A finite-state mutual exclusion protocol

Process 1:

```
          while true  {
              .....
nc :      x = 2;
rq :      b1 = true;
wt :      wait until(x == 1 || ¬ b2) {
cs :          ... critical section ...}
          b1 = false;
              .....
          }
```

Process 2:

```
          while true  {
              .....
nc :      x = 1;
rq :      b2 = true;
wt :      wait until(x == 2 || ¬ b1) {
cs :          ... critical section ...}
          b2 = false;
              .....
          }
```

A finite-state mutual exclusion protocol

Desirable properties:

- Mutual exclusion

  it is **never** *the case that Process 1 and Process 2 are in their critical sections at the same time*

- Accessibility

  whenever a process leaves its noncritical section, it will **eventually** *enter its critical section*

Is the protocol correct?

Process 1:

```
          while true  {
              .....
nc :      x = 2;
rq :      b1 = true;
wt :      wait until(x == 1 || ¬ b2) {
cs :          ... critical section ...}
          b1 = false;
              .....
          }
```

Process 2:

```
          while true  {
              .....
nc :      x = 1;
rq :      b2 = true;
wt :      wait until(x == 2 || ¬ b1) {
cs :          ... critical section ...}
          b2 = false;
              .....
          }
```

Mutual exclusion is violated

Possible state sequence:

$$\langle nc_1, \quad nc_2, \quad x = 1, \quad b_1 = \text{false}, \quad b_2 = \text{false} \rangle$$
$$\langle nc_1, \quad rq_2, \quad x = 1, \quad b_1 = \text{false}, \quad b_2 = \text{false} \rangle$$
$$\langle rq_1, \quad rq_2, \quad x = 2, \quad b_1 = \text{false}, \quad b_2 = \text{false} \rangle$$
$$\langle wt_1, \quad rq_2, \quad x = 2, \quad b_1 = \text{true}, \quad b_2 = \text{false} \rangle$$
$$\langle cs_1, \quad rq_2, \quad x = 2, \quad b_1 = \text{true}, \quad b_2 = \text{false} \rangle$$
$$\langle cs_1, \quad wt_2, \quad x = 2, \quad b_1 = \text{true}, \quad b_2 = \text{true} \rangle$$
$$\langle cs_1, \quad cs_2, \quad x = 2, \quad b_1 = \text{true}, \quad b_2 = \text{true} \rangle$$

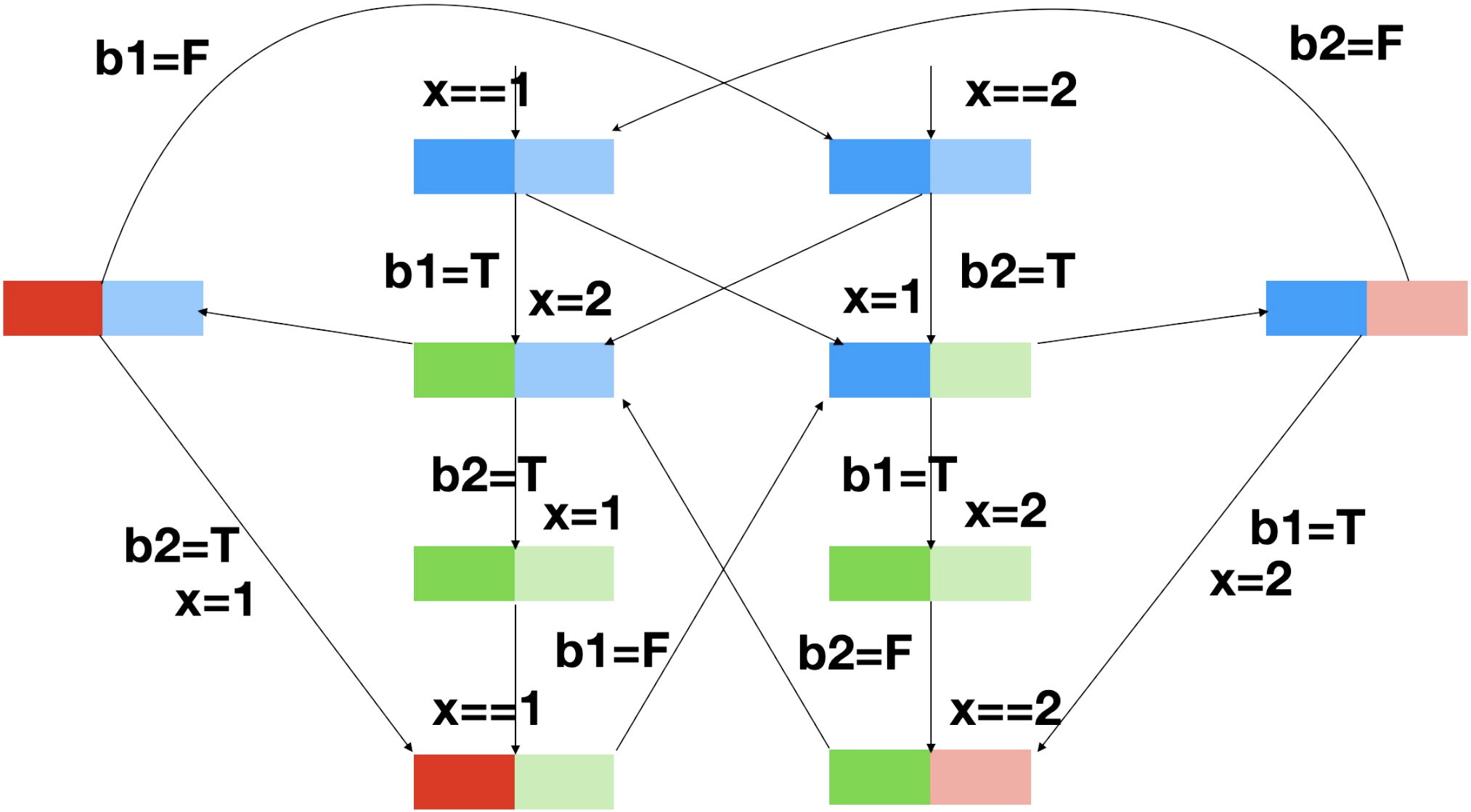
Peterson's mutual exclusion algorithm

Process 1:

```
          while true  {
              ......
nc :          ⟨(b1, x) = (true, 2); ⟩
wt :          wait until(x == 1 || ¬ b2) {
cs :              . . . critical section . . .}
              b1 = false;
              ......
          }
```

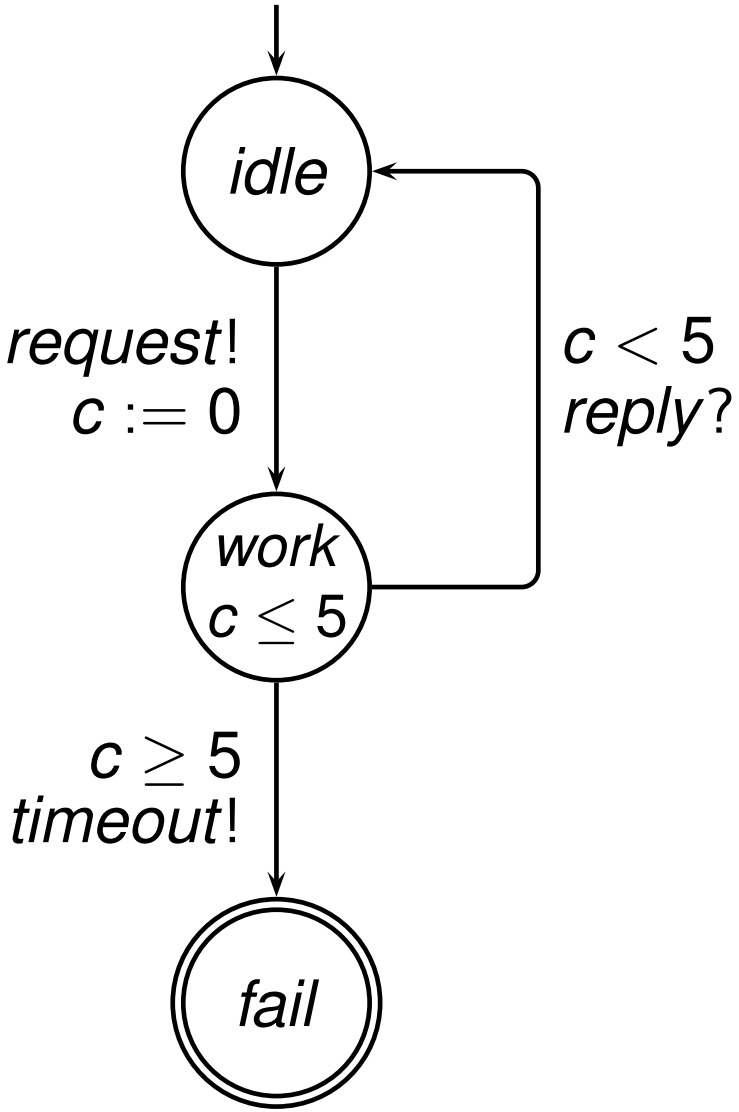
Process 2:

```
          while true  {
              ......
nc :          ⟨(b2, x) = (true, 1); ⟩
wt :          wait until(x == 2 || ¬ b1) {
cs :              . . . critical section . . .}
              b2 = false;
              ......
          }
```

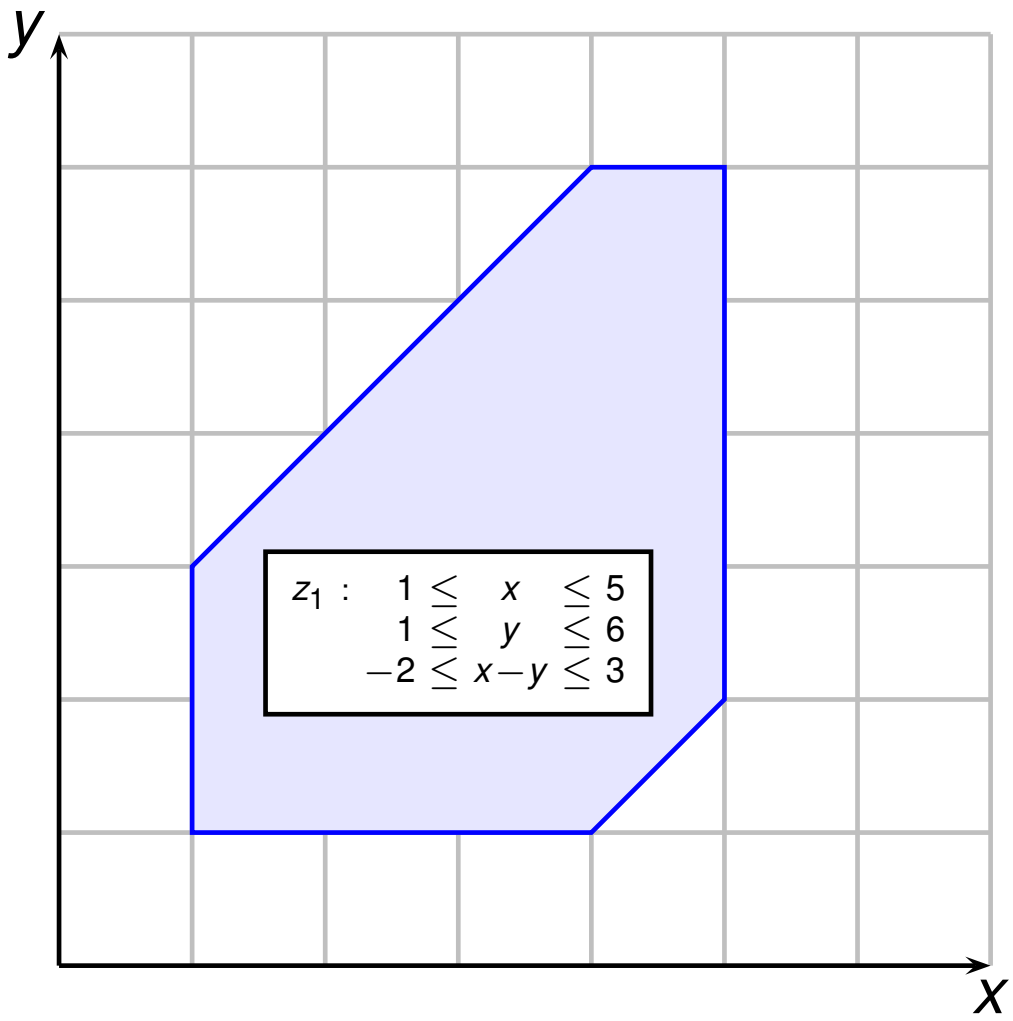

b1=F
x==1
x==2
b2=F
b1=T
x=2
x=1
b2=T
b2=T
x=1
b1=T
x=2
b2=T
x=1
b1=T
x=2
b1=F
b2=F
x==1
x==2

Timed Automata

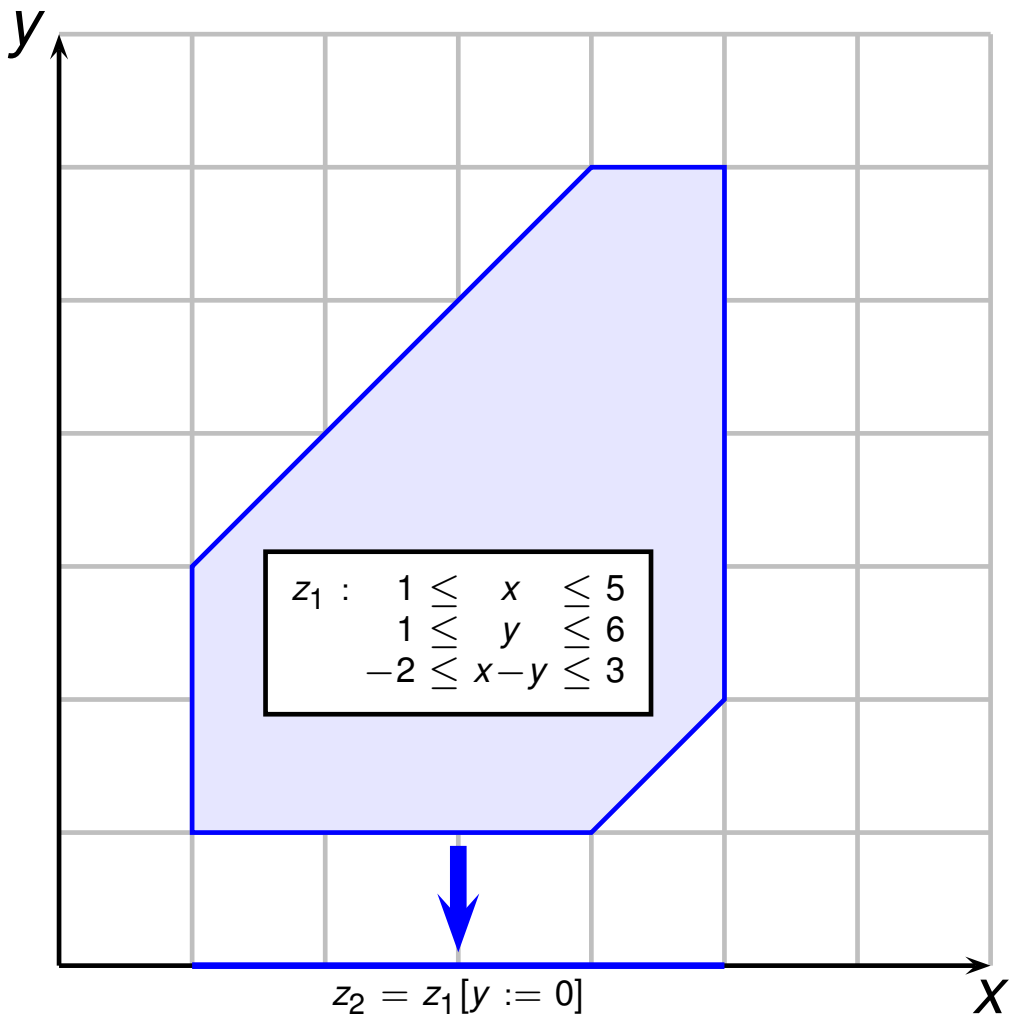
- Finite-state systems + clocks
- Locations with invariants
- Transitions:
  - guard
  - synchronization label
  - clock resets
- state = location + clock valuation
  $\rightarrow$ infinitely many states!
  $\rightarrow$ idea: finite number of equivalence classes



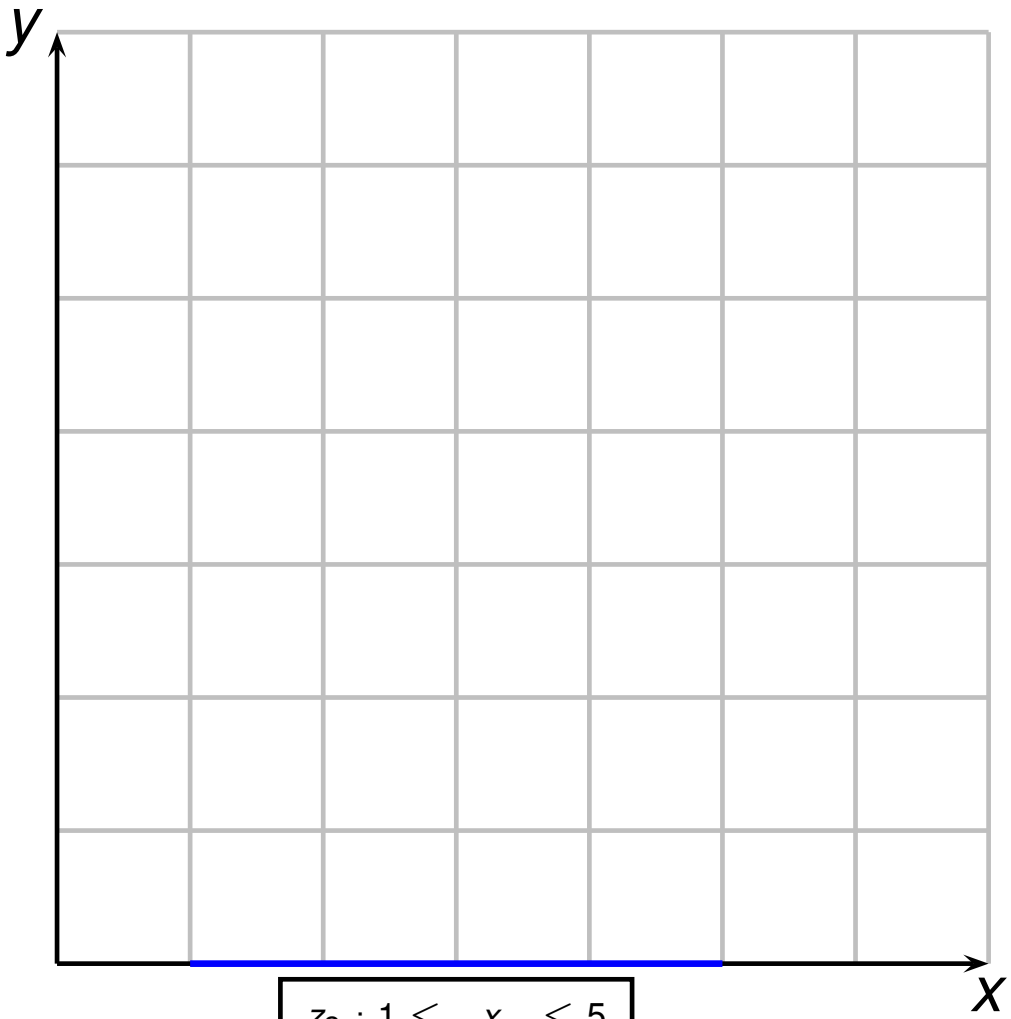
Clock zones: abstraction for timed automata



Clock zones: abstraction for timed automata

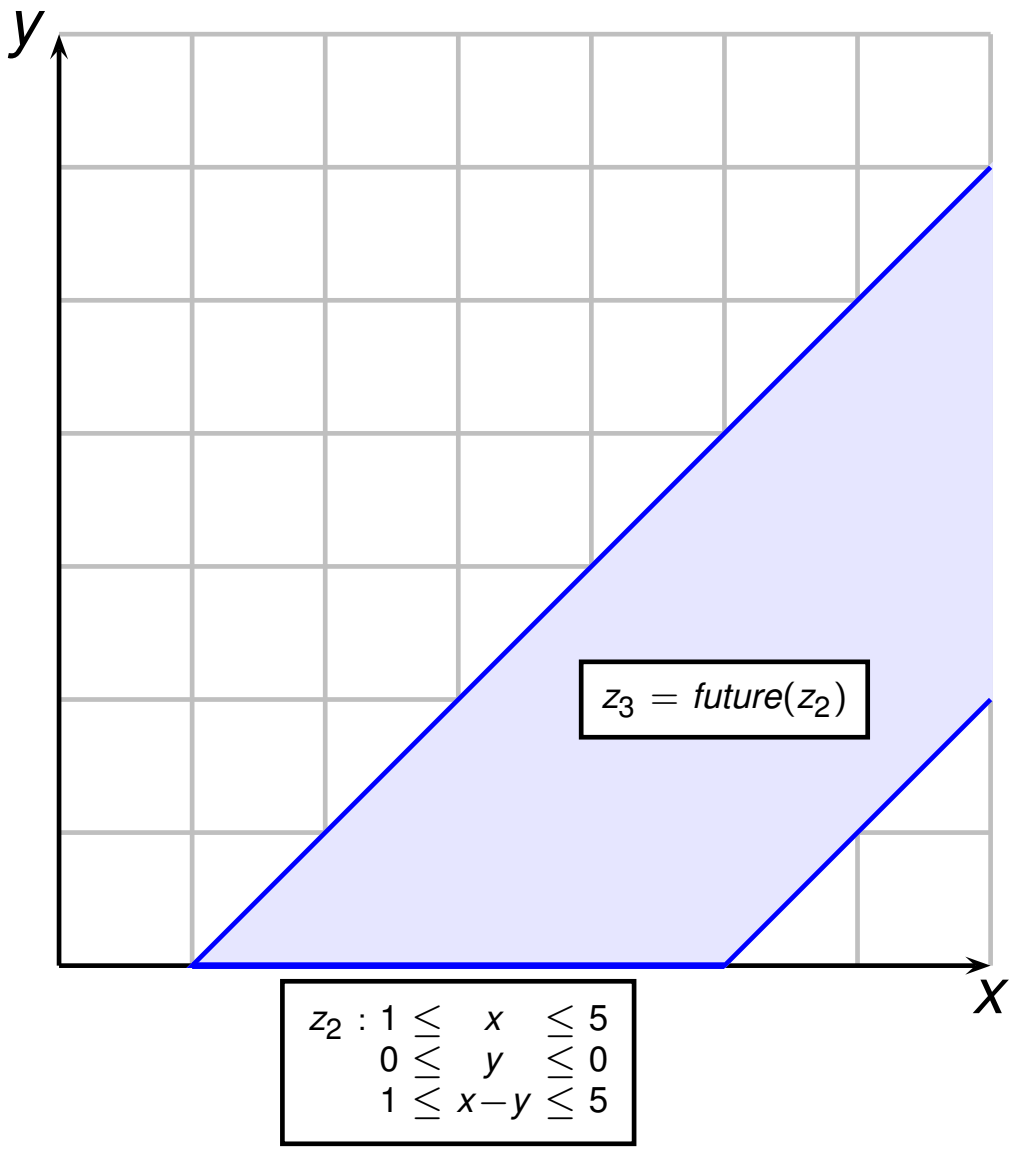


Clock zones: abstraction for timed automata

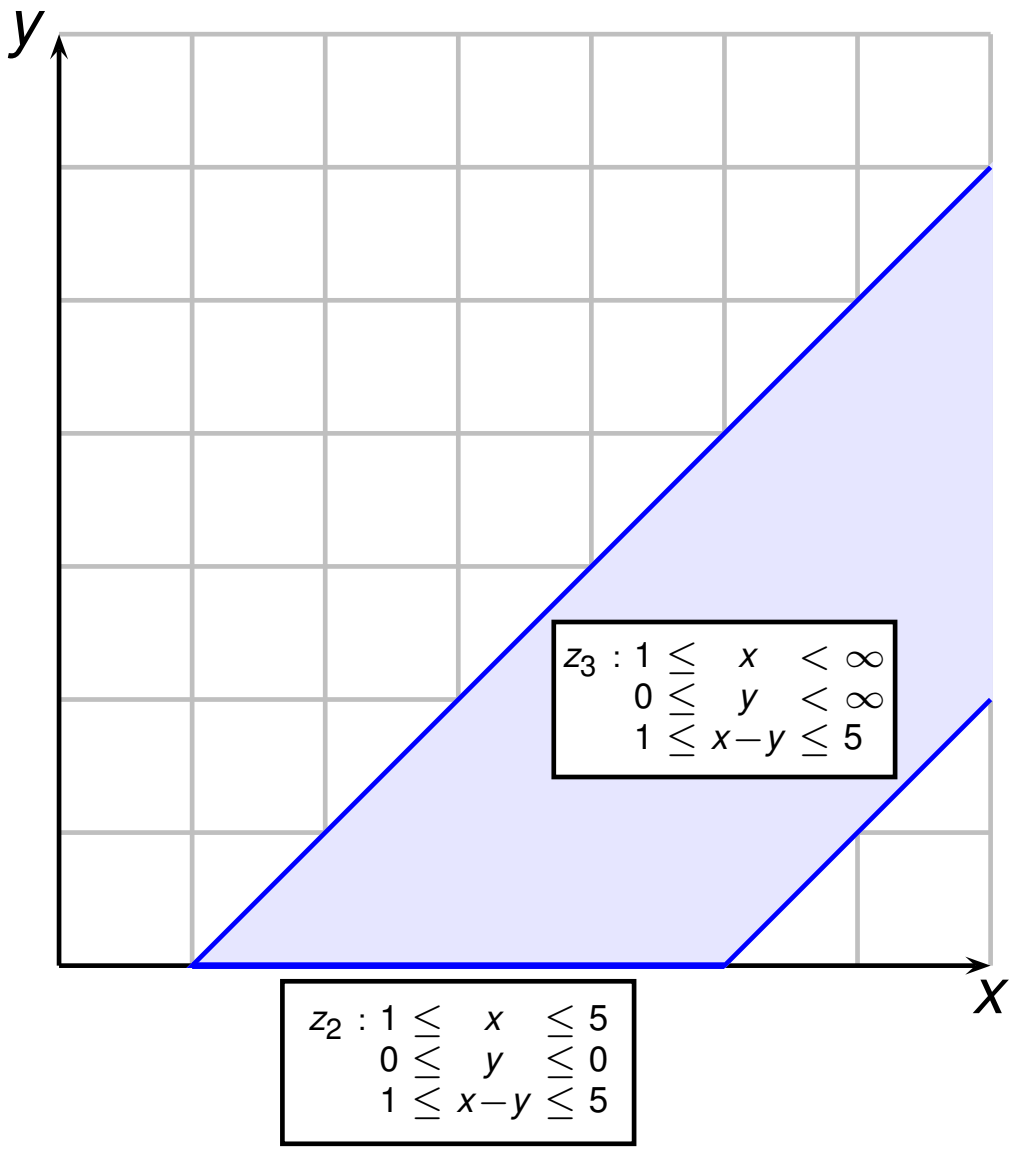


$$z_2 : \begin{array}{rcccl} 1 & \leq & x & \leq & 5 \\ 0 & \leq & y & \leq & 0 \\ 1 & \leq & x-y & \leq & 5 \end{array}$$

Clock zones: abstraction for timed automata

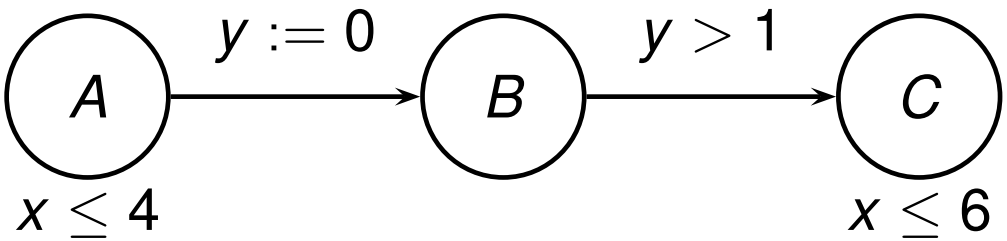


Clock zones: abstraction for timed automata

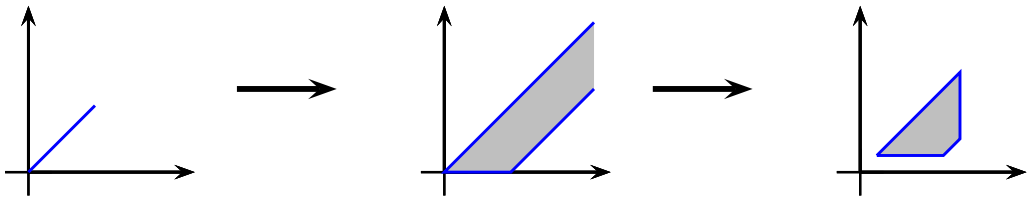


Clock zones: abstraction for timed automata

- timed automaton



- zone graph



- abstract states: (q, z) – finite number of states!
- reachability in timed automata is **decidable**.

Infinite-state systems: software

```
method isqrt(N : int) returns (R : int)
        requires N >= 0 ;
        ensures  (R + 1) * (R + 1) > N ;
        ensures  R * R <= N ;
{
    R := 0 ;
        while ((R + 1) * (R + 1) <= N)
        {
                R := R + 1 ;
        }
}
```

Infinite-state systems: software

```
method isqrt(N : int) returns (R : int)
        requires N >= 0 ;
        ensures  (R + 1) * (R + 1) > N ;
        ensures  R * R <= N ;
{
    R := 0 ;
        while ((R + 1) * (R + 1) <= N)
                invariant R * R <= N ;
        {
                R := R + 1 ;
        }
}
```

Course overview

- Transition systems
- Linear-time temporal logic
- Linear-time properties
- Automata-theoretic model checking
- Bounded model checking
- Computation tree logic
- Symbolic model checking
- Equivalences and abstraction
- Timed automata
- Deductive verification
- Decision procedures for verification
- Automatic abstraction refinement

Transition systems

Transition systems

- model to describe the behaviour of systems
- directed graphs where **nodes** represent **states** and **edges** represent **transitions**
- state:
  - in **hardware**: the current value of the registers together with the values of the input bits
  - in **software**: the current values of all program variables + the program counter
- transition: (“state change”)
  - in **hardware**: the change of the registers and output bits for a new input
  - in **software**: the execution of a program statement

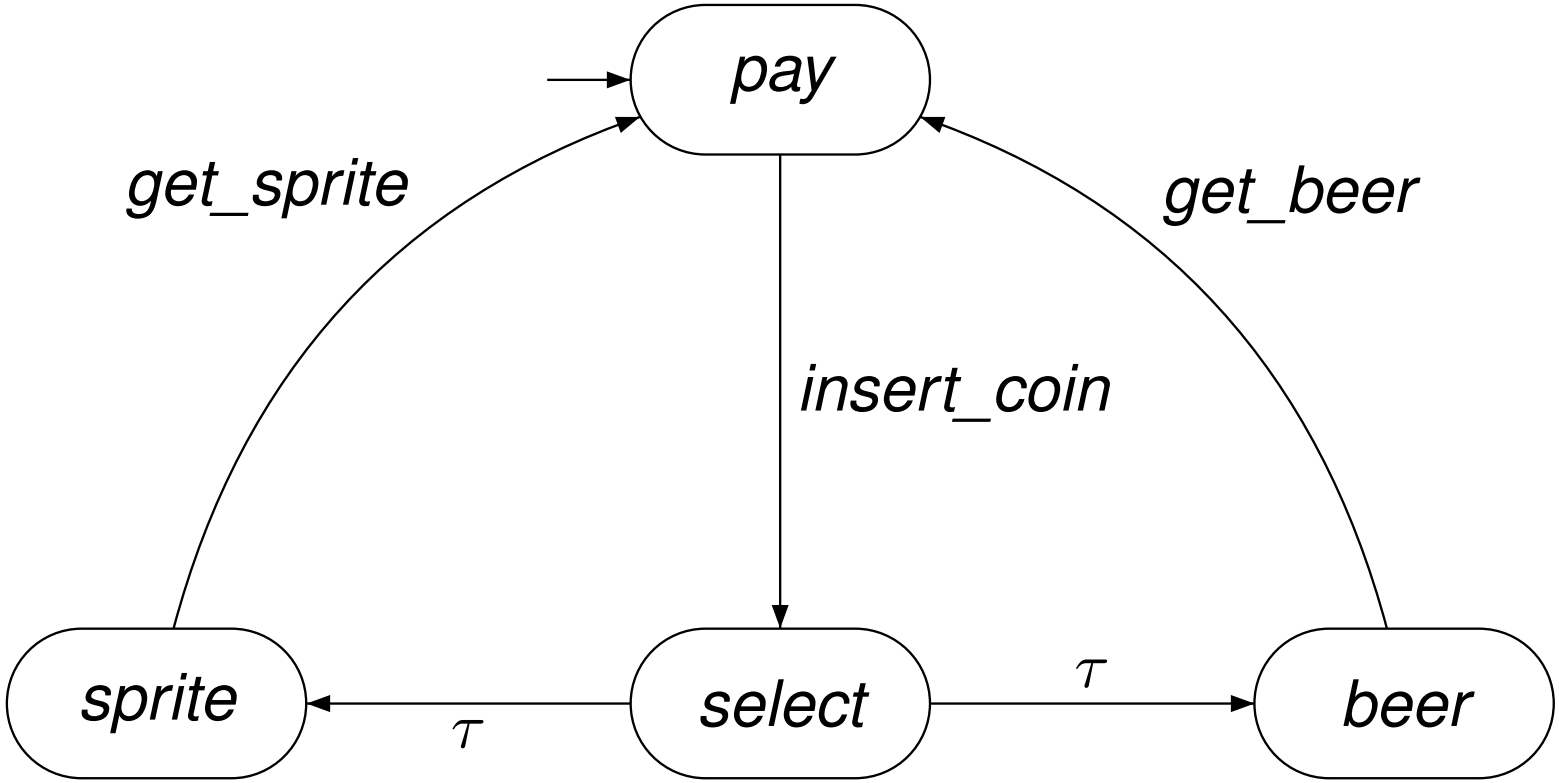
Transition systems

A **transition system** TS is a tuple $(S, Act, \rightarrow, I, AP, L)$ where

- S is a set of states
- Act is a set of actions
- $\longrightarrow \subseteq S \times Act \times S$ is a transition relation
- $I \subseteq S$ is a set of initial states
- AP is a set of atomic propositions
- $L : S \rightarrow 2^{AP}$ is a labeling function

Notation: $s \xrightarrow{\alpha} s'$ for $(s, \alpha, s') \in \longrightarrow$

A beverage vending machine



labeling: $L(s) = \{s\}$

alternative labeling:
$L(pay) = \emptyset, L(sprite) = L(beer) = \{drink\}, L(select) = \{paid\}$

Direct successors and predecessors

$$Post(s,\alpha) = \Big\{\, s' \in S \mid\ s \xrightarrow{\alpha} s' \,\Big\}, \quad Post(s) = \bigcup_{\alpha \in Act} Post(s,\alpha)$$

$$Pre(s,\alpha) = \Big\{\, s' \in S \mid\ s' \xrightarrow{\alpha} s \,\Big\}, \quad Pre(s) = \bigcup_{\alpha \in Act} Pre(s,\alpha).$$

$$Post(C,\alpha) = \bigcup_{s \in C} Post(s,\alpha), \quad Post(C) = \bigcup_{s \in C} Post(s) \text{ for } C \subseteq S.$$

$$Pre(C,\alpha) = \bigcup_{s \in C} Pre(s,\alpha), \quad Pre(C) = \bigcup_{s \in C} Pre(s) \text{ for } C \subseteq S.$$

A state s is called terminal if $Post(s) = \emptyset$

Action- and *AP*-determinism

- A transition system is action-deterministic iff:

$$|I| \leq 1 \quad \text{and} \quad |Post(s, \alpha)| \leq 1$$

for all s, α.

- A transition system is *AP*-deterministic iff:

$$|I| \leq 1 \quad \text{and} \quad |\underbrace{Post(s) \cap \{ s' \in S \mid L(s') = A \}}_{\text{equally labeled successors of } s}| \leq 1$$

for all $s, A \in 2^{AP}$.

The role of nondeterminism

Nondeterminism is an important modeling feature

- to model concurrency by interleaving
  - no assumption about the relative speed of processes
- to model implementation freedom
  - only describes **what** a system should do, not **how**
- to model under-specified systems,
  or abstractions of real systems
  - use incomplete information

Executions

- A finite execution fragment of TS is an alternating sequence of states and actions ending with a state:

  $\varrho = s_0\, \alpha_1\, s_1\, \alpha_2 \ldots \alpha_n\, s_n$ such that $s_i \xrightarrow{\alpha_{i+1}} s_{i+1}$ for all $0 \leq i < n$.

- An infinite execution fragment of TS is an infinite, alternating sequence of states and actions:

  $\rho = s_0\, \alpha_1\, s_1\, \alpha_2\, s_2\, \alpha_3 \ldots$ such that $s_i \xrightarrow{\alpha_{i+1}} s_{i+1}$ for all $0 \leq i$.

- An execution of TS is an initial, maximal execution fragment
  - a maximal execution fragment is either finite ending in a terminal state, or infinite
  - an execution fragment is initial if $s_0 \in I$

Example executions

$$\rho_1 = \mathit{pay} \xrightarrow{\mathit{coin}} \mathit{select} \xrightarrow{\tau} \mathit{sprite} \xrightarrow{\mathit{sget}} \mathit{pay} \xrightarrow{\mathit{coin}} \mathit{select} \xrightarrow{\tau} \mathit{sprite} \xrightarrow{\mathit{sget}} \ldots$$

$$\rho_2 = \mathit{select} \xrightarrow{\tau} \mathit{sprite} \xrightarrow{\mathit{sget}} \mathit{pay} \xrightarrow{\mathit{coin}} \mathit{select} \xrightarrow{\tau} \mathit{beer} \xrightarrow{\mathit{bget}} \ldots$$

$$\varrho = \mathit{pay} \xrightarrow{\mathit{coin}} \mathit{select} \xrightarrow{\tau} \mathit{sprite} \xrightarrow{\mathit{sget}} \mathit{pay} \xrightarrow{\mathit{coin}} \mathit{select} \xrightarrow{\tau} \mathit{sprite}$$

- ρ_1 and ϱ are **initial**,
- ρ_2 is **not initial**
- ϱ is **not maximal** as it does not end in a terminal state
- assuming that ρ_1 and ρ_2 are infinite, they are **maximal**

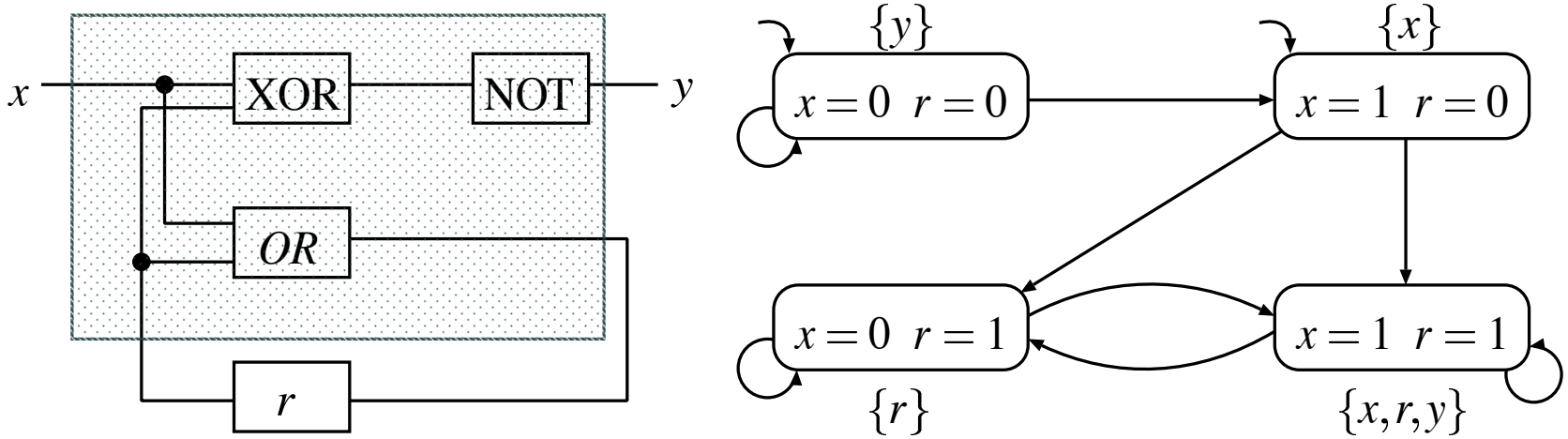
Reachable states

State $s \in S$ is called reachable in TS
if there exists an initial finite execution fragment

$$s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_n} s_n = s \,.$$

Reach(TS) denotes the set of all reachable states in TS.

Modeling hardware: sequential circuits



Transition system representation of a simple hardware circuit:

- **input** variable x, **output** variable y, and **register** r
- output function $\neg(x \oplus r)$
- register evaluation function $x \vee r$

Atomic propositions

Consider **two possible state-labelings:**

- Let $AP = \{ x, y, r \}$
  - $L(\langle x = 0, r = 1 \rangle) = \{ r \}$ and $L(\langle x = 1, r = 1 \rangle) = \{ x, r, y \}$
  - $L(\langle x = 0, r = 0 \rangle) = \{ y \}$ and $L(\langle x = 1, r = 0 \rangle) = \{ x \}$
  - **example property:** "once the r becomes 1, it remains 1"
- Let $AP' = \{ x, y \}$
  - $L(\langle x = 0, r = 1 \rangle) = \emptyset$ and $L(\langle x = 1, r = 1 \rangle) = \{ x, y \}$
  - $L(\langle x = 0, r = 0 \rangle) = \{ y \}$ and $L(\langle x = 1, r = 0 \rangle) = \{ x \}$
  - **example property:** "y is set infinitely often"
  - the register valuation is no longer visible

Modeling software: data-dependent systems

The beverage vending machine revisited:

"Abstract" transitions:

$$start \xrightarrow{true:coin} select \quad \text{and} \quad start \xrightarrow{true:refill} start$$

$$select \xrightarrow{nsprite>0:sget} start \quad \text{and} \quad select \xrightarrow{nbeer>0:bget} start$$

$$select \xrightarrow{nsprite=0 \wedge nbeer=0:ret\_coin} start$$

Action	Effect on variables
coin	
ret_coin	
sget	$nsprite := nsprite - 1$
bget	$nbeer := nbeer - 1$
refill	$nsprite := max$; $nbeer := max$

Some preliminaries

- typed variables with a valuation that assigns values to variables
  - e.g., $\eta(x) = 17$ and $\eta(y) = -2$
- Boolean conditions over *Var*
  - propositional logic formulas whose propositions are of the form "$\overline{x} \in \overline{D}$", where $\overline{x}$ denotes a tuple of variables
  - for example: $(-3 < x \leq 5) \ \wedge \ (y = \mathit{green})$
- effect of the actions is formalized by means of a mapping:

$$\mathit{Effect} : \mathit{Act} \times \mathit{Eval}(\mathit{Var}) \rightarrow \mathit{Eval}(\mathit{Var})$$

example: $\alpha \equiv x := y+5$ and
evaluation $\eta(x) = 17$ and $\eta(y) = -2$

- $\mathit{Effect}(\alpha, \eta)(x) \ = \ \eta(y) + 5 = 3,$
- $\mathit{Effect}(\alpha, \eta)(y) \ = \eta(y) = -2$

Program graphs

A **program graph** PG over set Var of typed variables is a tuple

$$(Loc, Act, Effect, \longrightarrow, Loc_0, g_0) \quad \text{where}$$

- Loc is a set of locations with initial locations $Loc_0 \subseteq Loc$
- Act is a set of actions
- $Effect : Act \times Eval(Var) \to Eval(Var)$ is the effect function
- $\longrightarrow \subseteq Loc \times (\underbrace{Cond(Var)}_{\text{Boolean condition over } Var} \times Act) \times Loc$
  is the transition relation
- $g_0 \in Cond(Var)$ is the initial condition.

Notation: $\ell \xrightarrow{g:\alpha} \ell'$ denotes $(\ell, g, \alpha, \ell') \in \longrightarrow$

Beverage vending machine

- $Loc = \{ start, select \}$ with $Loc_0 = \{ start \}$
- $Act = \{ bget, sget, coin, ret\_coin, refill \}$
- $Var = \{ nsprite, \ \ nbeer \}$ with domain $\{ 0, 1, \ldots, max \}$
- *Effect*:

$$
\begin{array}{lcl}
Effect(coin, \eta) & = & \eta \\
Effect(ret\_coin, \eta) & = & \eta \\
Effect(sget, \eta) & = & \eta[nsprite := nsprite - 1] \\
Effect(bget, \eta) & = & \eta[nbeer := nbeer - 1] \\
Effect(refill, \eta) & = & \eta[nsprite := max, nbeer := max]
\end{array}
$$

- $g_0 = (nsprite = max \ \wedge \ nbeer = max)$

From program graphs to transition systems

- Basic strategy: unfolding
  - **state** = location (current control) ℓ + data valuation η
  - **initial state** = initial location satisfying the initial condition g_0
- Propositions and labeling
  - **propositions:** "ℓ" and "$x \in D$" for $D \subseteq dom(x)$
  - $\langle \ell, \eta \rangle$ is **labeled with** "ℓ" and all conditions that hold in η
- if $\ell \xrightarrow{g:\alpha} \ell'$ and g holds in η, then $\langle \ell, \eta \rangle \xrightarrow{\alpha} \langle \ell', \mathit{Effect}(\alpha, \eta) \rangle$