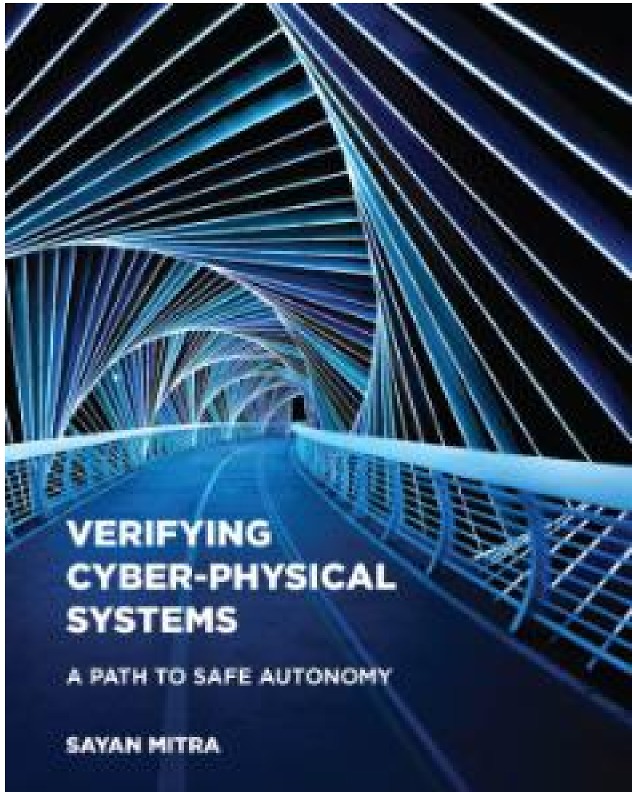
Lecture 2: Modeling Computation

Huan Zhang

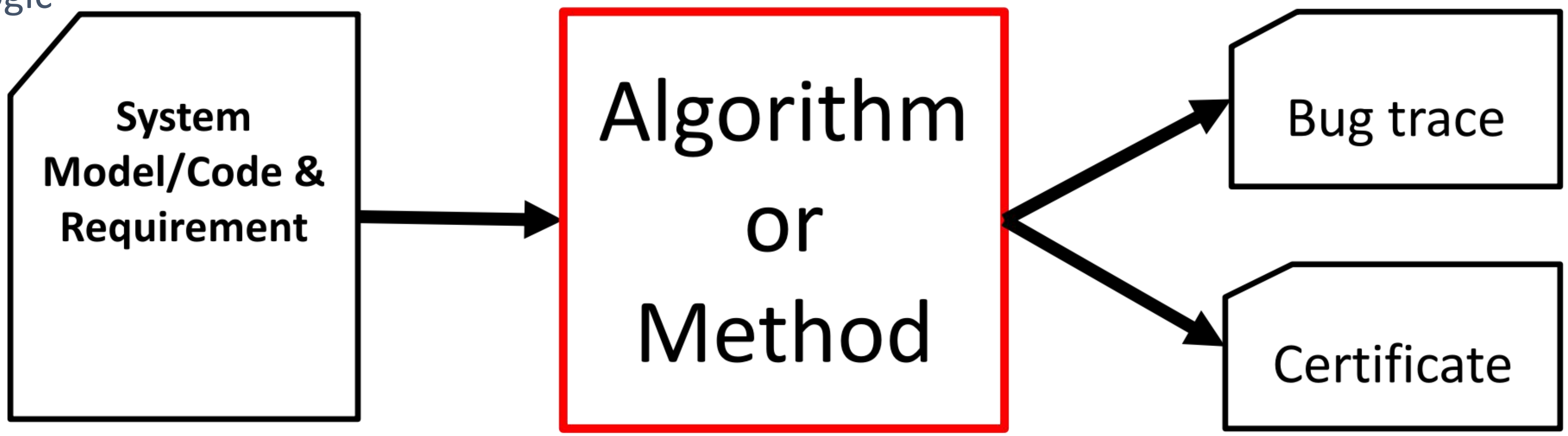
huan@huan-zhang.com



Slides adapted from Prof. Sayan Mitra's slides in Fall 2021

Review: The verification problem: model + requirement + algorithm

Model: boolean logic



Counterexamples

Proofs!

Requirement: check if the two functions return the same integer

Verification algorithm to solve boolean satisfiability (DPLL, CDCL)

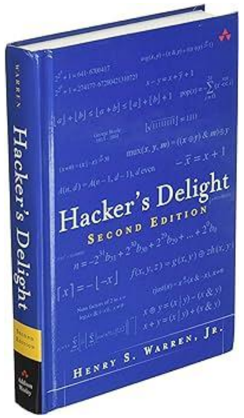
Review: Formal Verification Example

Formal verification aims to prove that **for all possible inputs,** the results of the two functions are formally the same (mathematically, the same integer is returned)

Naive implementation **==** Clever implementation

```
int popcount(uint32_t x) {
  int c = 0;
  for (int i = 0; i < 32; i++) {
    c += x & 1;
    x >>= 1;
  }
  return c;
}
```

```
int popcount (uint32_t x) {
    x = x - ((x >> 1) & 0x55555555);
    x = (x & 0x33333333) + ((x >> 2) & 0x33333333);
    x = ((x + (x >> 4) & 0xf0f0f0f) * 0x1010101) >> 24;
    return x;
}
```



Example source: Marijn J.H. Heule, "SAT and SMT Solvers in Practice"

Works for this simple program but more complex ones won't work (e.g., think about a loop without a constant number of iterations)

```c
int popcount(uint32_t x) {
  int c = 0;
  for (int i = 0; i < 32; i++) {
    c += x & 1;
    x >>= 1;
  }
  return c;
}
```

```
(define-fun slow ((x (_ BitVec 32))) (_ BitVec 32)
 (bvadd
   (ite (= #b1 ((_ extract  0  0) x)) #x00000001 #x00000000)
   (ite (= #b1 ((_ extract  1  1) x)) #x00000001 #x00000000)
   (ite (= #b1 ((_ extract  2  2) x)) #x00000001 #x00000000)
                      ...
   (ite (= #b1 ((_ extract 30 30) x)) #x00000001 #x00000000)
   (ite (= #b1 ((_ extract 31 31) x)) #x00000001 #x00000000)))
```

```c
int popcount (uint32_t x) {
    x = x - ((x >> 1) & 0x55555555);
    x = (x & 0x33333333) + ((x >> 2) & 
0x33333333);
    x = ((x + (x >> 4) & 0xf0f0f0f) * 0x1010101)
>> 24;
    return x;
}
```

```
(define-fun line1 ((x (_ BitVec 32))) (_ BitVec 32)
  (bvsub x (bvand (bvlshr x #x00000001) #x55555555)))

(define-fun line2 ((x (_ BitVec 32))) (_ BitVec 32)
  (bvadd (bvand x #x33333333)
         (bvand (bvlshr x #x00000002) #x33333333)))

(define-fun line3 ((x (_ BitVec 32))) (_ BitVec 32)
  (bvlshr (bvmul (bvand (bvadd (bvlshr x #x00000004)
         x) #x0f0f0f0f) #x01010101) #x00000018))

(define-fun fast ((x (_ BitVec 32))) (_ BitVec 32)
   (line3 (line2 (line1 x))))
```

Outline for this class

Goal of this course: model anything!

This lecture: model **computations**

Today: **Automaton** as a model for computations (e.g., your program)

More in the rest of the class: model physical process (e.g., motors), model machine learning models (e.g., neural nets), ...

Automata or discrete transition systems

- The “state” of a system captures all the information needed to predict the system’s future behavior
- Behavior of a system is a sequence of states
- *Our ultimate goal: write programs (verification algorithms) that prove properties about all behaviors of a system*
- “Transitions” capture how the state can change

All models are wrong, some are useful

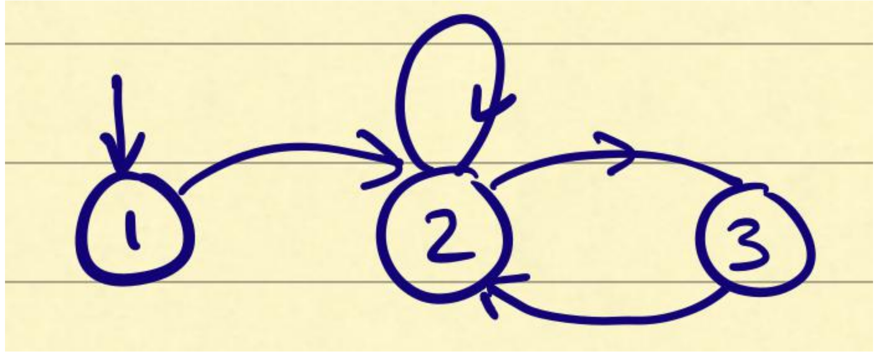
The complete state of a computing system has a lot of information

- values of program variables, network messages, position of the program counter, bits in the CPU registers, etc.
- thus, modeling requires judgment about what is important and what is not

Mathematical formalism used is called *automaton* a.k.a. *discrete* transition system

Automata or discrete transition systems

- Example: you probably know the finite state machine (FSM)
  - States: {1, 2, 3}
  - Start state: {1}
  - Transistions



- Automata is more general:
  - We define “states” implicitly using variables
  - The number of state is arbitrary

Example: Dijkstra’s mutual exclusion algorithm

Informal Description: A token-based mutual exclusion algorithm on a ring network

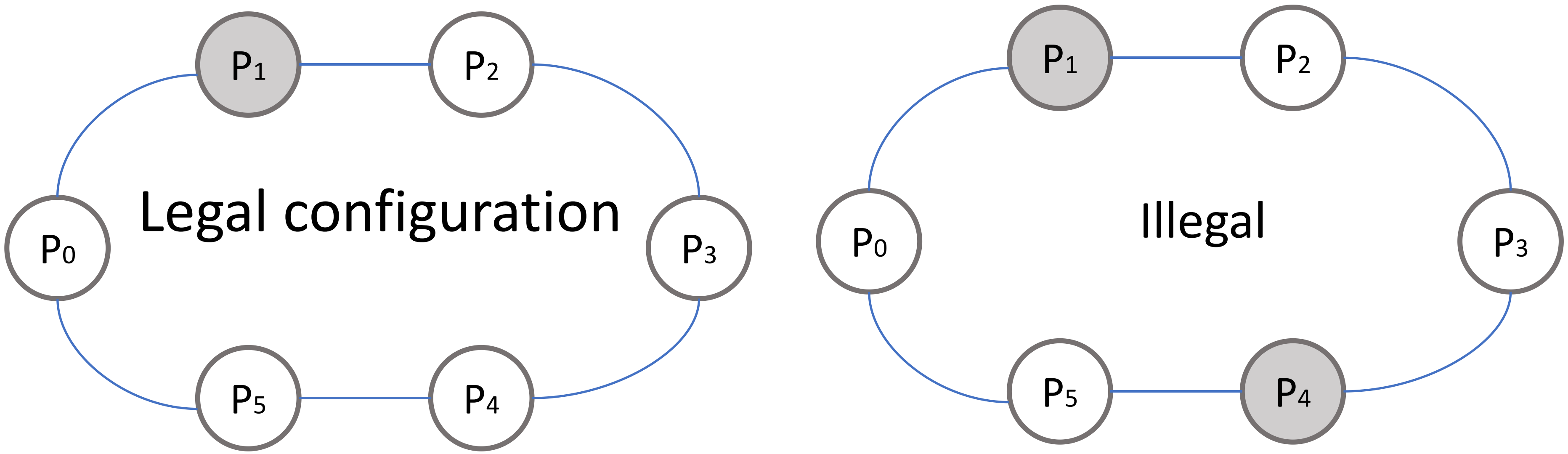
- Collection of processes that send and receive bits over a ring network so that only one of them has a “token” to access a critical resource (e.g., a shared calendar)

Discrete model

- Each process has variables that take only discrete values
- Time elapses in discrete steps

Self-stabilizing Systems in Spite of Distributed Control, CACM, 1974.

Token-based mutual exclusion in unidirectional ring

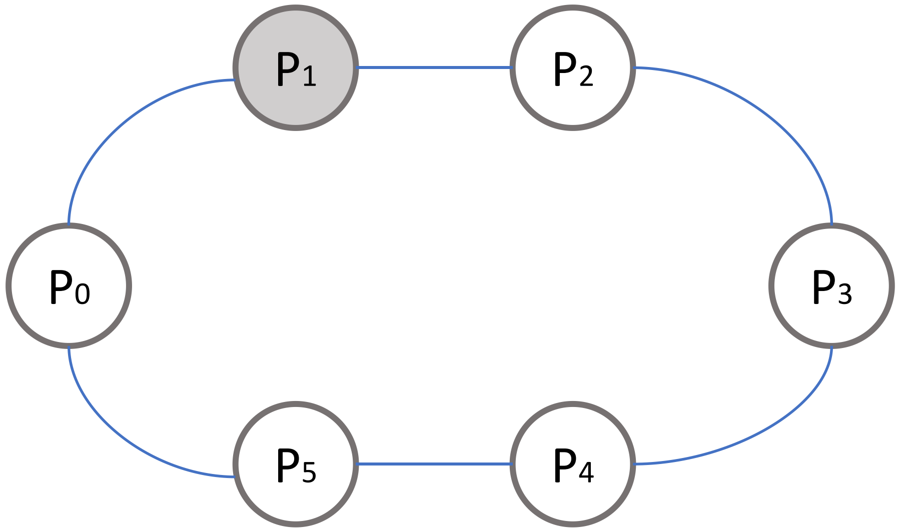


N processes with ids 0, 1, …, N-1

Unidirectional ring: each i>0 process P_i reads the state of only the predecessor P_{i-1}; P_0 reads only P_{N-1}

1. Legal configuration = exactly one "token" in the ring
2. Single token circulates in the ring
3. Even if multiple tokens arise because of faults, if the algorithm continues to work correctly, then eventually there is a single token; this is the *self stabilizing* property

Dijkstra's mutual exclusion Algorithm ['74]



N processes: 0, 1, …, N-1

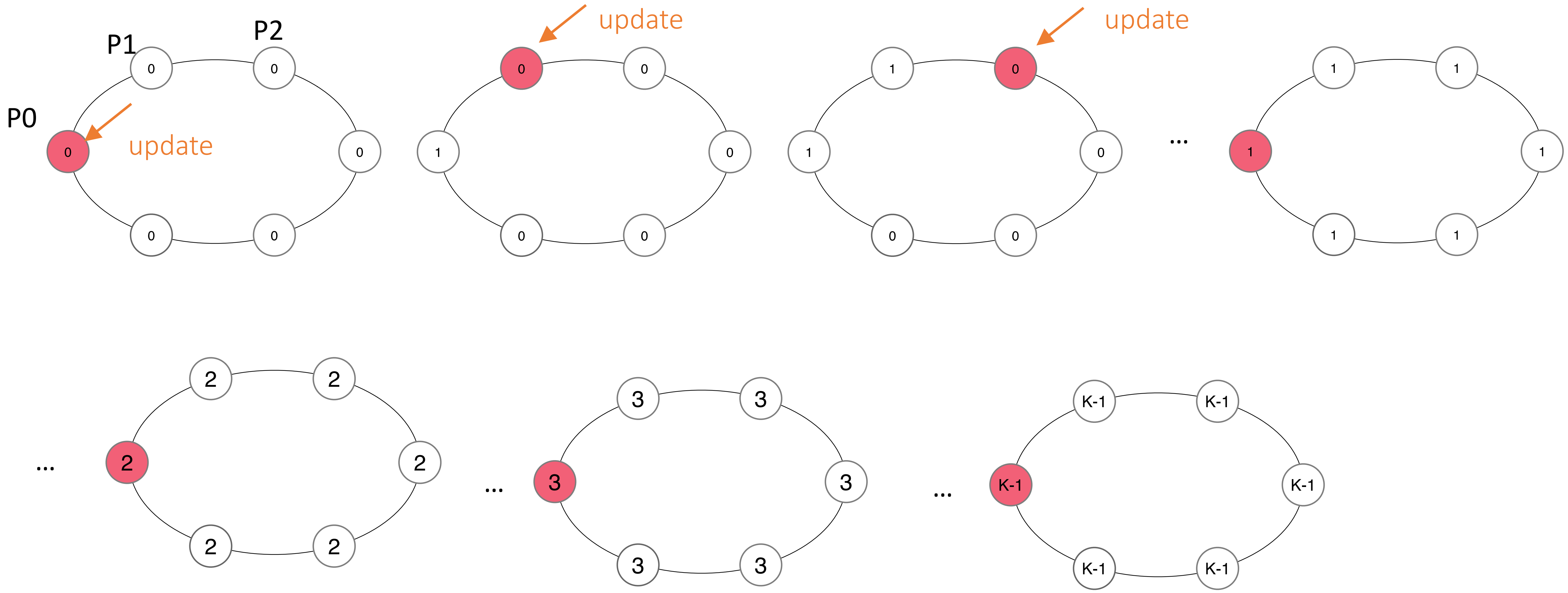
state of each process j is a single integer variable x[j] ∈ {0, 1, 2, K-1}, where K > N

p_i has TOKEN if and only if the blue conditional below is true

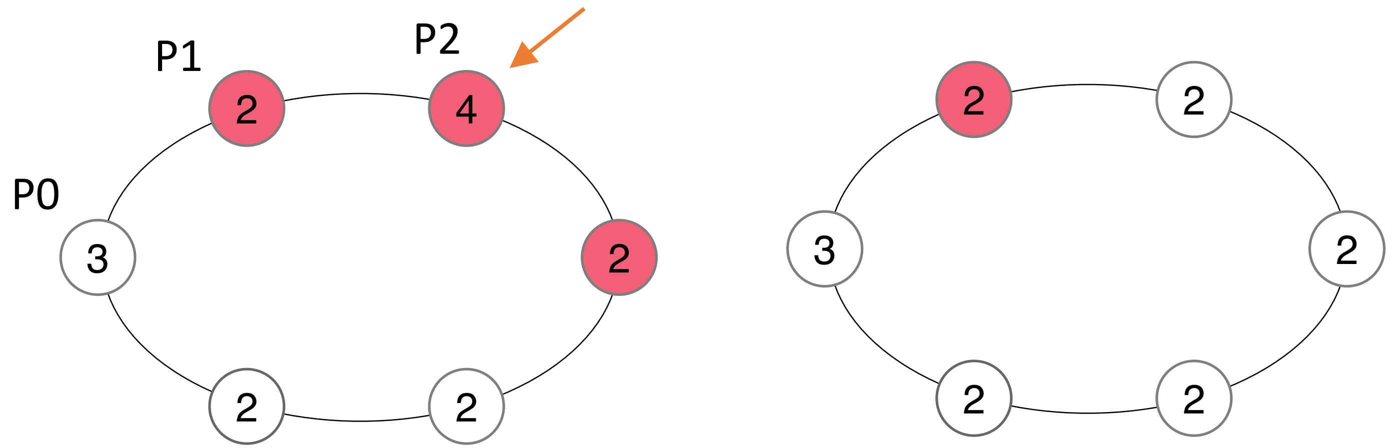
The "update" action is defined differently for P0 vs. others

P_0	if x[0] = x[N-1]	then x[0] := x[0] + 1 mod K
P_j, j > 0	if x[j] ≠ x[j -1]	then x[j] := x[j-1]

Sample executions: from a legal state (single token)

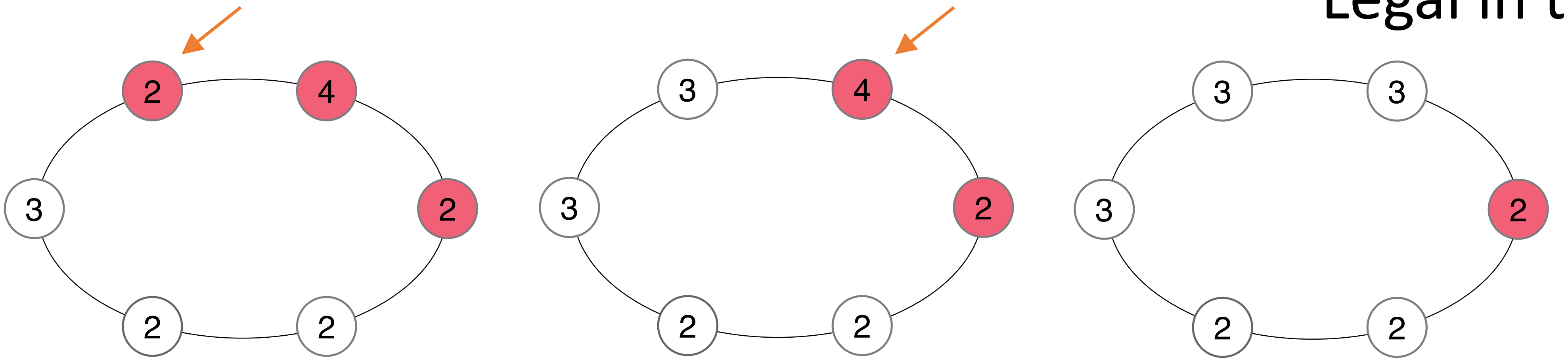


Execution from an illegal state

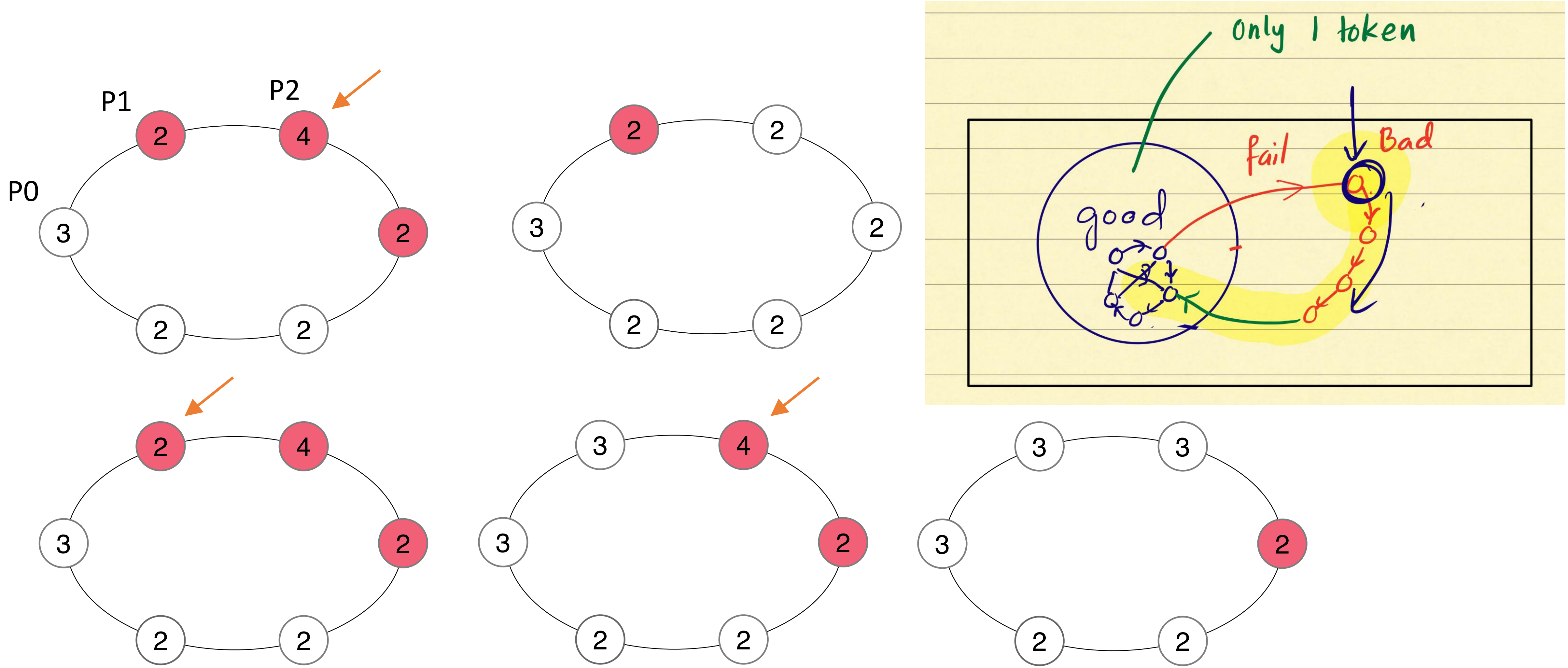


Legal in single "step"

Legal in two steps



Execution from an illegal state



Put it in a more formal way: A language for specifying automata (IOA)

```
automaton DijkstraTR(N:Nat, K:Nat), where K > N
  type ID: enumeration [0,...,N-1]
  type Val: enumeration [0,...,K-1]
  actions
    update(i:ID)
  variables
    x:[ID -> Val]
  transitions
    update(i:ID)
      pre i = 0 /\ x[i] = x[N-1]
      eff x[i] := (x[i] + 1) %  K

    update(i:ID)
      pre i >0  /\ x[i] ~= x[i-1]
      eff x[i] := x[i-1]
```

A language for specifying automata

```
automaton DijkstraTR(N:Nat, K:Nat), where K > N
  type ID: enumeration [0,...,N-1]
  type Val: enumeration [0,...,K-1]
  actions
     update(i:ID)
  variables
     x:[ID -> Val]
   transitions
     update(i:ID)
       pre i = 0 /\ x[i] = x[N-1]
       eff x[i] := (x[i] + 1) %  K

      update(i:ID)
        pre i >0  /\ x[i] ~= x[i-1]
        eff x[i] := x[i-1]
```

Name of automaton and formal parameters

symbols -> maps, /\ and, \/ or, ~= not equal, % mod

A language for specifying automata

```
automaton DijkstraTR(N:Nat, K:Nat), where K > N
  type ID: enumeration [0,...,N-1]
  type Val: enumeration [0,...,K-1]
  actions
    update(i:ID)
  variables
    x:[ID -> Val]
  transitions
    update(i:ID)
      pre i = 0 /\ x[i] = x[N-1]
      eff x[i] := (x[i] + 1) %  K

    update(i:ID)
      pre i >0  /\ x[i] ~= x[i-1]
      eff x[i] := x[i-1]
```

user defined type declarations

symbols -> maps, /\ and, \/ or, ~= not equal, % mod

A language for specifying automata

```
automaton DijkstraTR(N:Nat, K:Nat), where K > N
  type ID: enumeration [0,...,N-1]
  type Val: enumeration [0,...,K-1]
  actions
    update(i:ID)
  variables
    x:[ID -> Val]
  transitions
    update(i:ID)
      pre i = 0 /\ x[i] = x[N-1]
      eff x[i] := (x[i] + 1) %  K

    update(i:ID)
      pre i >0  /\ x[i] ~= x[i-1]
      eff x[i] := x[i-1]
```

declaration of “actions” or transition labels; actions can have parameter; this declares the actions update(0), update(1), ..., update(N-1)

symbols -> maps, /\ and, \/ or, ~= not equal, % mod

A language for specifying automata

```
automaton DijkstraTR(N:Nat, K:Nat), where K > N
  type ID: enumeration [0,...,N-1]
  type Val: enumeration [0,...,K-1]
  actions
    update(i:ID)
  variables
    x:[ID -> Val]
  transitions
    update(i:ID)
      pre i = 0 /\ x[i] = x[N-1]
      eff x[i] := (x[i] + 1) %  K

    update(i:ID)
      pre i >0  /\ x[i] ~= x[i-1]
      eff x[i] := x[i-1]
```

declaration of state variables or variables; this declares an array x[0], x[1], …, x[N-1] of Val's

symbols -> maps, /\ and, \/ or, ~= not equal, % mod

A language for specifying automata

```
automaton DijkstraTR(N:Nat, K:Nat), where K > N
  type ID: enumeration [0,...,N-1]
  type Val: enumeration [0,...,K-1]
  actions
    update(i:ID)
  variables
    x:[ID -> Val]
  transitions
    update(i:ID)
     pre i = 0 /\ x[i] = x[N-1]
     eff x[i] := (x[i] + 1) %  K

    update(i:ID)
     pre i >0  /\ x[i] ~= x[i-1]
     eff x[i] := x[i-1]
```

declaration of transitions: for each action this defines when the action can occur (pre) and how the state is updated when the action does occur (eff)

symbols -> maps, /\ and, \/ or, ~= not equal, % mod

The language defines an automaton

An automaton is a tuple $\mathcal{A} = \langle X, \Theta,\ A,\ \mathcal{D}\rangle$ where

- X is a set of names of variables; each variable $x \in X$ is associated with a type, $type(x)$
  - A valuation for X maps each variable in X to its type
  - Set of all valuations:$val(X)$ this is sometimes identified as the state space of the automaton
- $\Theta \subseteq val(X)$ is the set of initial or start states
- A is a set of names of actions or labels
- $\mathcal{D} \subseteq val(X) \times A \times val(X)$ is the set of transitions
  - a transition is a triple (u, a, u')
  - We write it as $u \rightarrow_a u'$

Well formed specifications in IOA Language define automata variables and valuations

variables s, v: Real; a: Bool

X = {s, v, a}

Example valuations of X

- $\langle s \mapsto 0,\ v \mapsto 5.5,\ a \mapsto 0 \rangle$
- $\langle s \mapsto 10,\ v \mapsto -2.5,\ a \mapsto 1 \rangle$

set of all possible **valuations** or "state space" is written as $val(X)$

$$val(X) = \{\langle s \mapsto c_1,\ v \mapsto c_2,\ a \mapsto c_3 \rangle \mid c_1, c_2 \in R,\ c_3 \in \{0,1\}\}$$

Well formed specifications in IOA Language define automata variables and valuations

variables s, v: Real; a: Bool

X = {s, v, a}

Example valuations of X

- $\langle s \mapsto 0,\ v \mapsto 5.5,\ a \mapsto 0 \rangle$
- $\langle s \mapsto 10,\ v \mapsto -2.5,\ a \mapsto 1 \rangle$

set of all possible **valuations** or "state space" is written as $val(X)$

$$val(X) = \{\langle s \mapsto c_1,\ v \mapsto c_2,\ a \mapsto c_3 \rangle \mid c_1, c_2 \in R,\ c_3 \in \{0,1\}\}$$

type ID: [0,…,N-1], Vals: [0, …, K-1]

variables x: [ID>Vals] defines an array

Fix N = 5, K = 7

x: [{0,…,4} -> {0,…,6}]

Example valuations:

$\langle x \mapsto \langle 0 \mapsto 0,\ 1 \mapsto 0,\ 2 \mapsto 0,\ 3 \mapsto 0,\ 4 \mapsto 0 \rangle\rangle$

$\langle x \mapsto \langle 0 \mapsto 7,\ 1 \mapsto 0,\ 2 \mapsto 0,\ 3 \mapsto 0,\ 4 \mapsto 0, \rangle\rangle$

Valuations are usually denoted by bold small characters

E.g.,

$\boldsymbol{u} = \langle x \mapsto \langle 0 \mapsto 0,\ 1 \mapsto 0,\ 2 \mapsto 0,\ 3 \mapsto 0,\ 4 \mapsto 0 \rangle\rangle$

Notations: ⌈ and []

$\boldsymbol{u}\lceil x$ is the value of variable *x* in $\boldsymbol{u}$

$\boldsymbol{u}\lceil\ x[4]$ =0 array notation [] works with ⌈ as expected

States and predicates

A *predicate* over a set of variable X is a Boolean-valued formula involving the variables in X. Examples:

- ϕ_1: $\mathrm{x}[1] = 1$
- ϕ_2: $\forall i \in ID,\ x[i] = 0$

A valuation u satisfies a predicate ϕ if substituting the values of the variables in u in ϕ makes it evaluate to True.

We write u$\vDash \phi$

Examples: $u = \langle x \mapsto \langle 0 \mapsto 0,\ 1 \mapsto 0,\ 2 \mapsto 0,\ 3 \mapsto 0,\ 4 \mapsto 0 \rangle\rangle$; $v = \langle x \mapsto \langle 0 \mapsto 0,\ 1 \mapsto 1,\ 2 \mapsto 0,\ 3 \mapsto 0,\ 4 \mapsto 0 \rangle\rangle$

- $u \vDash \phi_2,\ \ (u \nvDash \phi_1)$, $v \vDash \phi_1$ and $v \nvDash \phi_2$

$[[\phi]]$: set of all valuations that satisfy ϕ

- $[[\phi_1]] = \{\langle x \mapsto \langle 1 \mapsto 1,\ i \mapsto c_i\rangle_{\{i=0,2,\ldots,5\}}\rangle | c_i \in \{0, \ldots, 7\}\}$
- $[[\phi_2]] = \{\langle x \mapsto \langle 0 \mapsto 0,\ 1 \mapsto 0,\ 2 \mapsto 0,\ 3 \mapsto 0,\ 4 \mapsto 0,\ 5 \mapsto 0 \rangle\rangle\}$
- $\Theta \subseteq val(x)$ is the set of initial states of the automaton; often specified by a predicate over X

Actions

- **actions** section defines the set of actions of the automaton
- Examples
  - **actions** update(i:ID)
    defines $A = \{update[0], \ldots, update[5]\}$

  - **actions** brakeOn, brakeOff
    defines $A = \{brakeOn, brakeOff\}$

Transitions defined by preconditions and effects

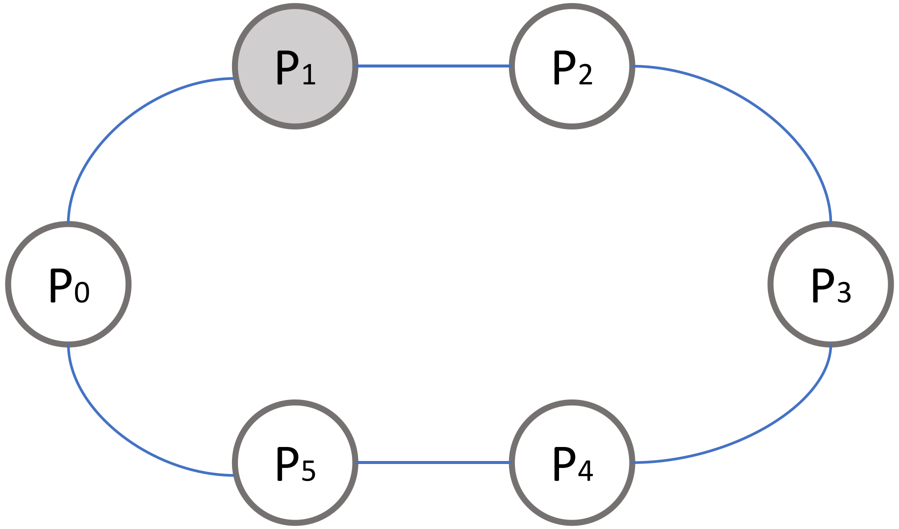
$\mathcal{D} \subseteq val(X) \times A \times val(X)$ is the set of transitions

$\mathcal{D} = \{(\boldsymbol{u}, a, \boldsymbol{u}') |$ such that $\boldsymbol{u} \vDash Pre_a$ and $(\boldsymbol{u}, \boldsymbol{u}') \vDash Eff_a\}$

$(\boldsymbol{u}, a, \boldsymbol{u}') \in \mathcal{D}$ is written as $\boldsymbol{u} \rightarrow_a \boldsymbol{u}'$

Example:

```
update(i:ID)
   pre i = 0 /\ x[i] = x[n-1]
   eff x[i] := x[i] + 1 mod k;
update(i:ID)
   pre i ≠ 0 /\ x[i] ≠x[i-1]
   eff x[i] := x[i-1];
```



Transitions defined by preconditions and effects

$\mathcal{D} \subseteq val(X) \times A \times val(X)$ is the set of transitions

$\mathcal{D} = \{(\boldsymbol{u}, a, \boldsymbol{u}')|$ such that $\boldsymbol{u} \vDash Pre_a$ and $(\boldsymbol{u}, \boldsymbol{u}') \vDash Eff_a\}$

$(\boldsymbol{u}, a, \boldsymbol{u}') \in \mathcal{D}$ is written as $\boldsymbol{u} \rightarrow_a \boldsymbol{u}'$

Example:

```
update(i:ID)
    pre i = 0 /\ x[i] = x[n-1]
    eff x[i] := x[i] + 1 mod k;
update(i:ID)
    pre i ≠ 0 /\ x[i] ≠x[i-1]
    eff x[i] := x[i-1];
```

$(\boldsymbol{u}, update(i), \boldsymbol{u}') \in \mathcal{D}$ *iff*

(a) $(i = 0 \wedge \boldsymbol{u}[\, x[0] = \boldsymbol{u}[x[5]$
$\quad \wedge \boldsymbol{u}'[x[0] = \boldsymbol{u}\,[x[0] + 1\, mod\ K) \vee$

(b) $(i \neq 0 \wedge \boldsymbol{u}\,[x[i] \neq \boldsymbol{u}\,[x[i-1]$
$\quad \wedge \boldsymbol{u}'\,[x[i] = \boldsymbol{u}[x[i-1]])$

Executions, Reachability, and Invariants

Give an automaton $\mathcal{A} = \langle X, \Theta, A, \mathcal{D} \rangle$

An execution models a particular behavior of the automaton $\mathcal{A}$

An *execution* of $\mathcal{A}$ is an alternating (possibly infinite) sequence of states and actions $\alpha = u_0 a_1 u_1 a_2 u_3$...such that:

1. $u_0 \in \Theta$
2. $\forall\, i$ in the sequence, $u_i \rightarrow_{a_{i+1}} u_{i+1}$

For a *finite* execution, $\alpha = u_0 a_1 u_1 a_2 u_3$ the *last state* $\alpha.lstate = u_3$, the *first state* $\alpha.fstate = u_0$, and the length of the execution is 3.

In general, how many executions does an $\mathcal{A}$ have?

Nondeterminism

For an action $a \in A$, Pre(a) is the formula defining its precondition, and Eff(a) is the relation defining the effect.

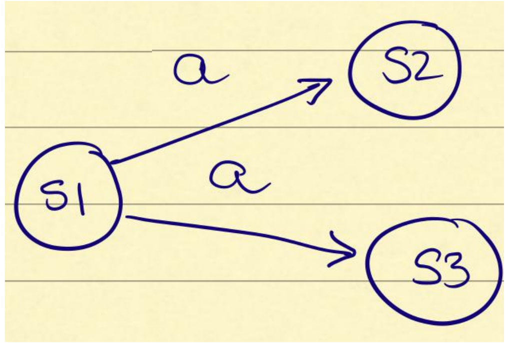
States satisfying precondition are said to *enable* the action

In general eff(a) could be a relation (nodeterministic!), but for this example it is a function

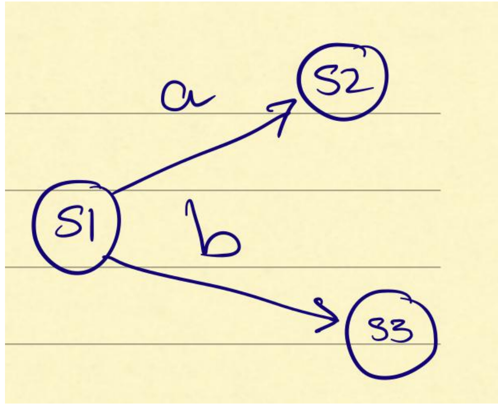
Nondeterminism

Nondeterminism

- Multiple post-states from the same action (internal)
- Multiple actions enabled from the same state (external)



internal



external

Reachable states and invariants

A state $\boldsymbol{u}$ is ***reachable*** if there exists an execution α such that $\alpha.lstate = \boldsymbol{u}$

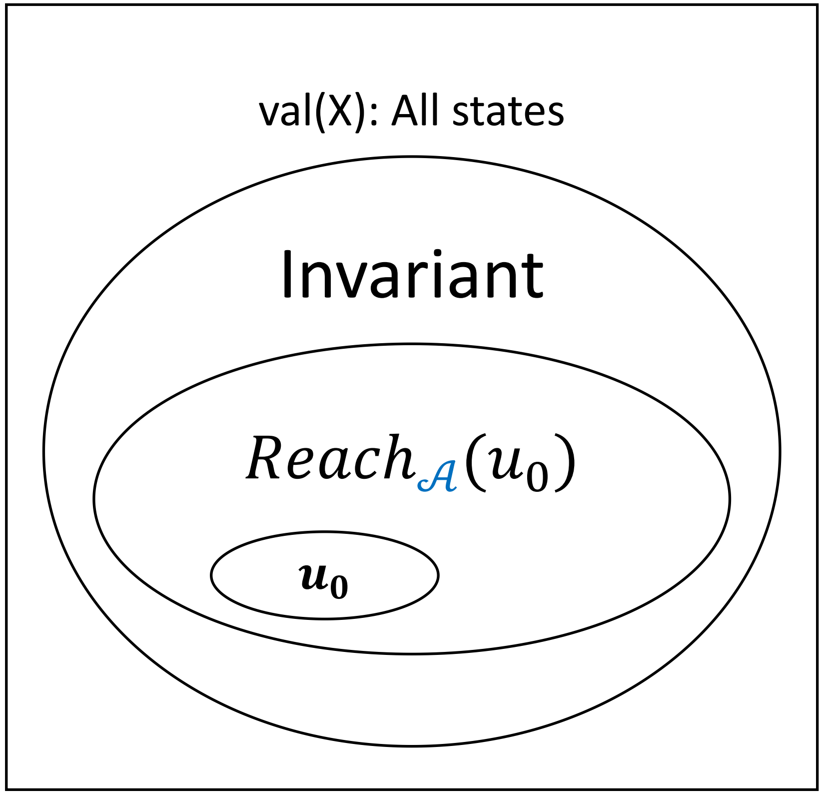
$Reach_{\mathcal{A}}(\Theta)$: set of states reachable from Θ by automaton $\mathcal{A}$

An ***invariant*** is a set of states I such that $Reach_{\mathcal{A}}(u) \subseteq I, \forall u \in \Theta$

Reachable states and invariants

$Reach_{\mathcal{A}}(\Theta)$: set of states reachable from Θ by automaton $\mathcal{A}$

An ***invariant*** is a set of states I such that $Reach_{\mathcal{A}}(u) \subseteq I, \forall u \in \Theta$

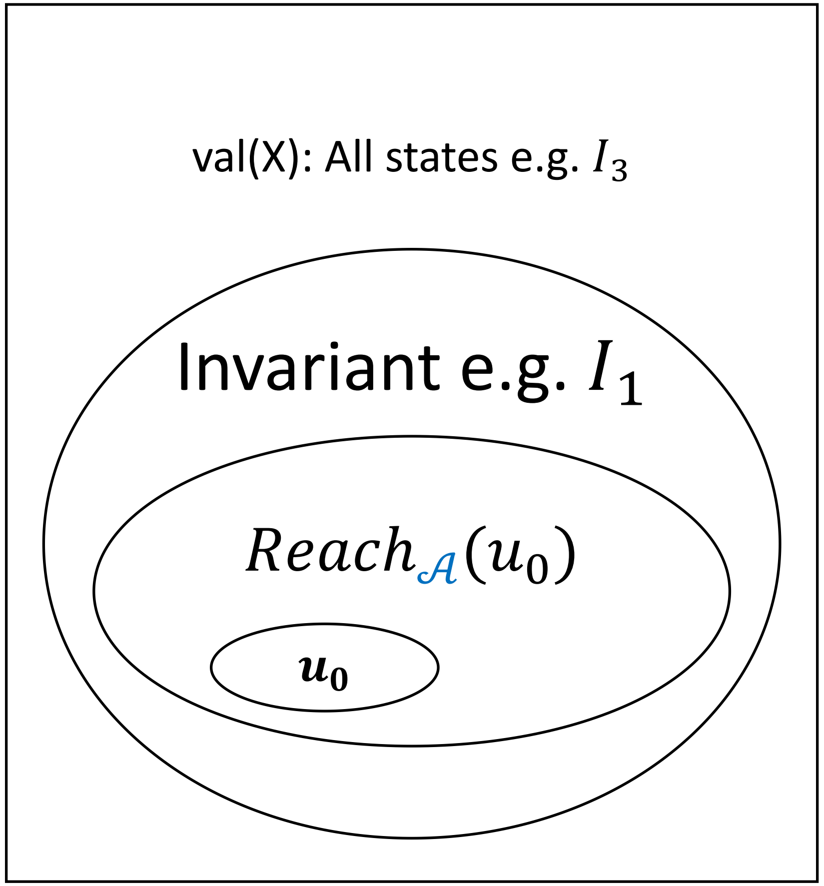


Candidate invariants for token Ring

I_1: "Exactly one process has the token".

$I_{\geq 1}$: "At least one process has a token".

I_3: "All processes have values at most K-1".



Reachability: a basic verification problem

- Q1. Given $\mathcal{A}$, is a state $\boldsymbol{u} \in val(X)$ reachable?

Reachability as graph search

- Q1. Given $\mathcal{A}$, is a state $\boldsymbol{u} \in val(X)$ reachable?

- Define a graph $G_{\mathcal{A}} = \langle V, E \rangle$ where
  - $V = val(X)$
  - $E = \{(u, u') | \exists\ a \in A,\ u \rightarrow_a u'\}$

- Q2. Does there exist a path in $G_{\mathcal{A}}$ from any state in Θ to u ?

Reachability as graph search

- Q1. Given $\mathcal{A}$, is a state $\boldsymbol{u} \in val(X)$ reachable?
- Define a graph $G_{\mathcal{A}} = \langle V, E \rangle$ where
  - $V = val(X)$
  - $E = \{(u, u') | \exists\ a \in A,\ u \rightarrow_a u'\}$
- Q2. Does there exist a path in $G_{\mathcal{A}}$ from any state in Θ to u ?
- Perform DFS/BFS on $G_{\mathcal{A}}$

Proving invariants by induction (Chapter 7)

Theorem 7.1. Given a automaton $\mathcal{A}$ = $\langle X, \Theta,\ A,\ \mathcal{D}\rangle$ and a set of states $I \subseteq val(X)$ if:

- (Start condition) for any $\boldsymbol{x} \in \Theta$ implies $\boldsymbol{x} \in I$, and
- (Transition closure) for any $\boldsymbol{x} \rightarrow_a \boldsymbol{x}'$ and $\boldsymbol{x} \in I$ implies $\boldsymbol{x}' \in I$

then I is an (inductive) invariant of $\mathcal{A}$. That is $Reach_{\mathcal{A}}(\Theta) \subseteq I$.

Proving invariants by induction (Chapter 7)

Theorem 7.1. Given a automaton $\mathcal{A} = \langle X, \Theta, A, \mathcal{D} \rangle$ and a set of states $I \subseteq val(X)$ if:

- (Start condition) for any $x \in \Theta$ implies $x \in I$, and
- (Transition closure) for any $x \rightarrow_a x'$ and $x \in I$ implies $x' \in I$

then I is an (inductive) invariant of $\mathcal{A}$. That is $Reach_{\mathcal{A}}(\Theta) \subseteq I$.

Proof. Consider any reachable state $\boldsymbol{x}$. By the definition of a reachable state, there exists an execution α of $\mathcal{A}$ such that $\alpha.lstate = \boldsymbol{x}$.

We proceed by induction on the length α

For the base case, α consists of a single starting state $\alpha = \boldsymbol{x} \in \Theta$, and by the Start condition, $\boldsymbol{x} \in I$.

For the inductive step, $\alpha = \alpha' a\, \boldsymbol{x}$ where $a \in A$. By the induction hypothesis, we know that $\alpha'.lstate \in I$.

Invoking Transition closure on $\alpha'.lstate \rightarrow_a \boldsymbol{x}$ we obtain $\boldsymbol{x} \in I$. QED

Proving invariants by induction for Dijkstra

Theorem 7.1. Given a automaton $\mathcal{A} = \langle X, \Theta, A, \mathcal{D} \rangle$ and a set of states $I \subseteq val(X)$ if:

- (Start condition) for any $x \in \Theta$ implies $x \in I$, and
- (Transition closure) for any $x \rightarrow_a x'$ and $x \in I$ implies $x' \in I$

then I is an (inductive) invariant of $\mathcal{A}$. That is $Reach_{\mathcal{A}}(\Theta) \subseteq I$.

- I: "Exactly one process has the token".

(Start condition): since $\forall i \;\; \boldsymbol{x}[x[i]] = 0$, only P0 has token, so $\boldsymbol{x} \in I$

(Transition closure): Fix a $\boldsymbol{x} \rightarrow_a \boldsymbol{x}'$ such that $\boldsymbol{x} \in I$.

Two cases to consider.

$a = update(0)$

$a = update(i), \; \mathrm{i} > 0$

```
automaton DijkstraTR(N:Nat, K:Nat), where K > N
  type ID: enumeration [0,...,N-1]
  type Val: enumeration [0,...,K-1]
  actions
     update(i:ID)
  variables
     x:[ID -> Val] initially forall i:ID x[i] = 0
  transitions
     update(i:ID)
       pre i = 0 /\ x[i] = x[(N-1)]
       eff x[i] := (x[i] + 1) % K

      update(i:ID)
        pre i >0 /\ x[i] ~= x[i-1]
        eff x[i] := x[i-1]
```

Proving invariants by induction for Dijkstra

Theorem 7.1. Given a automaton $\mathcal{A} = \langle X, \Theta, A, \mathcal{D} \rangle$ and a set of states $I \subseteq val(X)$ if:

- (Start condition) for any $x \in \Theta$ implies $x \in I$, and
- (Transition closure) for any $x \to_a x'$ and $x \in I$ implies $x' \in I$

then I is an (inductive) invariant of $\mathcal{A}$. That is $Reach_{\mathcal{A}}(\Theta) \subseteq I$.

```
automaton DijkstraTR(N:Nat, K:Nat), where K > N
  type ID: enumeration [0,...,N-1]
  type Val: enumeration [0,...,K-1]
  actions
     update(i:ID)
  variables
     x:[ID -> Val] initially forall i:ID x[i] = 0
   transitions
     update(i:ID)
       pre i = 0 /\ x[i] = x[(N-1)]
       eff x[i] := (x[i] + 1) %  K

      update(i:ID)
       pre i >0  /\ x[i] ~= x[i-1]
       eff x[i] := x[i-1]
```

- I: "Exactly one process has the token".

(Start condition): since $\forall i \;\; \boldsymbol{x}[x[i]] = 0$, only P0 has token, so $\boldsymbol{x} \in I$

(**Transition closure**): Fix a $\boldsymbol{x} \to_a \boldsymbol{x}'$ such that $\boldsymbol{x} \in I$.

Two cases to consider.

1. If $a = update(0)$ then
    - (a) since $\boldsymbol{x} \vDash Pre(update(0))$ it follows that $\boldsymbol{x}[x[0] = \boldsymbol{x}[x[N-1]$
    - (b) since $\boldsymbol{x} \vDash I_1$ it follows that $\forall i > 0,\; \boldsymbol{x}[x[i]] = \boldsymbol{x}[x[i-1]$
    - (c) $\boldsymbol{x}'[x[0] \neq \boldsymbol{x}'[x[N-1]$ by applying (a) and $Eff(update(0))$ to $\boldsymbol{x}$
    - (d) $\boldsymbol{x}'[x[1] \neq \boldsymbol{x}'[x[0]$ by applying (b) and $Eff(update(0))$ to $\boldsymbol{x}$
    - (e) $\forall i > 1 \; \boldsymbol{x}'[x[i] = \boldsymbol{x}'[x[i-1]$ by applying (b) and $Eff(update(0))$ to $\boldsymbol{x}$

    Now there is only one token hold by P1. Therefore $\boldsymbol{x}' \in I$.
2. If $a = update(i)$, $\mathrm{i} > 0$ then fix arbitrary $i > 0$ … (do it as an exercise)

From above **Theorem** it follows that I is an invariant of DijkstraTR

Review

Goal of this course: model anything!

This lecture: model **computations**

Today: **Automaton** as a model for computations (e.g., your program)

More in the rest of the class: model physical process (e.g., motors), model machine learning models (e.g., neural nets), ...

```
automaton DijkstraTR(N:Nat, K:Nat), where K > N
  type ID: enumeration [0,...,N-1]
  type Val: enumeration [0,...,K-1]
  actions
    update(i:ID)
  variables
    x:[ID -> Val] initially forall i:ID x[i] = 0
   transitions
    update(i:ID)
      pre i = 0 /\ x[i] = x[(N-1)]
      eff x[i] := (x[i] + 1) %  K

     update(i:ID)
       pre i >0  /\ x[i] ~= x[i-1]
       eff x[i] := x[i-1]
```