition graphs

Safety: those properties whose violation always has a finite witness

("if something bad happens on an infinite run, then it happens already on some finite prefix")

Liveness: those properties whose violation never has a finite witness

("no matter what happens along a finite run, something good could still happen later")



Run: $q_1 \to q_3 \to q_1 \to q_3 \to q_1 \to q_2 \to q_2 \to$

Trace: $a \to b \to a \to b \to a \to a,b \to a,b \to$

State-transition graph $S = (Q, A, \rightarrow, [\,])$

Finite runs:	$\mathrm{finRuns}(S) \subseteq Q^*$
Infinite runs:	$\mathrm{infRuns}(S) \subseteq Q^\omega$
Finite traces:	$\mathrm{finTraces}(S) \subseteq (2^A)^*$
Infinite traces:	$\mathrm{infTraces}(S) \subseteq (2^A)^\omega$

Safety: the properties that can be checked on finRuns

Liveness: the properties that cannot be checked on finRuns

This is much easier.

$\swarrow$

Safety: the properties that can be checked on finRuns

Liveness: the properties that cannot be checked on finRuns

(they need to be checked on infRuns)

Example: Mutual exclusion

It cannot happen that both processes are in their critical sections simultaneously.

Example: Mutual exclusion

It cannot happen that both processes are in their critical sections simultaneously.

Safety

Example: Bounded overtaking

Whenever process P1 wants to enter the critical section, then process P2 gets to enter at most once before process P1 gets to enter.

Example: Bounded overtaking

Whenever process P1 wants to enter the critical section, then process P2 gets to enter at most once before process P1 gets to enter.

Safety

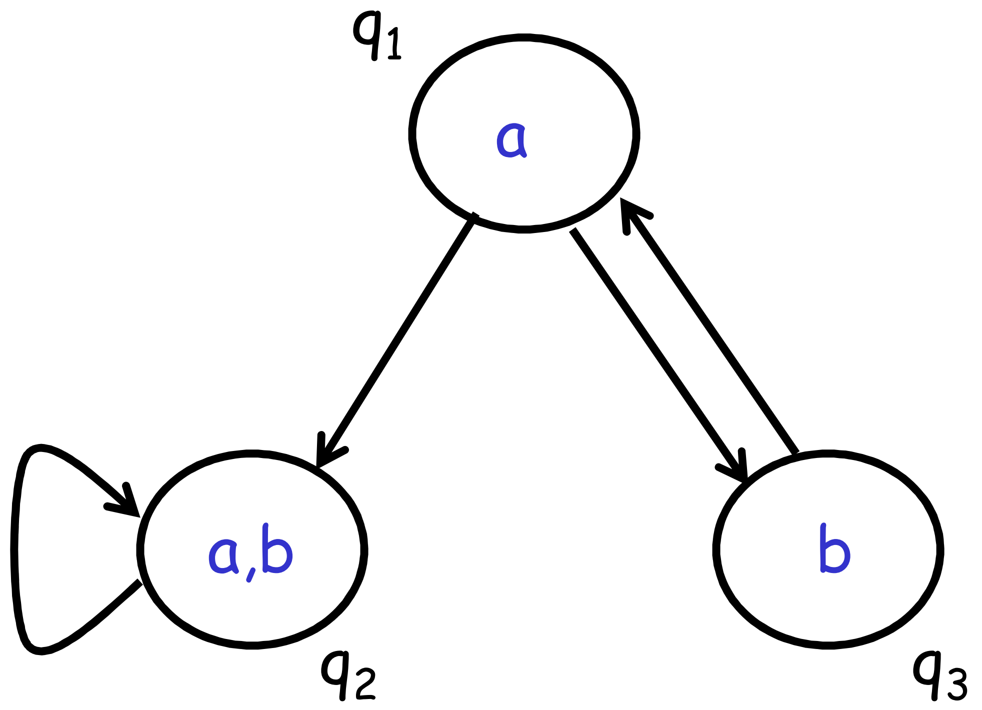
Example: Starvation freedom

Whenever process P1 wants to enter the critical section, provided process P2 never stays in the critical section forever, P1 gets to enter eventually.

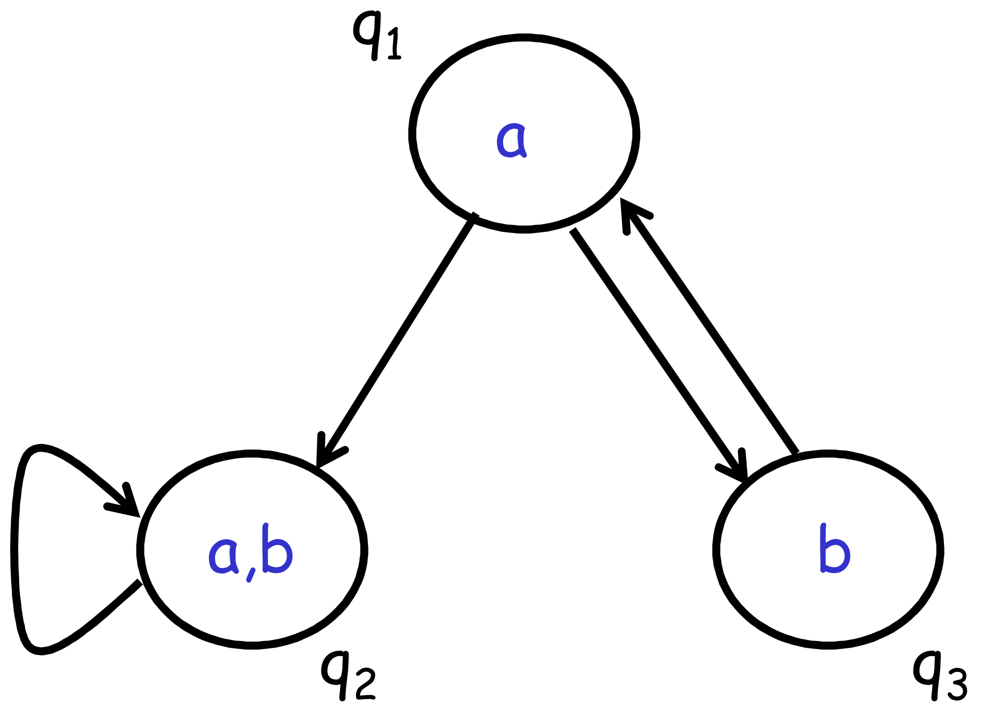
Example: Starvation freedom

Whenever process P1 wants to enter the critical section, provided process P2 never stays in the critical section forever, P1 gets to enter eventually.

Liveness



infRuns $\Rightarrow$ finRuns



infRuns $\Rightarrow$ finRuns

$\Leftarrow^*$

closure

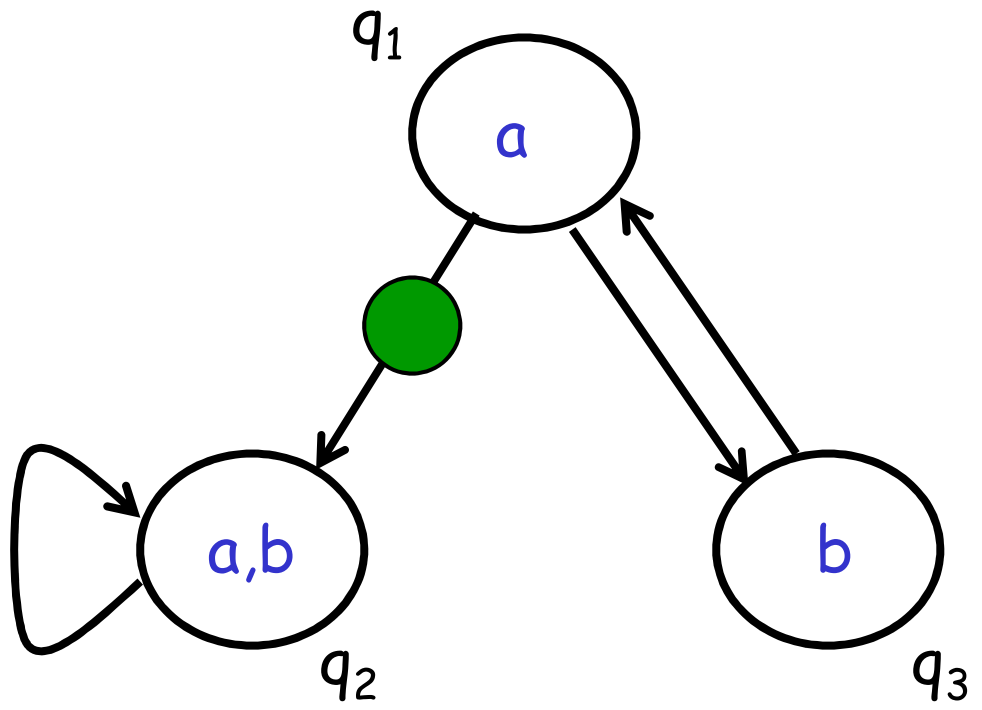
*finite branching

For state-transition graphs,
all properties are safety properties !

Example: Starvation freedom

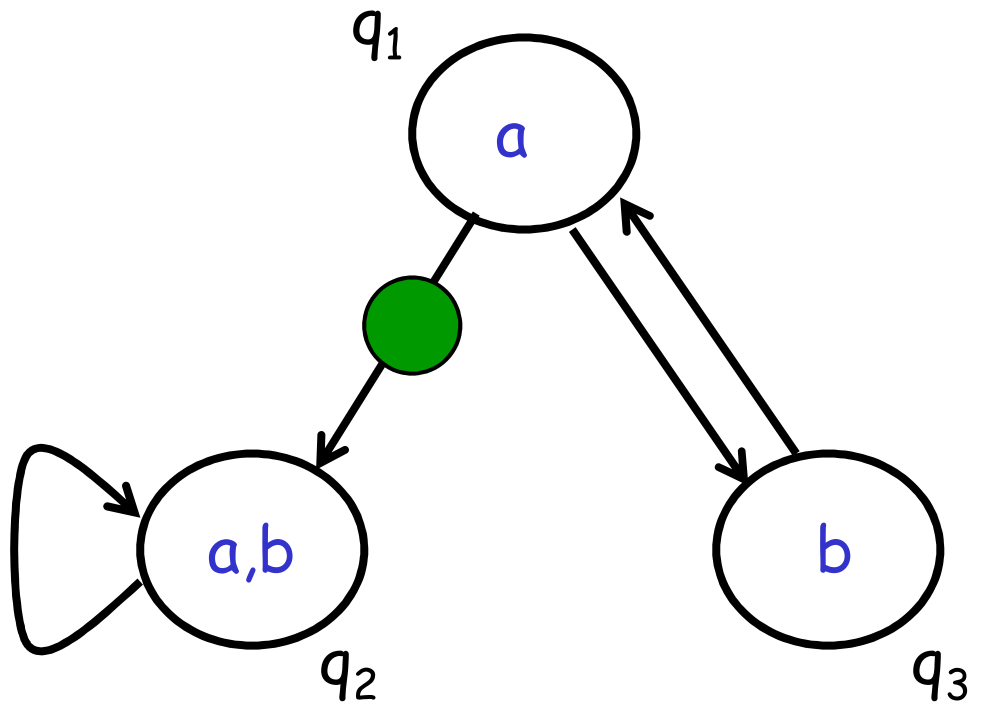
Whenever process P1 wants to enter the critical section, provided process P2 never stays in the critical section forever, P1 gets to enter eventually.

Liveness



Fairness constraint:

the green transition cannot be ignored forever



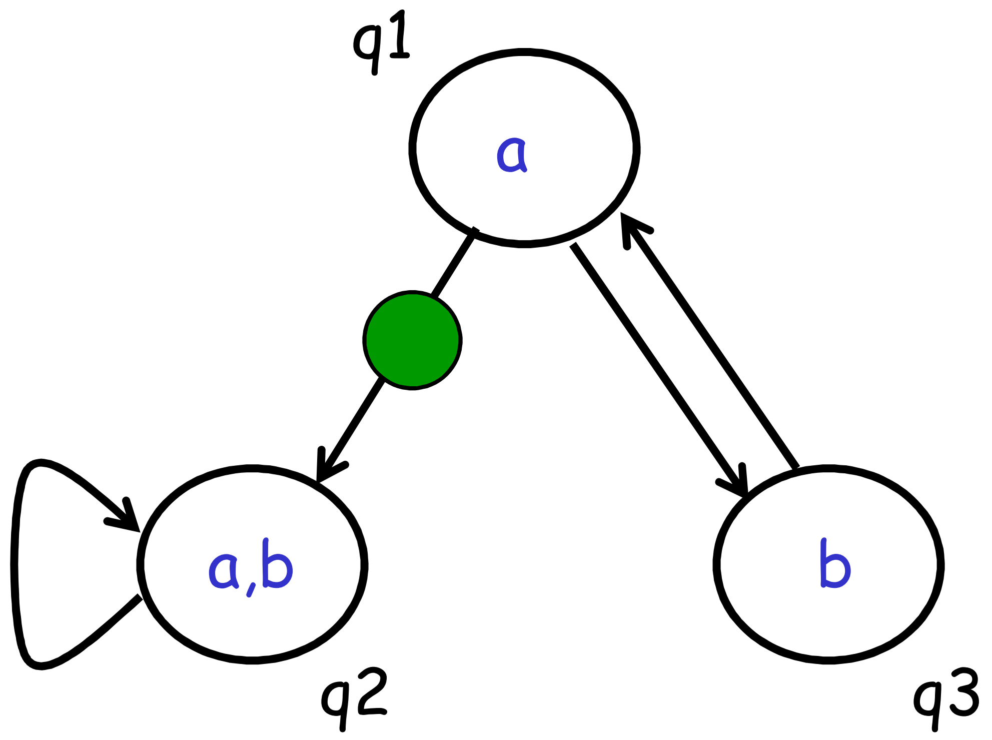
Without fairness: infRuns = $q_1\ (q_3\ q_1)^*\ q_2^{\omega} \cup (q_1\ q_3)^{\omega}$

With fairness: infRuns = $q_1\ (q_3\ q_1)^*\ q_2^{\omega}$

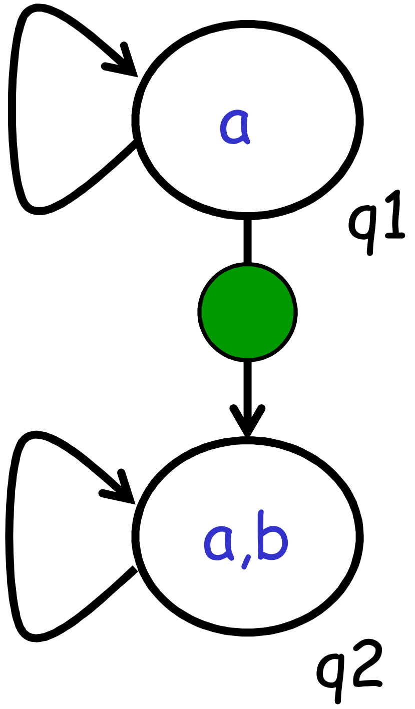
Two important types of fairness

1 Weak (Buchi) fairness:
a specified set of transitions cannot be enabled forever without being taken

2 Strong (Streett) fairness:
a specified set of transitions cannot be enabled infinitely often without being taken



Strong fairness



Weak fairness

Weak fairness is sufficient for asynchronous models ("no process waits forever if it can move").

Strong fairness is necessary for modeling synchronous interaction (rendezvous).

Weak fairness is sufficient for
asynchronous models
(“no process waits forever if it can move”).

Strong fairness is necessary for modeling
synchronous interaction (rendezvous).

Strong fairness makes model checking
more difficult.

Fairness changes only infRuns, not finRuns.

$$\Downarrow$$

Fairness can be ignored for checking safety properties.

Two remarks

The vast majority of properties to be verified are safety.

While nobody will ever observe the violation of a liveness property, fairness is a useful abstraction that turns complicated safety into simple liveness.

Three important decisions when choosing system properties:

1 operational vs. declarative:
automata vs. logic

2 may vs. must:
branching vs. linear time

✓ 3 prohibiting bad vs. desiring good behavior:
safety vs. liveness

The three decisions are orthogonal, and they lead to substantially different model-checking problems.

Branching vs. linear time

Branching time: something may (or may not) happen
(e.g., every req may be followed by grant)

Linear time: something must (or must not) happen
(e.g., every req must be followed by grant)

Branching vs. linear time

Branching time: something may (or may not) happen
(e.g., every req may be followed by grant)

Linear time: something must (or must not) happen
(e.g., every req must be followed by grant)

$\uparrow$

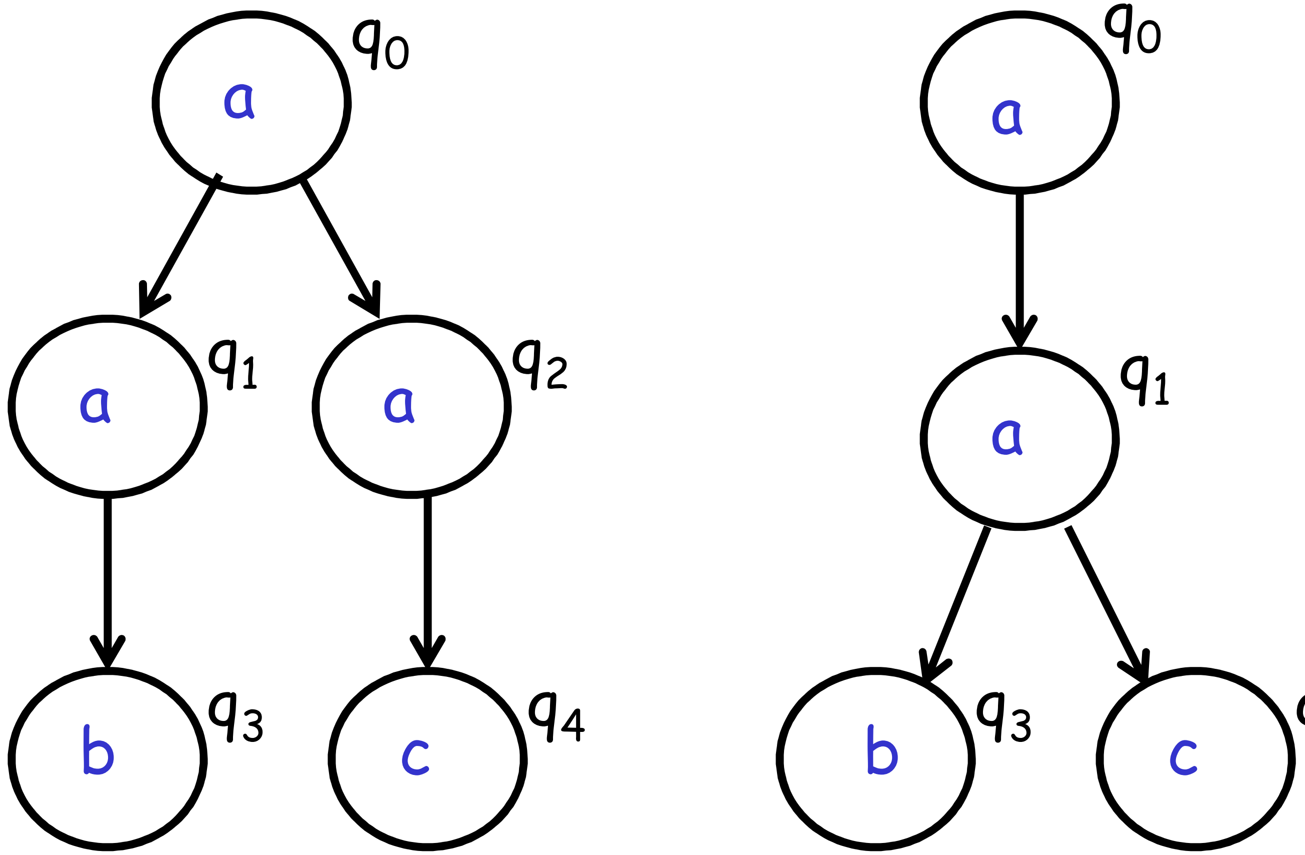
holds for all traces

Linear time: the properties that can be checked on infTraces

Branching time: the properties that cannot be checked on infTraces

↑

may refer to states



Same traces $\{aab, aac\}$

Different runs (trees) $\{q_0\, q_1\, q_3,\ q_0\, q_2\, q_4\}$, $\{q_0\, q_1\, q_3,\ q_0\, q_1\, q_4\}$

Observation a may occur.

Observation a may occur.

||

It is not the case that a must not occur.

Linear

We may reach an a from which we must not reach a b .

We may reach an a from which we must not reach a b .

Branching

One is rarely interested in may properties,

but certain may properties are easier to model check, and they imply interesting must properties.

(This is because when checking must properties, we sometimes have to "guess" states.)

	Linear	Branching
Safety	finTraces	finRuns
Liveness	infTraces	infRuns

Three important decisions when choosing system properties:

1 operational vs. declarative:
automata vs. logic

✓ 2 may vs. must:
branching vs. linear time

✓ 3 prohibiting bad vs. desiring good behavior:
safety vs. liveness

The three decisions are orthogonal, and they lead to substantially different model-checking problems.

Logics

Branching time

CTL
(Computation Tree Logic)

Linear time

LTL

Defining a logic

1. Syntax:

   What are the formulas?

2. Semantics:

   What are the models?

   Does model M satisfy formula φ ? $M \models \varphi$

CTL Syntax

$$\varphi \; ::= \; a \mid \varphi \wedge \varphi \mid \neg \varphi \mid \overset{\text{EX}}{\exists\bigcirc \varphi} \mid \overset{\text{EU}}{\varphi \, \exists\mathcal{U} \, \varphi} \mid \overset{\text{EG}}{\exists\Box\varphi}$$

boolean operators

boolean variable
(atomic observation)

modal operators

CTL Model

$$\begin{tikzcd} & (\, K, & q \,) & \\ \text{fair state-transition graph} \arrow[ur] & & & \text{state of K} \arrow[ul] \end{tikzcd}$$

CTL Semantics

$(K,q) \models \exists\Box\, \varphi$ iff exist $q_0, q_1, \ldots$ s.t.

1. $q = q_0 \rightarrow q_1 \rightarrow \ldots$ is an infinite fair run
2. for all $i \geq 0$, $(K,q_i) \models \varphi$

Defined modalities

$9\Diamond\ \varphi\ =\ \text{true}\ 9U\ \varphi$	EF	exists eventually
$8\Box\ \varphi\ =\ \neg\ 9\Diamond\ \neg\varphi$	*AG*	forall always

Example: mutex protocol

Mutual exclusion

$\forall\Box\ \neg\ (pc1=in \wedge pc2=in)$

Bounded overtaking

$\forall\Box\ (\ pc1=req \ \Rightarrow\ (pc2\neq in)\ \forall W\ (pc2=in)\ \forall W\ (pc2\neq in)\ \forall W\ (pc1=in))$

Starvation freedom

$\forall\Box\ (pc1=req \ \Rightarrow\ \forall\Diamond\ (pc1=in))$

If only universal properties are of interest,
why not omit the path quantifiers?

LTL Syntax

$$\varphi \;::=\; a \mid \varphi \wedge \varphi \mid \neg\, \varphi \mid \bigcirc\, \varphi \mid \varphi\, \mathsf{U}\, \varphi$$

LTL Model

infinite trace $t = t_0\ t_1\ t_2 \ldots$
(sequence of observations)

Language of deadlock-free state-transition graph K at state q :

L(K,q) ... set of infinite traces of K starting at q

$(K,q) \models^{\forall} \varphi$ iff for all $t \in L(K,q)$, $t \models \varphi$

$(K,q) \models^{\exists} \varphi$ iff exists $t \in L(K,q)$, $t \models \varphi$

LTL Semantics

$t \models a$	iff	$a \in t_0$
$t \models \varphi \wedge \psi$	iff	$t \models \varphi$ and $t \models \psi$
$t \models \neg\varphi$	iff	not $t \models \varphi$
$t \models \bigcirc\, \varphi$	iff	$t_1\, t_2 \ldots \models \varphi$
$t \models \varphi\ \mathrm{U}\ \psi$	iff	exists $n \geq 0$ s.t. 1. for all $0 \leq i < n$, $t_i\, t_{i+1} \ldots \models \varphi$ 2. $t_n\, t_{n+1} \ldots \models \psi$

Defined modalities

$\bigcirc$	X	next
U	U	until
$\Diamond\varphi = \text{true}\ U\ \varphi$	F	eventually
$\Box\varphi = \neg\Diamond\neg\varphi$	G	always
$\varphi\ W \psi = (\varphi\ U\ \psi) \vee \Box\varphi$	W	waiting-for

Important properties

Invariance	$\Box\, a$	safety
	$\Box\, \neg\, (pc1{=}in \wedge pc2{=}in)$	
Sequencing	$a \; W \; b \; W \; c \; W \; d$	safety
	$\Box\, (pc1{=}req \;\Rightarrow\; (pc2{\neq}in) \; W \; (pc2{=}in) \; W \; (pc2{\neq}in) \; W \; (pc1{=}in))$	
Response	$\Box\, (a \;\Rightarrow\; \Diamond\, b)$	liveness
	$\Box\, (pc1{=}req \;\Rightarrow\; \Diamond\, (pc1{=}in))$	

Composed modalities

$\Box\Diamond\, a$	infinitely often a
$\Diamond\Box\, a$	almost always a

Where did fairness go ?

Unlike in CTL, fairness can be expressed in LTL !

So there is no need for fairness in the model.

Weak (Buchi) fairness :

$\neg \Diamond\Box\, (\text{enabled} \wedge \neg\, \text{taken}) =$

$\Box\Diamond\, (\text{enabled} \Rightarrow \text{taken})$

Strong (Streett) fairness :

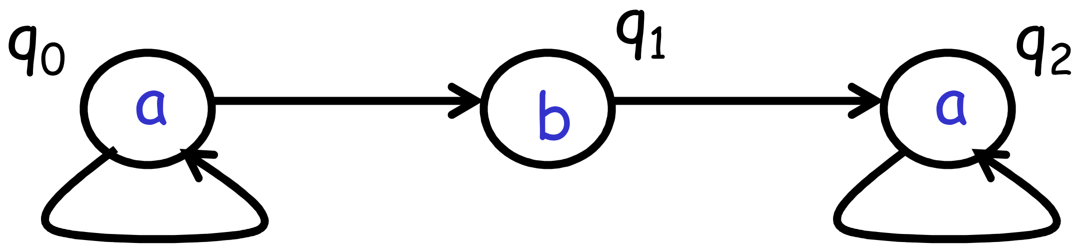
$(\Box\Diamond\, \text{enabled}) \Rightarrow (\Box\Diamond\, \text{taken})$

Starvation freedom

$$\Box\Diamond\,(pc2=in \Rightarrow \bigcirc\,(pc2=out)) \Rightarrow$$
$$\Box\,(pc1=req \Rightarrow \Diamond\,(pc1=in))$$

CTL cannot express fairness

$\exists \Box \Diamond b \quad \neq \quad \exists \Box \, \exists \Diamond b$


$$\begin{tikzcd} \overset{q_0}{a} \arrow[loop below] \arrow[r] & \overset{q_1}{b} \arrow[r] & \overset{q_2}{a} \arrow[loop below] \end{tikzcd}$$

LTL cannot express branching

Possibility $\forall\Box\,(a \Rightarrow \exists\Diamond\, b)$

So, LTL and CTL are incomparable.

(There are branching logics that can express fairness, e.g., $\mathrm{CTL}^* = \mathrm{CTL} + \mathrm{LTL}$, but they lose the computational attractiveness of CTL.)

System property: 2x2x2 choices

-safety (finite runs) vs. liveness (infinite runs)

-linear time (traces) vs. branching time (runs)

-logic (declarative) vs. automata (operational)

Automata

Safety:	finite automata
Liveness:	omega automata

Linear:	language containment
Branching:	simulation

Specification Automata

Syntax, given a set A of atomic observations:

S	finite set of states
$S_0 \subseteq S$	set of initial states
$\rightarrow \subseteq S \times S$	transition relation
$\phi: S \rightarrow PL(A)$	where the formulas of PL are
	$\varphi \ ::= \ a \mid \varphi \wedge \varphi \mid \neg \varphi$
	for $a \in A$

Language L(M) of specification automaton

$M = (S, S_0, \rightarrow, \phi)$:

finite trace $t_0, ..., t_n \in L(M)$

iff

there exists a finite run $s_0 \rightarrow s_1 \rightarrow ... \rightarrow s_n$ of M

such that

for all $0 \leq i \leq n$, $t_i \models \phi(s_i)$

Linear semantics of specification automata:

language containment

$(K,q) \models^{S} M \quad \text{iff} \quad L(K,q) \subseteq L(M)$

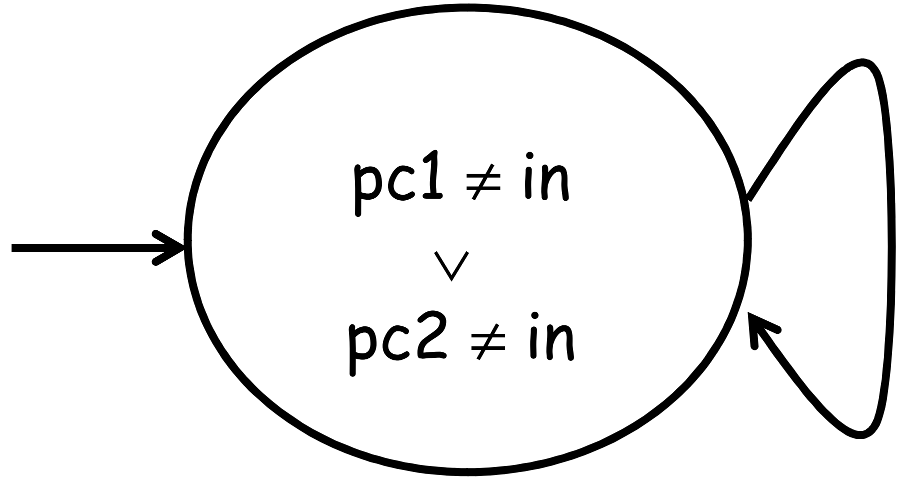
state-transition graph

state of K

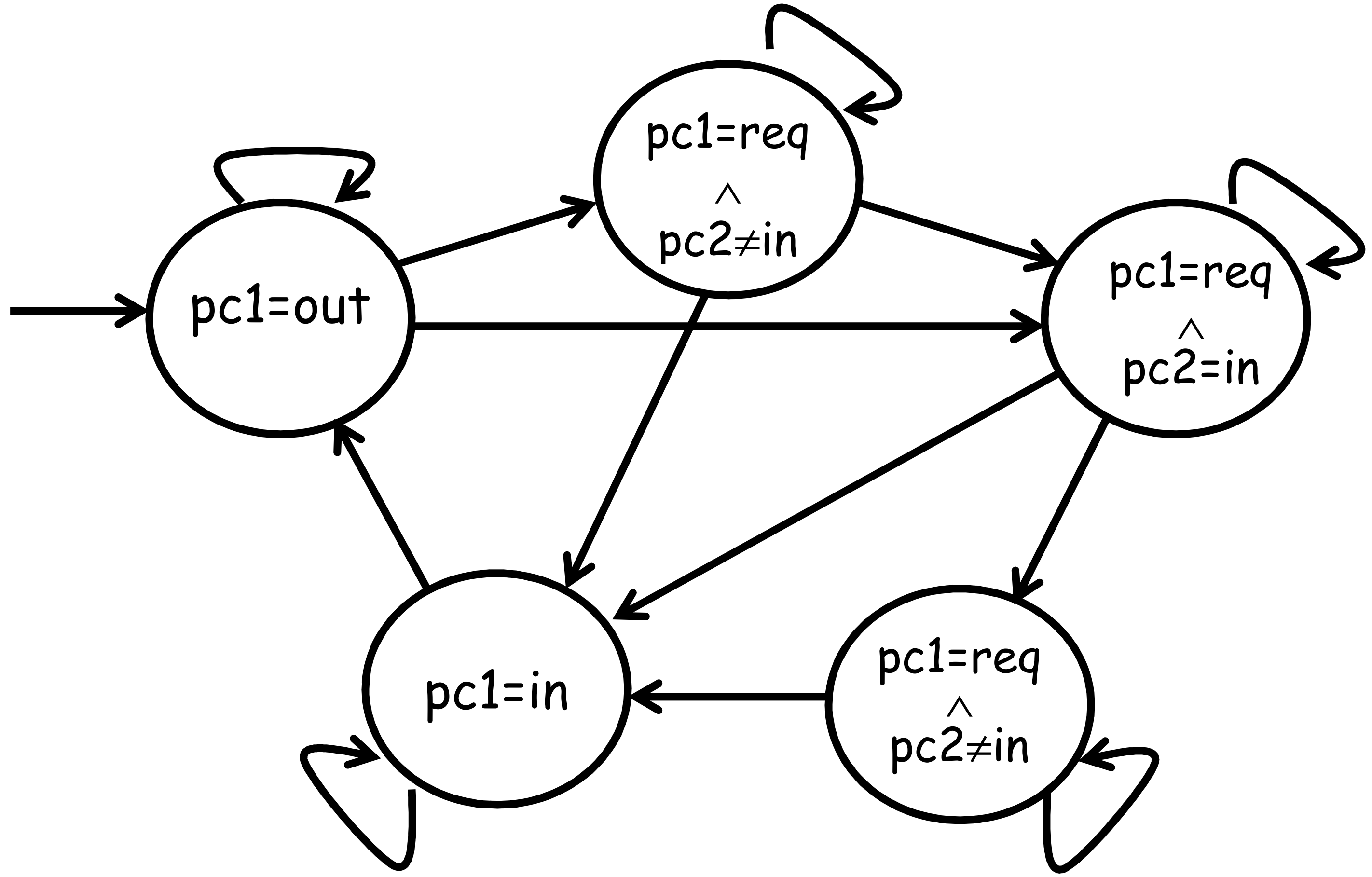
specification automaton

finite traces

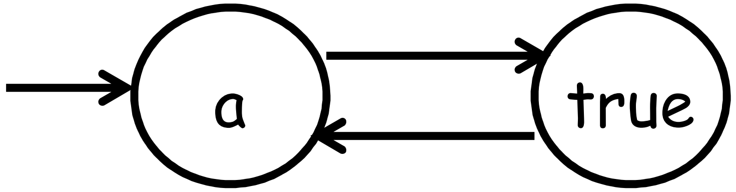
Invariance specification automaton



One-bounded overtaking specification automaton



Automata are more expressive than logic, because temporal logic cannot count :



This cannot be expressed in LTL.

(How about $a \wedge \Box\, (a \Rightarrow \bigcirc\bigcirc\, a)$?)

Checking language containment between finite automata is PSPACE-complete !

$$L(K,q) \subseteq L(M)$$

iff

$$L(K,q) \cap \text{complement}(\, L(M)\,) = \emptyset$$

involves determinization
(subset construction)

In practice:

1. require deterministic specification automata
2. use monitor automata
3. use branching semantics

Monitor Automata

Syntax:

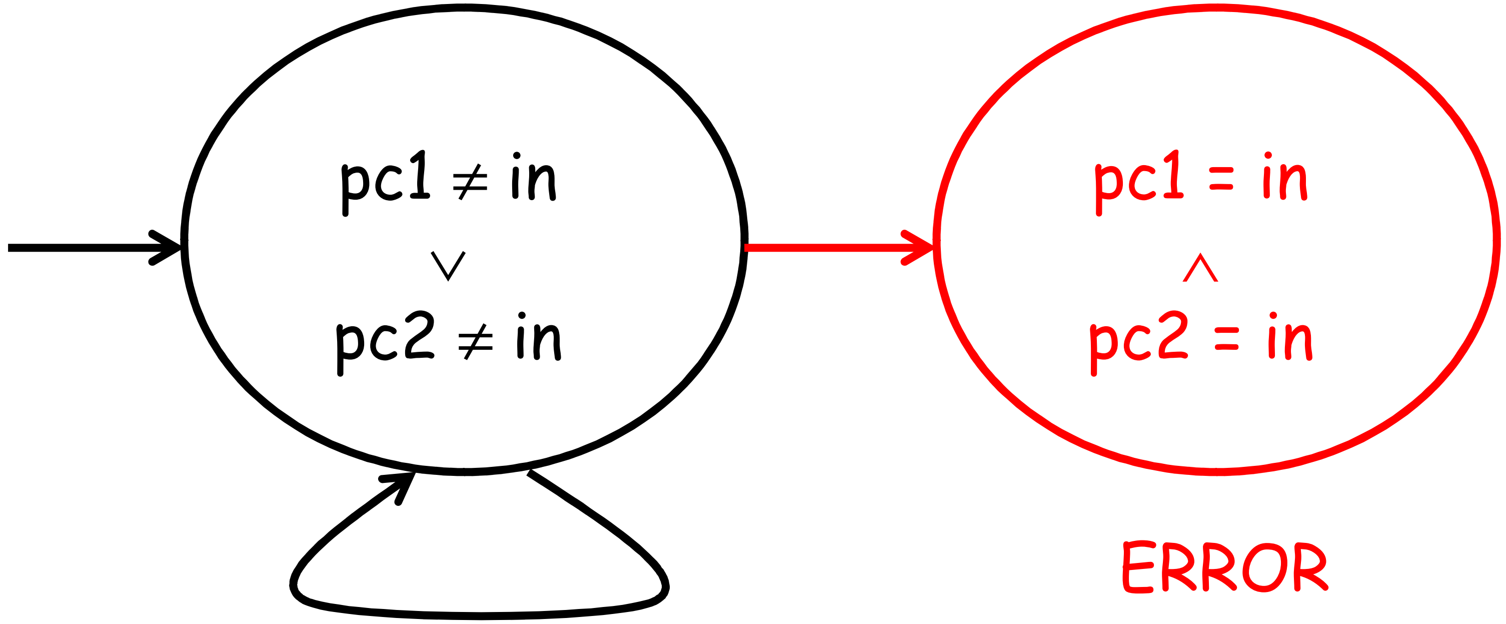
same as specification automata,
except also set $E \subseteq S$ of error states

Semantics:

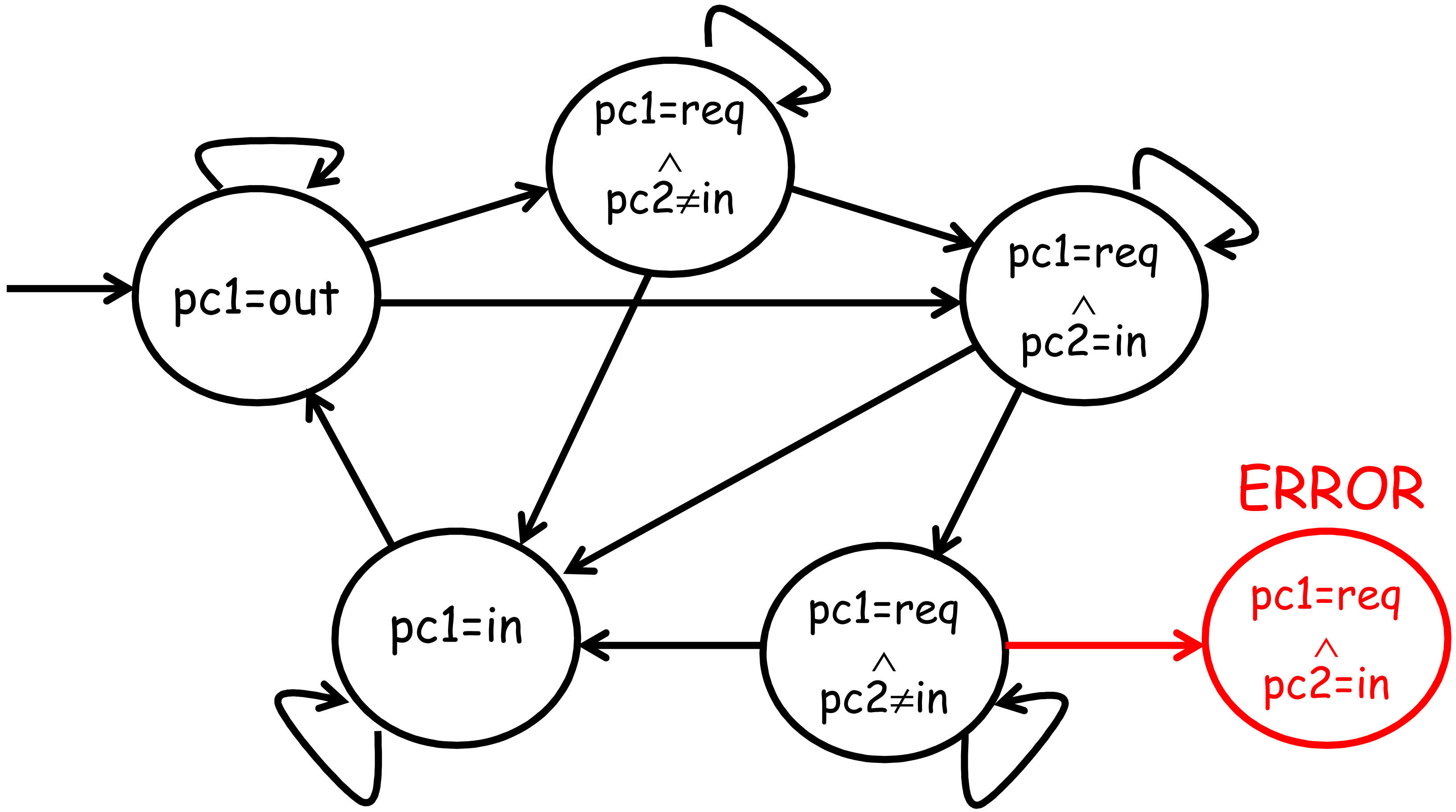
define $L(M)$ s.t. runs must end in error states

$(K,q) \models^M M$ iff $L(K,q) \cap L(M) = \emptyset$

Invariance monitor automaton



One-bounded overtaking monitor automaton



Specification automaton | Monitor automaton

$$\begin{tikzcd} M \arrow[r, shift left] & \text{complement}(M) \arrow[l, shift left] \end{tikzcd}$$

Specification automaton	Monitor automaton
-describe correct traces	-describe error traces
-check language containment (exponential)	-check emptiness (linear): reachability of error states

"All safety verification is reachability checking."

Exercise

1. Draw a state-transition graph that generates the Roman numerals
2. Define the property “there are no more than 3 adjacent I” using
   a. LTL
   b. a specification automaton
   c. a monitor automaton
3. Use one of the three specifications to model check the property (show all intermediate steps)