Modeling Computation

Verifying cyber-physical systems

mitras@illinois.edu

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 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

Example: Dijkstra's mutual exclusion algorithm

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

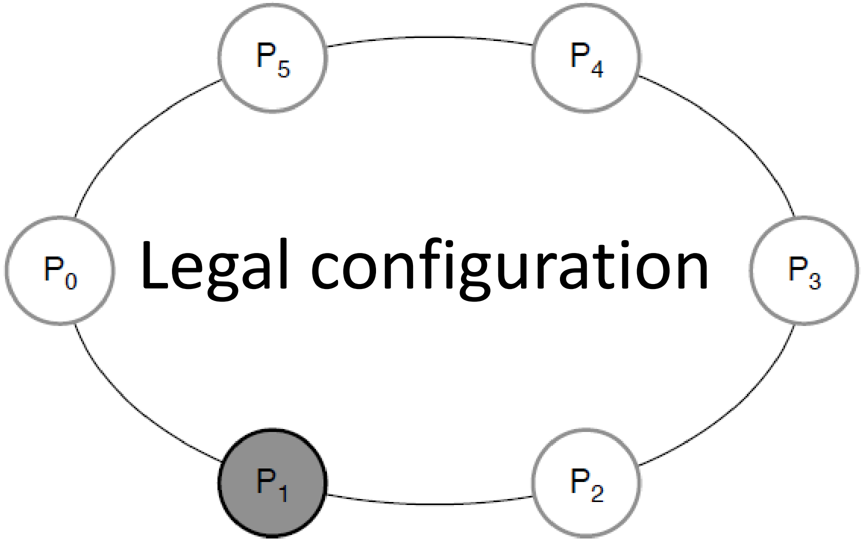
- Collection of processes that send and receive bits over a ring network so that only one of them has a "token" to access a critical resource (e.g., a shared calendar)

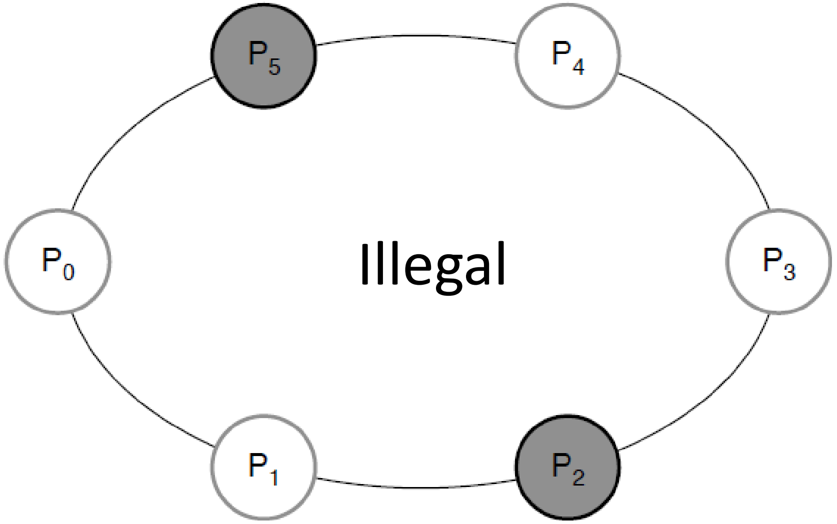
Discrete model

- 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



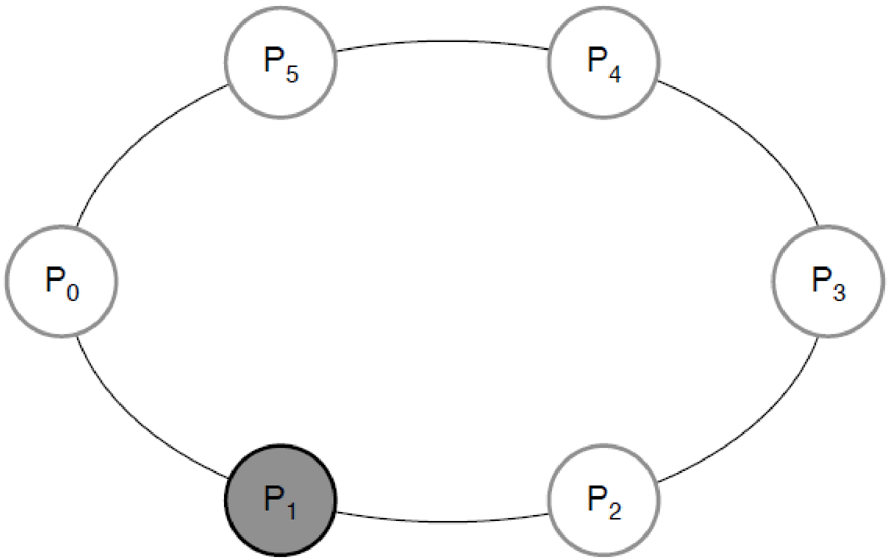


N processes with ids 0, 1, …, N-1

Unidirectional means: 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 Algorithm ['74]



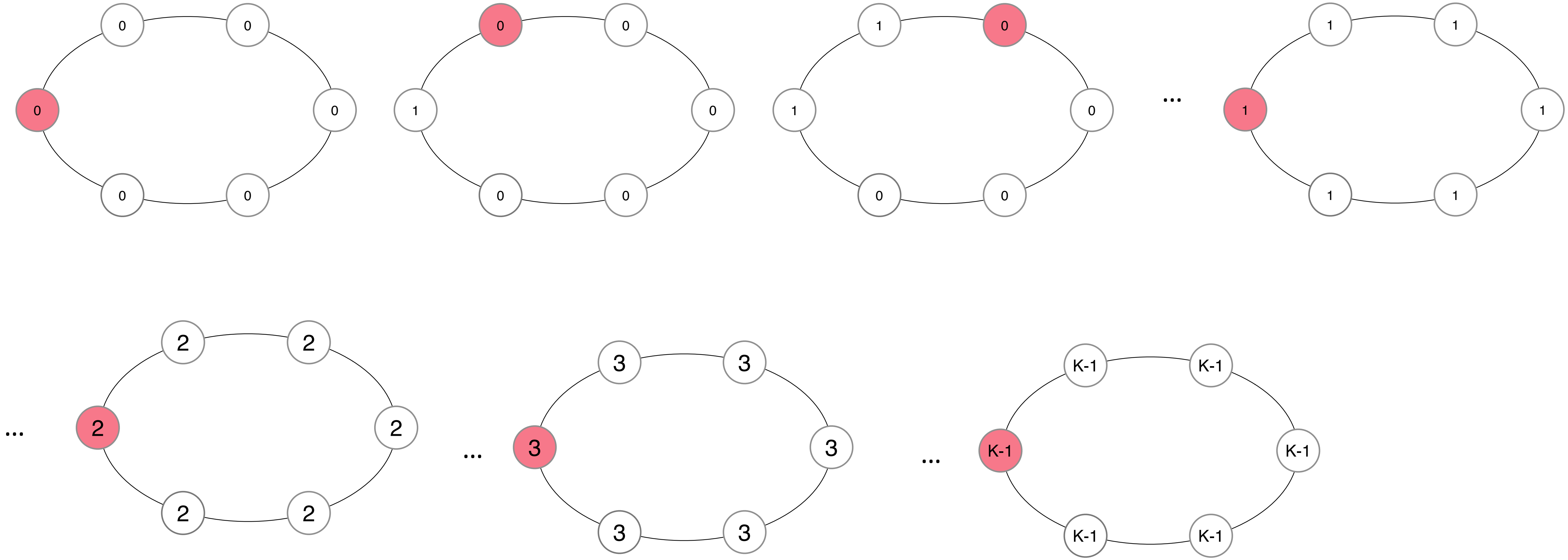
N processes: 0, 1, …, N-1

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

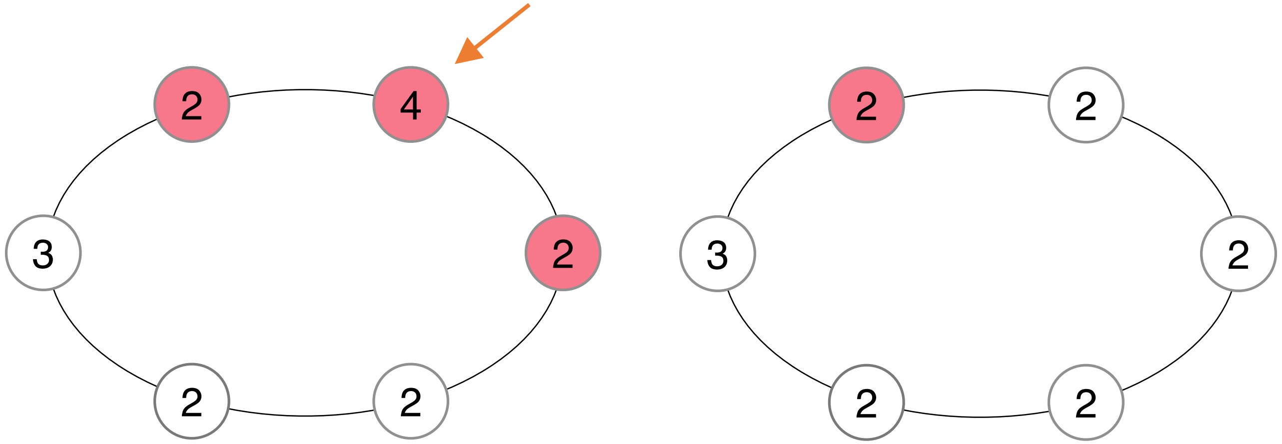
P_0	**if** x[0] = x[N-1]	**then** x[0] := x[0] + 1 mod K
P_j j > 0	**if** x[j] ≠ x[j -1]	**then** x[j] := x[j-1]

(p_i has TOKEN if and only if the blue conditional is true)

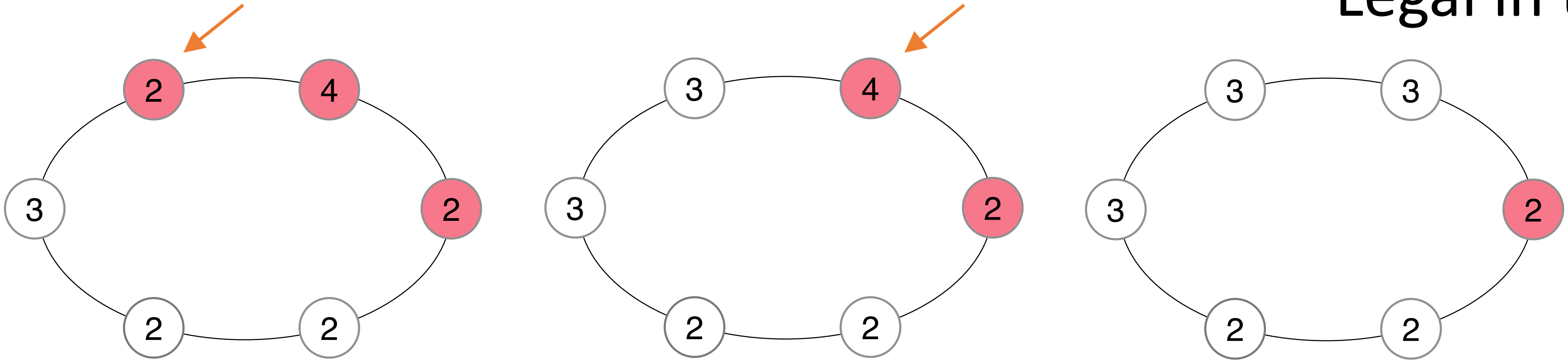
Sample executions: from a legal state (single token)



Execution from an illegal state



Legal in single “step”



Legal in two steps

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]
  actions
     update(i:ID)
  variables
     x:[ID -> K]
   transitions
     update(i:ID) where i = 0
       pre i = 0 /\ x[i] = x[(i-1)]
       eff x[i] := (x[i] + 1) %  K

      update(i:ID)$ where
        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]
  actions
     update(i:ID)
  variables
     x:[ID -> K]
   transitions
     update(i:ID) where i = 0
       pre i = 0 /\ x[i] = x[(i-1)]
       eff x[i] := (x[i] + 1) %  K

      update(i:ID)$ where
       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]
  actions
     update(i:ID)
  variables
     x:[ID -> Val]
   transitions
     update(i:ID) where i = 0
       pre i = 0 /\ x[i] = x[(i-1)]
       eff x[i] := (x[i] + 1) %  K

      update(i:ID)$ where
        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]
  actions
     update(i:ID)
  variables
     x:[ID -> Val]
   transitions
     update(i:ID) where i = 0
       pre i = 0 /\ x[i] = x[(i-1)]
       eff x[i] := (x[i] + 1) %  K

      update(i:ID)$ where
       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]
  actions
     update(i:ID)
  variables
     x:[ID -> Val]
   transitions
     update(i:ID) where i = 0
      pre i = 0 /\ x[i] = x[(i-1)]
      eff x[i] := (x[i] + 1) %  K

      update(i:ID)$ where
       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]
  actions
     update(i:ID)
  variables
     x:[ID -> Val]
   transitions
     update(i:ID) where i = 0
      pre i = 0 /\ x[i] = x[(i-1)]
      eff x[i] := (x[i] + 1) %  K

      update(i:ID)$ where
       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 $val(X)$

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

type ID: [0,…,N-1]

variables x: [ID>Vals]

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

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

$\boldsymbol{u} \lceil \ x$[4] =0 array notation [] works with $\lceil$ 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: $\forall i \in indices, x[i] = 0$

A valuation **u satisfies a predicate $\boldsymbol{\phi}$** if substituting the values of the variables in **u** in ϕ makes it evaluate to **True.**

We write u$\vDash$ $\boldsymbol{\phi}$

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

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

$[[\boldsymbol{\phi}]]$: set of all valuations that satisfy $\boldsymbol{\phi}$

- $[[\phi_1]] = \{\langle x \mapsto \langle 1 \mapsto 0, i \mapsto c_i \rangle_{\{i=0,2,\ldots,5\}}\rangle | c_i \in \{0, \ldots, 7\}$
- $[[\phi_2]] = \{\langle x \mapsto \langle 0 \mapsto 0, 1 \mapsto 0, 2 \mapsto 0, 3 \mapsto 0, 4 \mapsto 0, 5 \mapsto 0 \rangle\rangle\}$
- $\Theta \subseteq val(x)$ 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 $A = \{update[0], \dots, update[5]\}$

  - **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**

$\mathcal{D} = \{(\boldsymbol{u}, a, \boldsymbol{u}') \mid$ 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:

internal update(i:ID)

  pre i = 0 $\wedge$ x[i] = x[n-1]

  eff x[i] := x[i] + 1 mod k;

internal update(i:ID)

  pre i $\neq$ 0 $\wedge$ x[i] $\neq$x[i-1]

  eff x[i] := x[i-1];

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

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

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

Executions, Reachability, and Invariants

Automaton $\mathcal{A} = \langle X, \Theta, A, \mathcal{D} \rangle$ where

An executions 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, $u_i \rightarrow_{a_{i+1}} u_{i+1}$

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

Nondeterminism

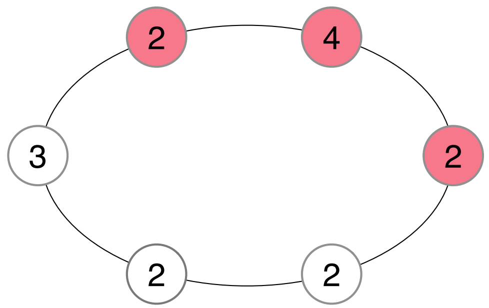
For an action $a \in A$, Pre(a) is the formula defining its **pre**condition, and Eff(a) is the relation defining the **eff**ect.

States satisfying precondition are said to **enable** the action

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

Nondeterminism

- Multiple actions enabled from the same state
- Multiple post-states from the same action



Reachable states and invariants

A state u is **reachable** if there exists an execution that ends at $\boldsymbol{u}$

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

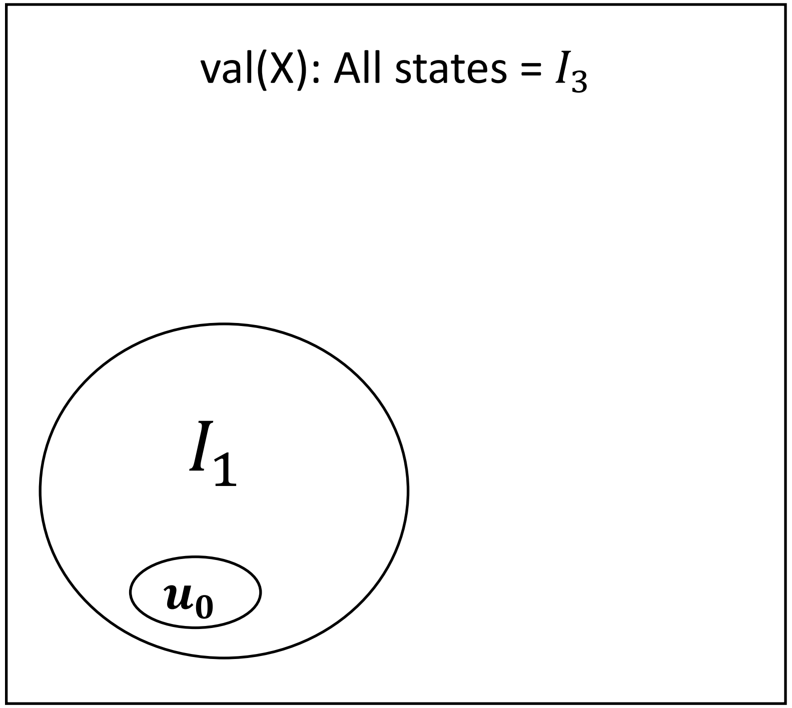
An **invariant** is a set of states I such that $Reach_{\mathcal{A}} \subseteq I$

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 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 $G_{\mathcal{A}}$

Reach as fixpoint of Post?

Importance of Invariants

Reading Assignments

- Modeling computation: Chapter 2 of CPSBook
- Specification language: Appendix C of CPSBook
- Next: Overview of complexity classes for understanding hardness of different verification problems
- Reading assignment: Appendix B of CPSBook
  - Turing Machines
  - Decidability
  - Complexity classes P, NP, PSPACE, NL
  - Reductions
- Reference: Any standard textbook on theory of computation, e.g., Introduction to Theory of Computation by Michael Sipser