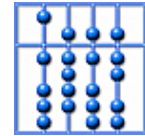


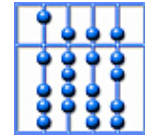
Vorlesung Specification of Distributed Systems

Dr. Bernhard Schätz
Leopold-Franzens Universität Innsbruck
Sommersemester 2005



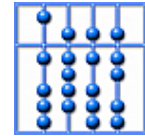
Overview

1. Introduction
2. Basics: Behavior, Interaction, Concurrency
3. Coroutines
4. Communicating Processes
5. Data Flow Models
6. State-Based Models
7. Coordination
8. Executions
9. Property Descriptions

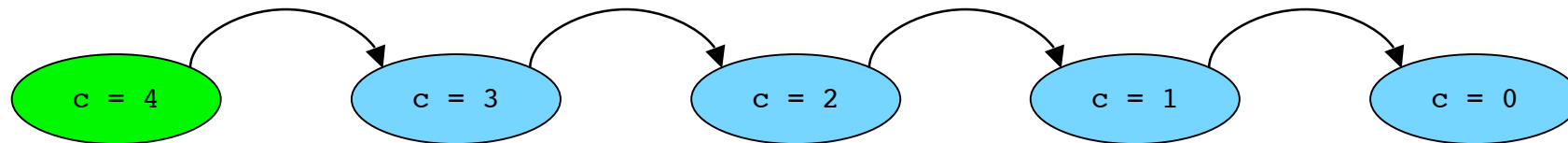


Overview

1. Introduction
2. Basics
3. Coroutines
 1. Modeling Complex Computation: Extended Transition Systems
 2. Model Implicit Interaction: Shared Memory Models
 3. Application: Threads
 4. Concepts: Interference, race conditions
4. Data Flow Models
5. Communicating Processes
6. Coordination
7. State-Based Models
8. Executions
9. Property Descriptions



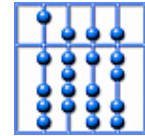
Recap: Modeling Computations



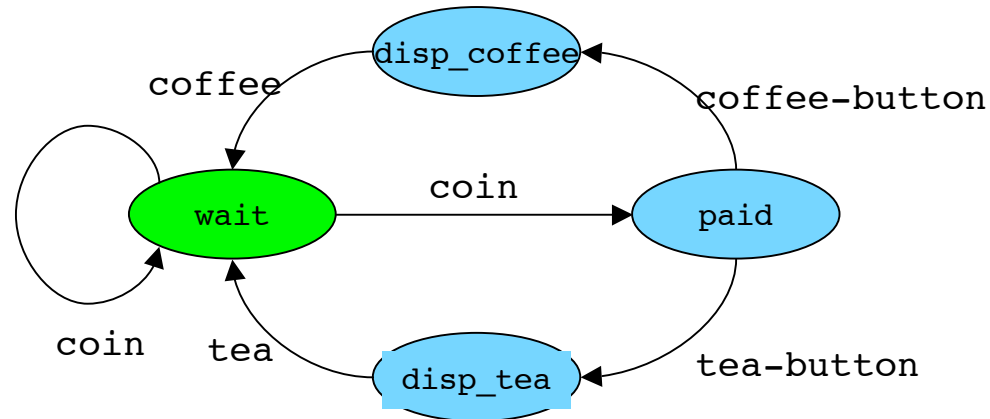
Concepts:

- State: Observation of a system at a specific instance of time
- Transition: Atomic action changing the state of a computation
- Transition relation: Set of possible actions of a computation
- Execution Trace: Sequence of states during a computation

Model: Transition System (S, s_0, T)



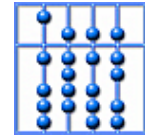
Recap: Modeling Reactivity



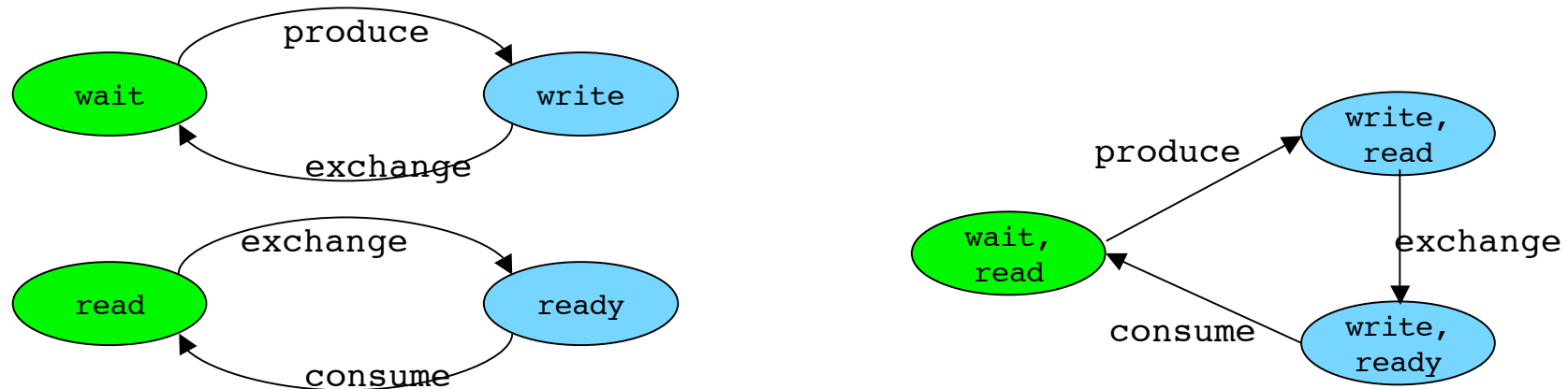
Concepts:

- Interactions (alphabet): Joint actions/observations of system and environment
- Observation Trace: Sequence of interaction during an execution
- Choice: Alternative behavior offered by a system
- Nondeterminism: Alternative behavior enforced by a system
- Input/Output: Interaction controlled by the environment/system

Model: Labeled Transition System (S, A, S_0, T)



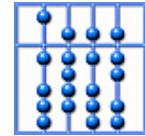
Recap: Modeling Concurrency



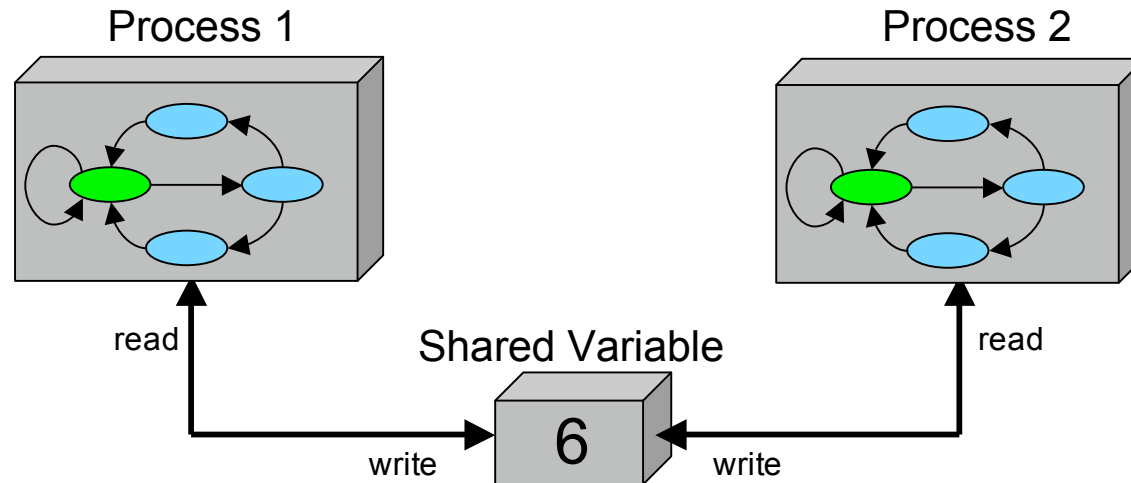
Concepts:

- Concurrent execution: Synchronized alternative execution
- Independent interaction: Interleaved execution of interactions
- Synchronized Interaction: Simultaneous execution of interactions

Model: Synchronized Product Automaton

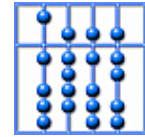


Motivation: Shared Memory Systems



Shared Memory Systems:

- Complex computations: Control and **data space**
- Loose coupling: **Unsynchronized** executions
- Implicit interaction: Communication via **shared variables**



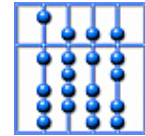
2.1 Describing Complex Computations: Extended Automata

Goal: Define a notation to describe the behavior of complex algorithmic computations

- Distinction between control and data state
- Abstract states as sets of concrete states
- Extended transitions by pre- and post-condition

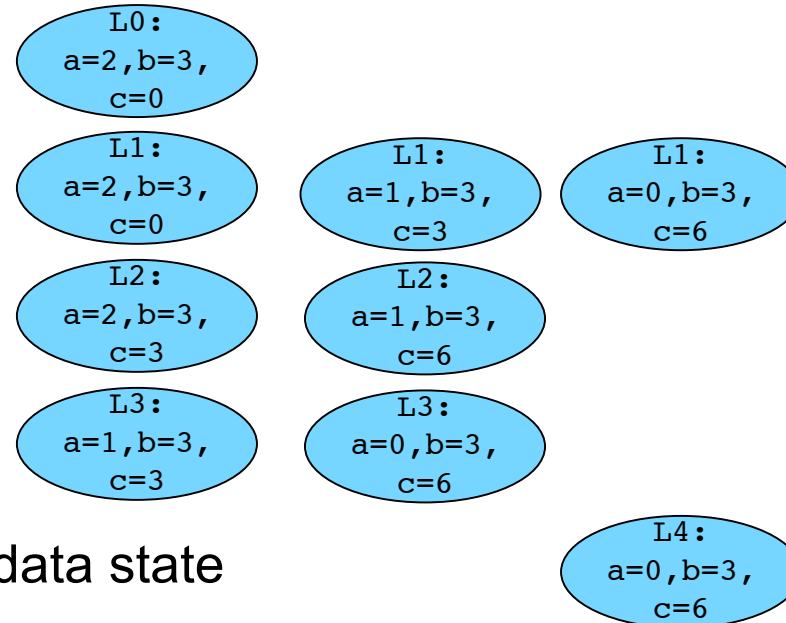
Concept: Control/data state, location, invariant, pre-/post condition

Notation: Extended Finite Automaton



Concept: Control State/Data State

```
L0:  int a=2,b=3,c=0;  
L1:  while(a>0) do {  
L2:      c = c + b;  
L3:      a = a - 1; }  
L4:
```

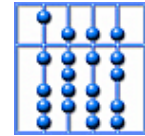


Purpose: Separating control and data state

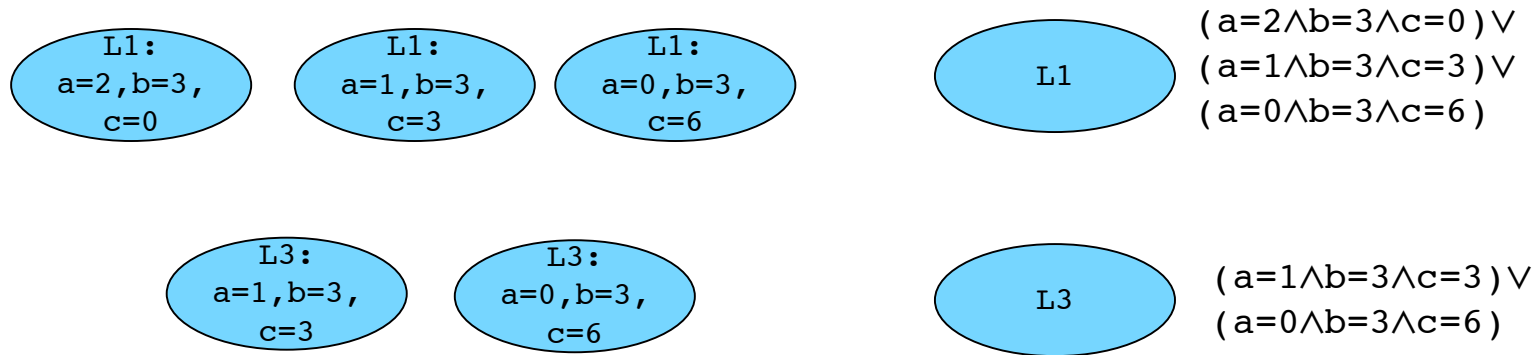
Concept: Control State/Data State = Location and set of assignments

- Control state: Location labels L
- Data state: Variable assignments V over variables X
- States: $S = L \times V$

Example: $L = \{L=, \dots, L4\}$, $X = \{a, b, c\}$, $V = \{ \dots, (a = 2, b = 3, c = 0), \dots \}$



Concept: Invariant Condition

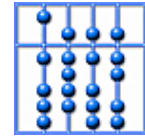


Purpose: Describing sets of data states

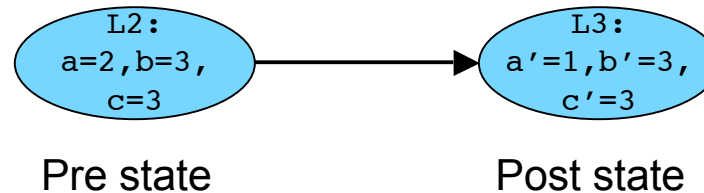
Concept: Invariant condition $\text{Inv}(X)$

- Condition label: Assigns predicate over data space
- Invariant condition: Characterizes assignments at location

Example: $a > 0 \wedge c = 0 \equiv \{(a=1, b=0, c=0), (a=1, b=1, c=0), \dots\}$



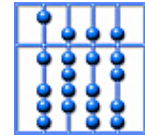
Notation: Extended Transitions



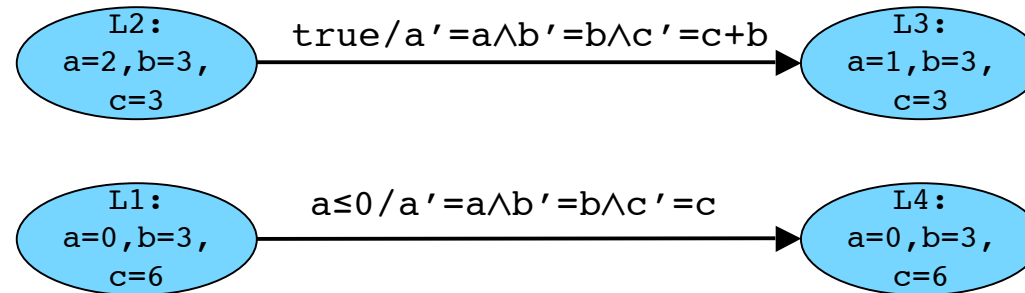
Purpose: Describe change by relating states

Concept: Change predicate characterizing pairs of data states in $V \times V$:

- Pre state: Data state prior to execution
- Post state: Data state after execution
- Notation: Pre/post state variables
 - Pre state variables: X
 - Post state variables: X'
- Example: $a' = a - 1 \wedge b' = b \wedge c' = c$



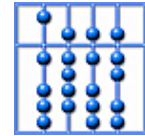
Notation: Extended Transitions



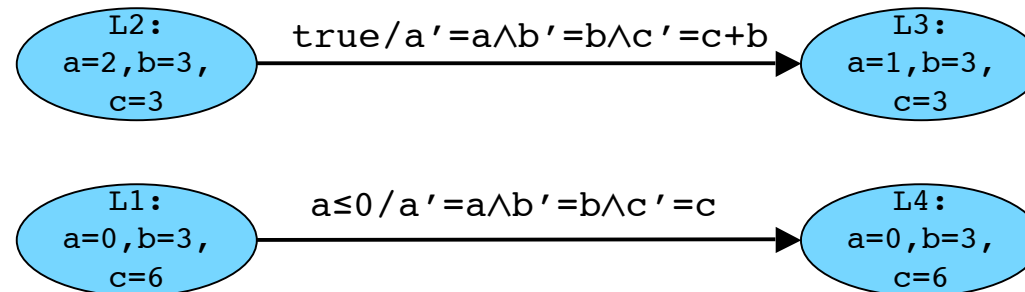
Purpose: Compact description of a collection of transitions

Notation: Extended Transition $T \subseteq S \times S$:

- Precondition: State prior to execution
 - Condition enabling execution of transition
 - Predicate $\text{Pre}(X)$ over pre data state
- Postcondition:
 - Condition valid after execution of transition
 - Predicate $\text{Post}(X, X')$ over pre and post data state
- Notation: $s \xrightarrow{\text{Pre}(X)/\text{Post}(X, X')} s'$



Notation: Extended Transitions

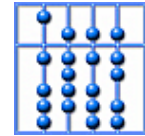


Interpretation:

$$s \xrightarrow{\text{Pre}(X)/\text{Post}(X,X')} s' \equiv \{((s, v), (s', v')) \mid v \in \text{Pre}(X) \wedge v' \in \text{Pre}(X) \wedge \text{Post}(X, X')\}$$

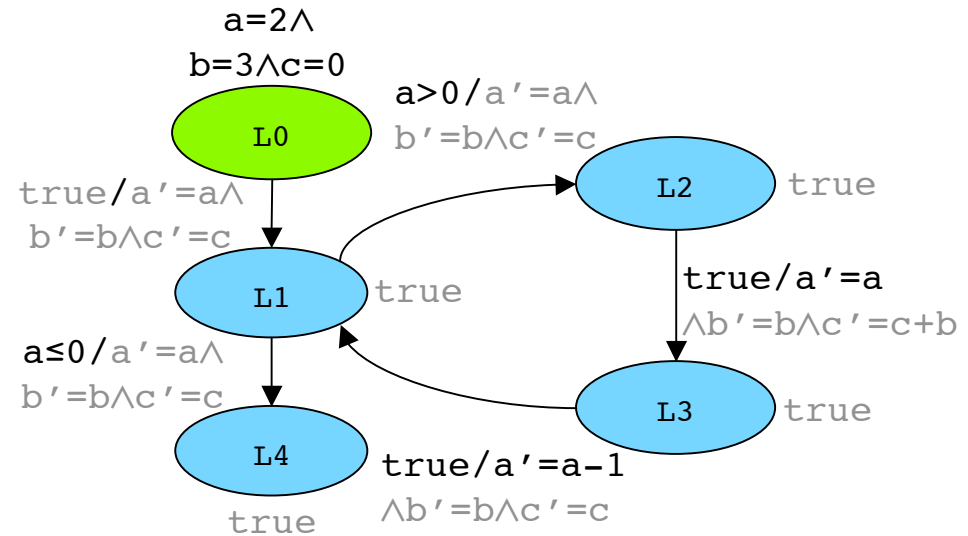
Example:

$$L2 \xrightarrow{a>0/a=a-1} L3 \equiv \{((L2, \{a=1, b=0, c=0\}), (L3, \{a=0, b=0, c=0\})), ((L2, \{a=1, b=0, c=1\}), (L3, \{a=0, b=0, c=1\})) \dots\}$$



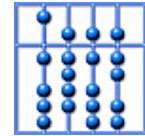
Notation: Extended Finite Automata

```
L0:  int a=2,b=3,c=0;
L1:  while(a>0) do {
L2:      c = c + b;
L3:      a = a - 1; }
L4:
```

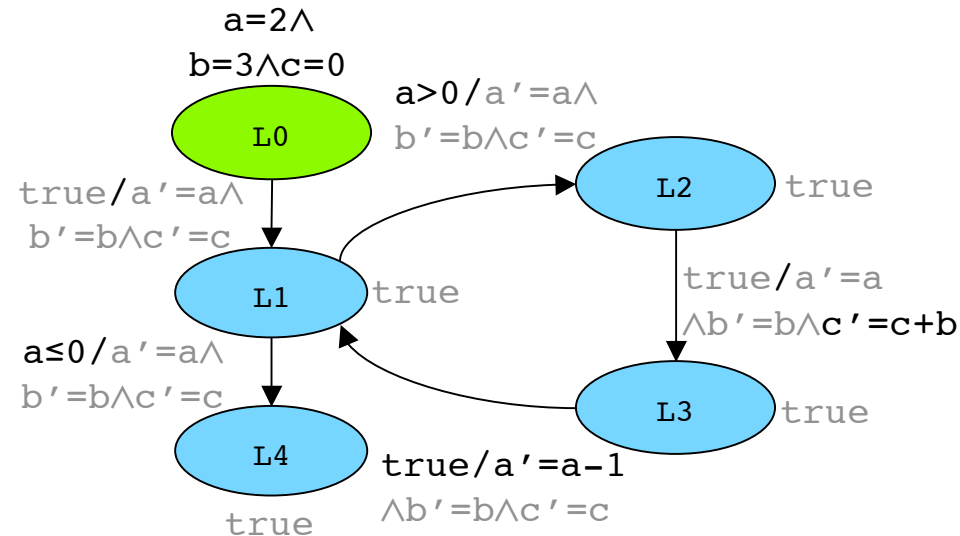


Notation: **Extended Finite Automaton** (N, N_0, E) over a set of variables X :

- Nodes N (finite): Marked with
 - (Unique) Locations L
 - State invariants $Inv(X)$
- Initial nodes $N_0 \subseteq N$
- Edges $E \subseteq N \times N$: Marked with
 - Pre-condition $Pre(X)$
 - Post-condition $Post(X, X')$

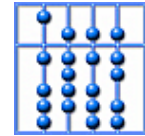


Notation: Extended Finite Automata

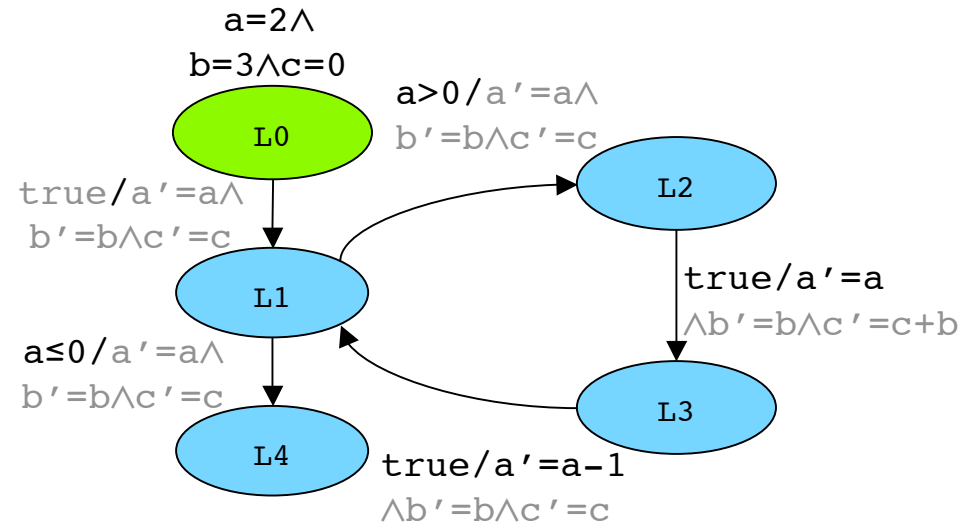


Example:

- $N: \{(L0, a=2 \wedge b=3 \wedge c=0), (L1, true), (L2, true), (L3, true), (L4, true)\}$
- $N_0 : \{(L0, a=2 \wedge b=3 \wedge c=0)\}$
- $E: (L0, true/a'=a \wedge b'=b \wedge c'=c, L1), (L1, a>0/a'=a \wedge b'=b \wedge c'=c, L2), (L2, true/a'=a \wedge b'=b \wedge c'=c+b, L3), (L3, true/a'=a-1 \wedge b'=b \wedge c'=c, L4), (L1, a \leq 0/a'=a \wedge b'=b \wedge c'=c, L4)\}$

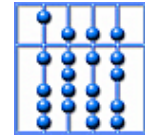


Model: Extended Transition System

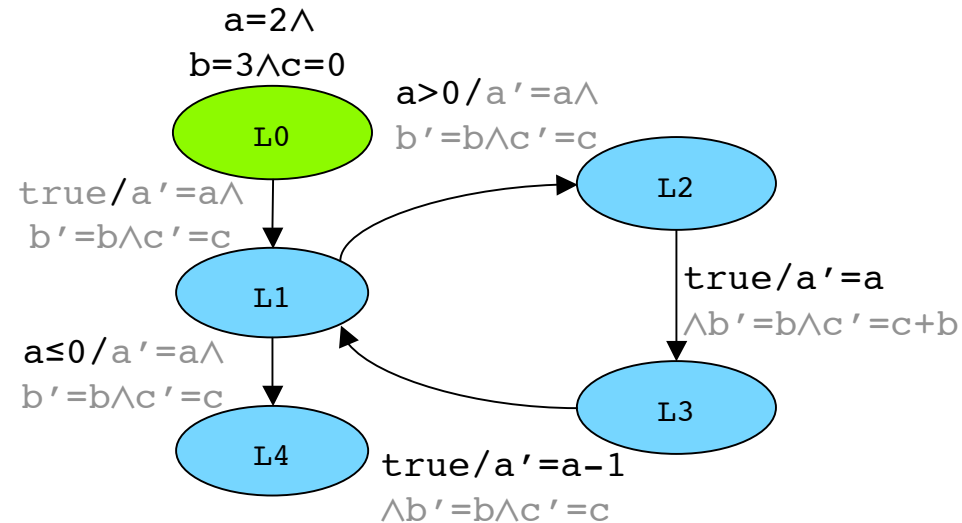


Interpretation: **Extended Transition System** (S, S_0, T) over variables X :

- S : Collection over all nodes $(l, \text{Inv}(X))$:
 $\{(l, v) \mid v \in \text{Inv}(X)\}$
- S_0 : Collection over all initial nodes $(l, \text{Inv}(X))$:
 $\{(l, v) \mid v \in \text{Inv}(X)\}$
- T : Collection over all edges $((l, \text{Inv}(X)), \text{Pre}(X)/\text{Post}(X'), (l', \text{Inv}'(X)))$:
 $\{((l, v), (l', v')) \mid v \in \text{Inv}(X) \wedge v' \in \text{Inv}'(X) \wedge v \in \text{Pre}(X) \wedge v' \in \text{Pre}(X') \wedge \text{Post}(X, X')\}$

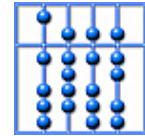


Model: Extended Transition System

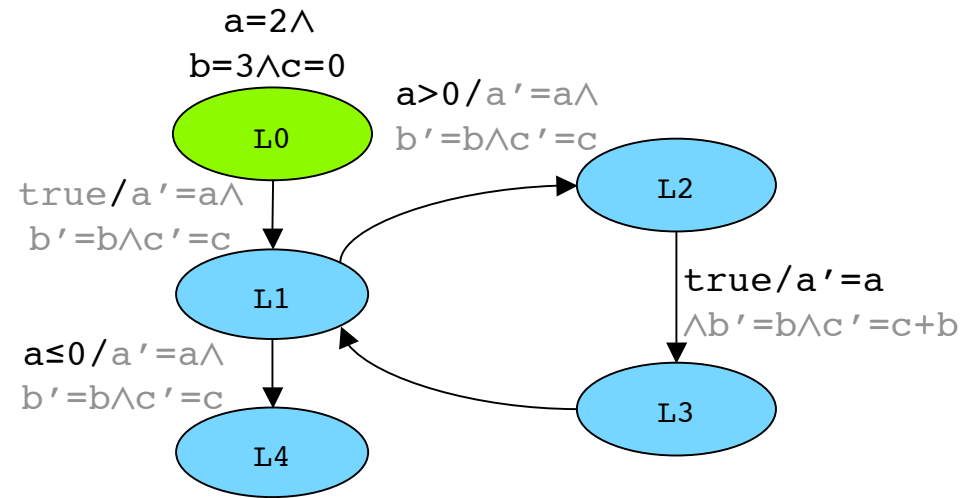


Example: Extended Transition System (S, S_0, T) over variables a, b , and c :

- S : Collection over $(L2, \text{true})$:
 $\{(L2, (a=1, b=0, c=0)), (L2, (a=1, b=1, c=0)), \dots\}$
- S_0 : Collection over $(L0, a=2 \wedge b=3 \wedge c=0)$:
 $\{(L2, (a=2, b=3, c=0))\}$
- T : Collection over all edges $(L1, a > 0 / a' = a \wedge b' = b \wedge c' = c, L2)$:
 $\{(L1, (a=2, b=3, c=0), L2, (a=2, b=3, c=0)),$
 $(L1, (a=1, b=3, c=3), L2, (a=1, b=3, c=3)), \dots\}$



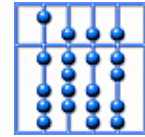
3.1 Summary: Modeling Complex Computations



Concepts:

- State: Observation control and data aspects of a system at a specific instance of time
- Invariant: Collection of data states (of a control location)
- Extended Transition relation: Set of possible actions of a computation characterized by pre- and postcondition

Model: Extended Transition System (S, S_0, T)



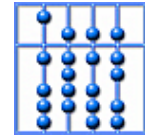
3.2 Shared Memory: Composing Extended Transition Systems

Goal: Define a model to describe the behavior of concurrent computations with shared variables

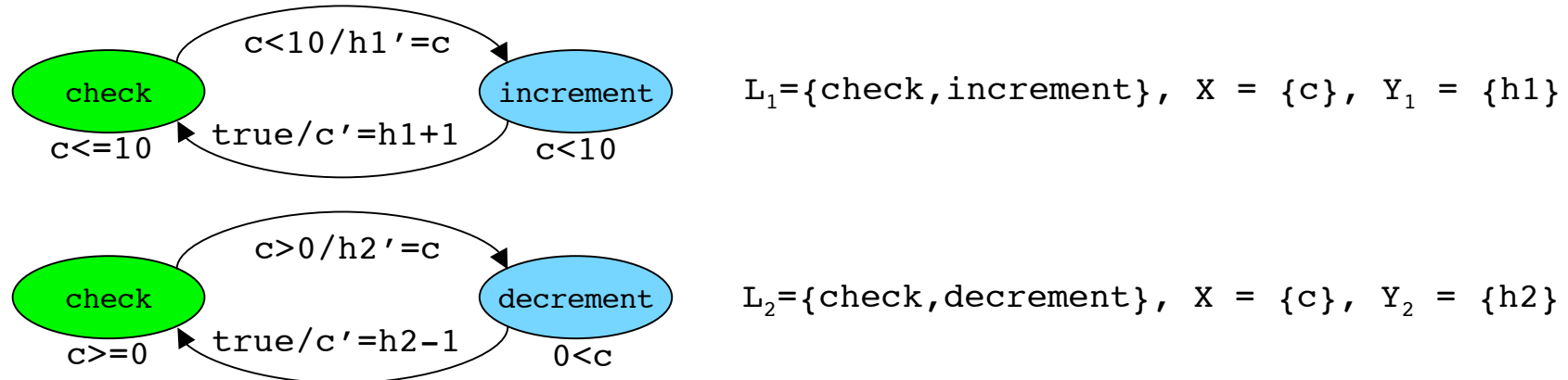
- Relevant: Address aspects evolving by implicit interaction
- General: Cover aspects independent of specific programming languages
- Abstract: Ignore aspects like execution time, memory limitations

Concept: Shared memory, joint execution

Model: Composed Extended State Transition Systems



Concept: Concurrent Execution

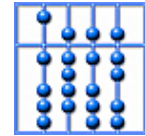


Purpose: Describing concurrent executions of extended transition systems

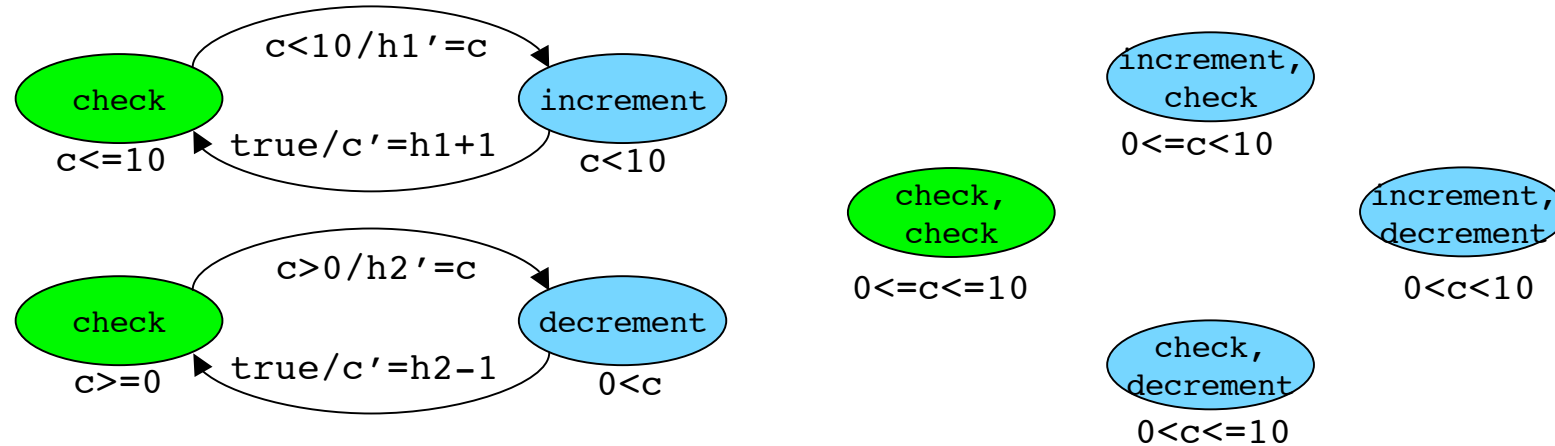
- Independent control space L_1 and L_2
- Joint data space on shared variables X
- Independent data space on local variables Y_1 and Y_2

Concept: States of $(S_1, S_{1,0}, T_1)$ and $(S_2, S_{2,0}, T_2)$ executed in parallel

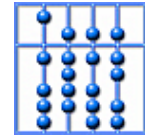
- Combination of all states of each ETS:
 $\{(l_1, l_2, v, w_1, w_2) \mid (l_1, v, w_1) \in S_1 \text{ and } (l_2, v, w_2) \in S_2\}$
- Initial states: $\{(l_1, l_2, v, w_1, w_2) \mid (l_1, v, w_1) \in S_{1,0} \text{ and } (l_2, v, w_2) \in S_{2,0}\}$



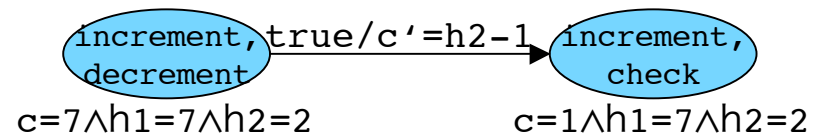
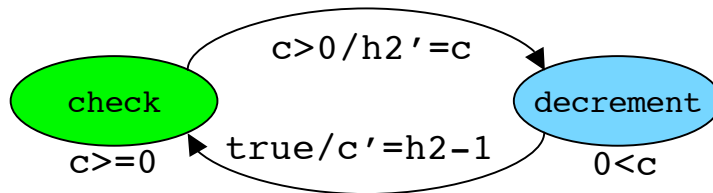
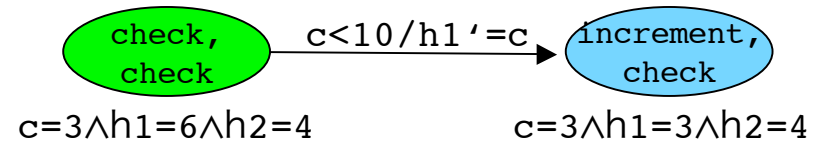
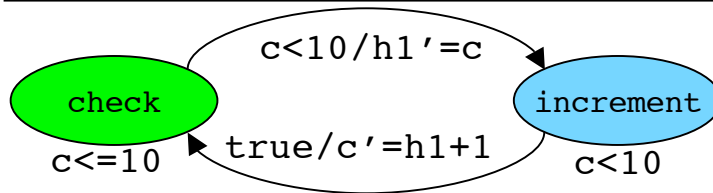
Concept: Concurrent Execution



- Example: $S = \{(\text{check}, c, h1) \mid c \leq 10\} \cup \{(\text{increment}, c, h1) \mid c < 10\}$,
 $S_0 = \{(\text{check}, c, h1) \mid c \leq 10\}$,
 $S' = \{(\text{check}, c, h2) \mid c \geq 0\} \cup \{(\text{decrement}, c, h2) \mid c > 0\}$,
 $S'_0 = \{(\text{check}, c, h2) \mid c \geq 0\}$
- Product state: $\{(\text{check}, \text{check}, c, h1, h2) \mid 0 \leq c \leq 10\} \cup$
 $\{(\text{increment}, \text{check}, c, h1, h2) \mid 0 \leq c < 10\} \cup$
 $\{(\text{check}, \text{decrement}, c, h1, h2) \mid 0 < c \leq 10\} \cup$
 $\{(\text{increment}, \text{decrement}, c, h1, h2) \mid 0 < c < 10\}$
- Initial state = $\{(\text{check}, \text{check}, c, h1, h2) \mid 0 \leq c \leq 10\}$



Concept: Independent Interaction



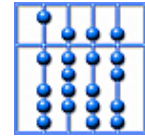
Purpose: Describing independent interactions of systems, **influencing only the state of the executing process**

Concept: Transitions of either $(S_1, S_{1,0}, T_1)$ or $(S_2, S_{2,0}, T_2)$

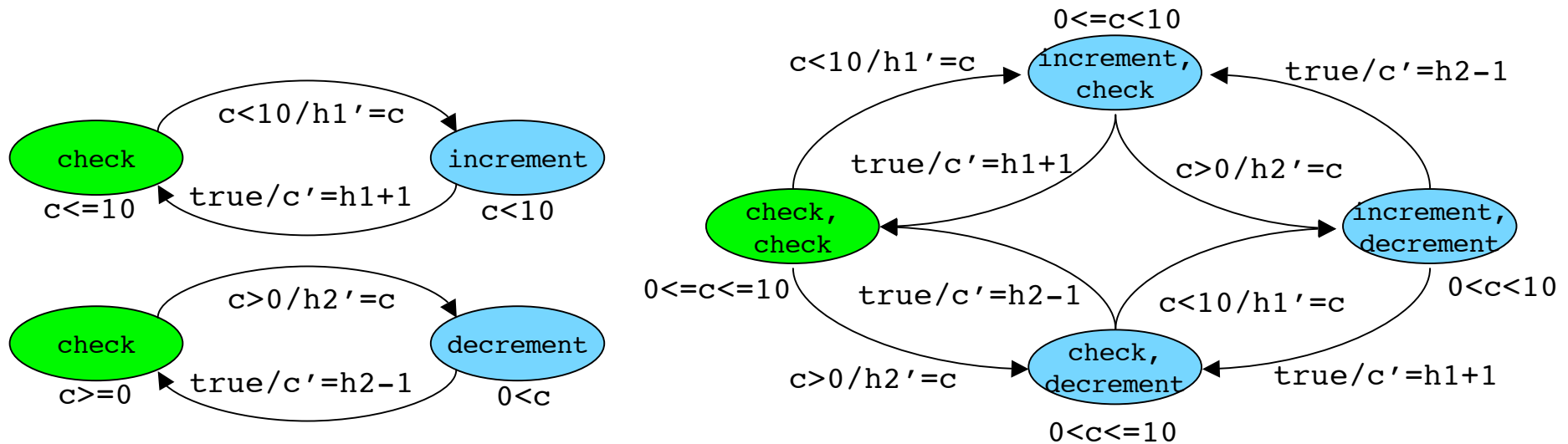
- $((l_1, l_2, v, w_1, w_2), (l'_1, l_2, v', w'_1, w'_2))$ for each $((l_1, v, w_1), (l'_1, v', w'_1)) \in T_1$
- $((l_1, l_2, v, w_1, w_2), (l_1, l'_2, v', w'_1, w'_2))$ for each $((l_2, v, w_2), (l'_2, v', w'_2)) \in T_2$

Example:

- $((\text{check}, \text{check}, 3, 6, 4), (\text{increment}, \text{check}, 3, 3, 4))$
- $((\text{increment}, \text{decrement}, 7, 7, 2), (\text{increment}, \text{check}, 1, 7, 2))$



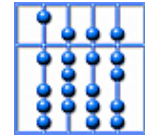
Model: Product Automaton



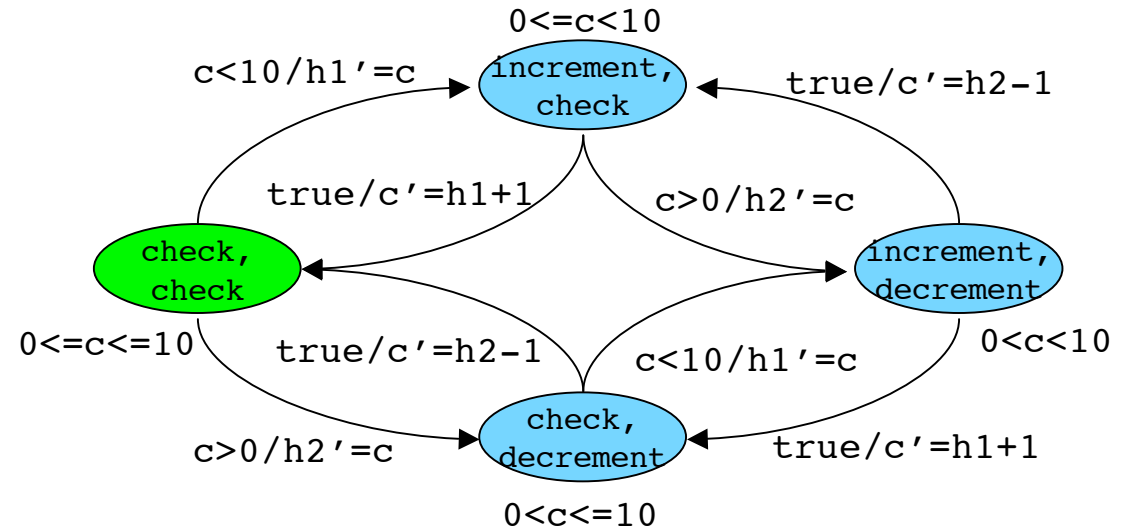
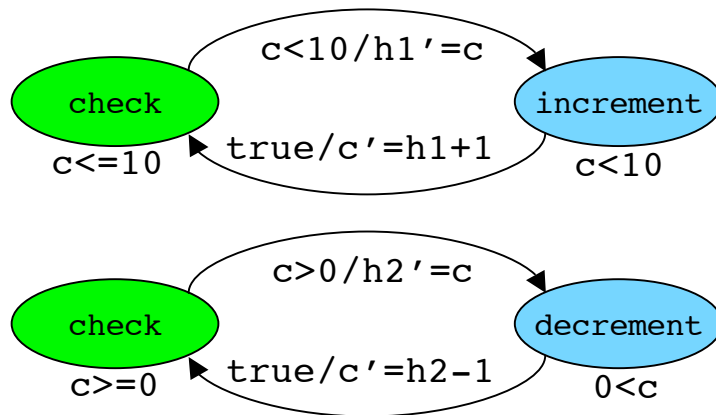
Purpose: Describing concurrent execution with independent interactions

Model: Product Transition System of $(S_1, S_{1,0}, T_1)$ and $(S_2, S_{2,0}, T_2)$

- States: $\{(l_1, l_2, v, w_1, w_2) \mid (l_1, v, w_1) \in S_1 \text{ and } (l_2, v, w_2) \in S_1\}$
- Initial states: $\{(l_1, l_2, v, w_1, w_2) \mid (l_1, v, w_1) \in S_{1,0} \text{ and } (l_2, v, w_2) \in S_{2,0}\}$
- Transition relation:
 - $((l_1, l_2, v, w_1, w_2), (l'_1, l_2, v', w'_1, w_2))$ for each $((l_1, v, w_1), (l'_1, v', w'_1)) \in T_1$
 - $((l_1, l_2, v, w_1, w_2), (l_1, l'_2, v', w_1, w'_2))$ for each $((l_2, v, w_2), (l'_2, v', w'_2)) \in T_2$



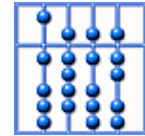
3.2 Summary: Modeling Shared Memory



Concepts:

- Shared Memory: Product space with joint global state and independent local and control state
- Implicit interaction: Interleaved execution of interactions

Model: Combined Extended Transition System



2.3 Application: Threads

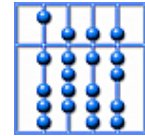
Paradigm: “Leight-weight” process

- Independent control space: Separate computation of each processes
 - Independent PC, local variables (registers)
 - Independent stack
- Shared data space: Shared global variables

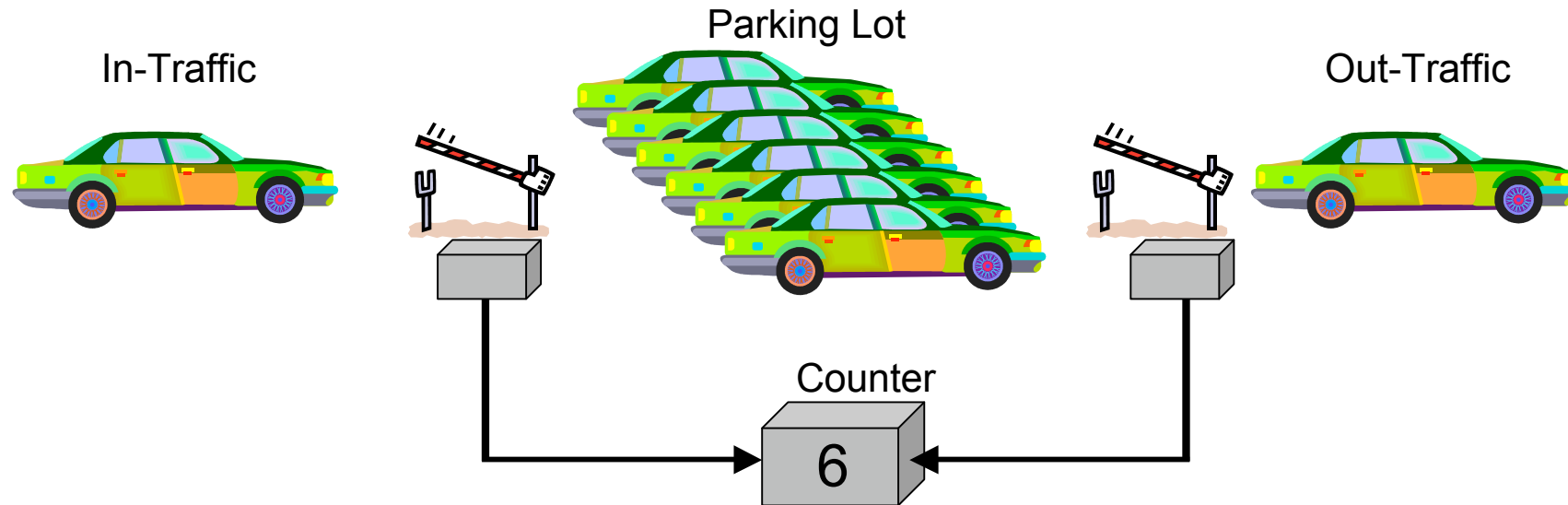
Purpose: Fast processes with minimum overhead

- Minimum space: Shared data space
- Minimum time: Simple interaction mechanism

Examples: Java threads, ANSI processes



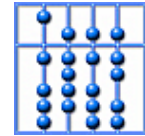
Example: Parking Lot



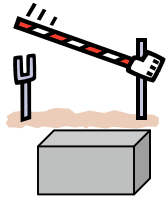
Parking Lot: Limited capacity of cars

- Current capacity of cars measured by counter
- In-bound traffic decrements counter
- Out-bound traffic increments counter

- Parallel implementation: entry and exit are controlled by separate processes



Example: Java Implementation



```
class InGate extends Thread {
    ...
    Counter capacity;

    InGate(Counter c)
    { capacity = c; }

    public void run() {

        ...
        while (true){
            ...
            capacity.decrement();
            ...
        }
    }
}
```

```
class OutGate extends Thread {
    ...
    Counter capacity;

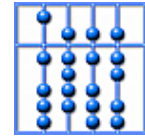
    OutGate(Counter c)
    { capacity = c; }

    public void run() {

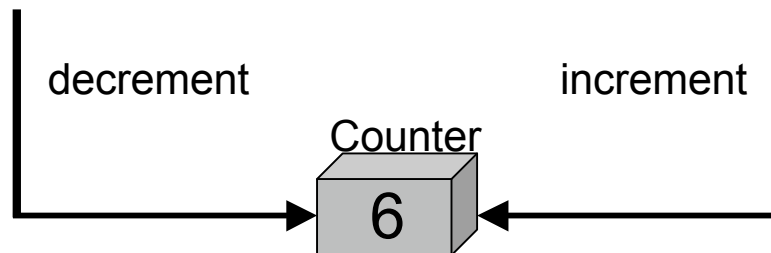
        ...
        while (true){
            ...
            capacity.increment();
            ...
        }
    }
}
```

Classes InGate, OutGate: Concurrent threads

- Using a (shared) counter object to measure capacity
- InGate: decrementing counter
- OutGate: incrementing counter



Example: Java Implementation



```
class Counter {
    int amount = 0;

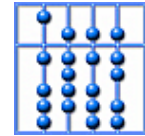
    Counter(int a)
    { amount = a; }

    void increment() {
        this.amount = this.amount + 1;
    }

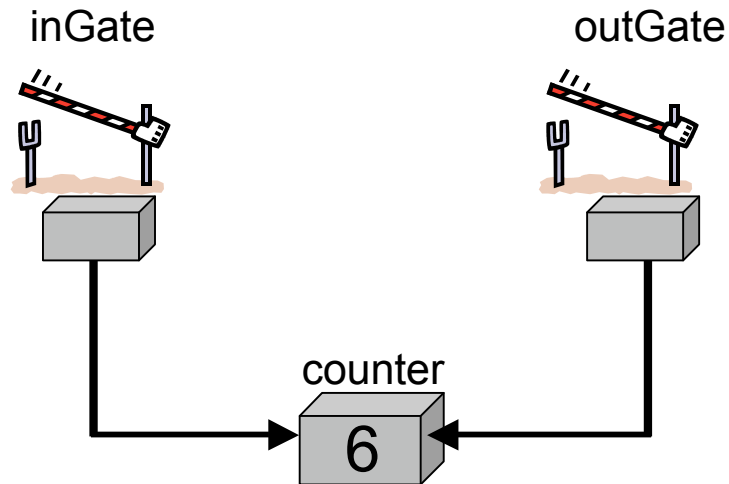
    void decrement() {
        if(this.amount > 0)
            this.amount = this.amount - 1;
    }
}
```

Class Counter: Passive object

- Measuring capacity
- Offering increment/decrement methods
- Used by InGate/OutGate threads



Example: Parking Lot



```
// Embedding into an applet
...

private void init() {

    counter = new Counter(100);
    inGate = new Gate(counter);
    outGate = new Gate(counter);

    inGate.start();
    outGate.start();

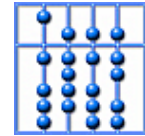
}

...
```

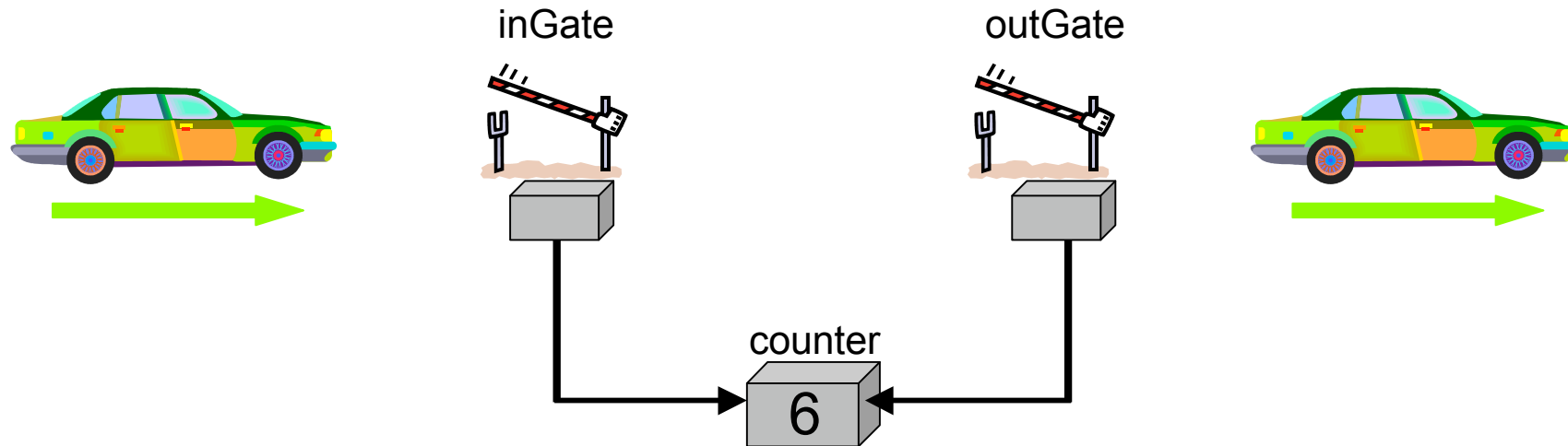
Parking Lot: Limited capacity of cars

- Current capacity of cars measured by counter
- In-bound traffic decrements counter
- Out-bound traffic increments counter

- Parallel implementation: entry and exit are controlled by separate processes

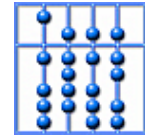


Example: Parking Lot



What happens if

- cars pass simultaneously through the entry and the exit gate
- and therefore simultaneously
 - inGate executes `counter.increment()`
 - outGate executes `counter.decrement()`leading to a concurrent change of the counter?



Example: Formalization

```
class InGate extends Thread {
  ...
  Counter capacity;

  InGate(Counter c)
  { capacity = c; }

  public void run() {
    ...
    while (true){
      ...
      capacity.decrement();
      ...
    }
  }
}
```

```
class Counter {
  int amount = 0;

  Counter(int a)
  { amount = a; }

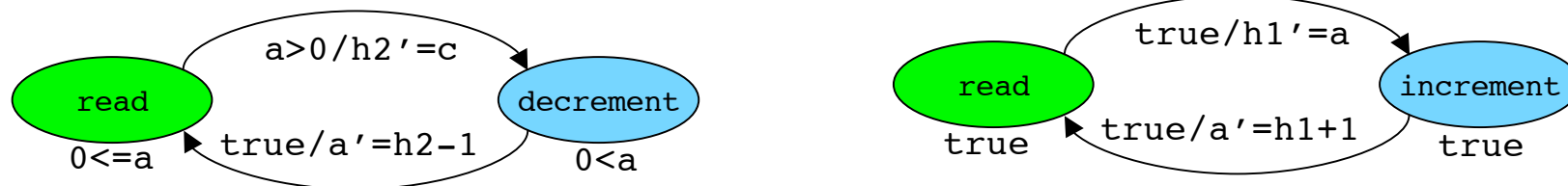
  void increment() {
    this.amount = this.amount + 1;
  }

  void decrement() {
    if(this.amount > 0)
      this.amount = this.amount -
1;
  }
}
```

```
class OutGate extends Thread {
  ...
  Counter capacity;

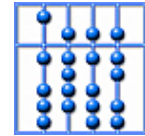
  OutGate(Counter c)
  { capacity = c; }

  public void run() {
    ...
    while (true){
      ...
      capacity.increment();
      ...
    }
  }
}
```

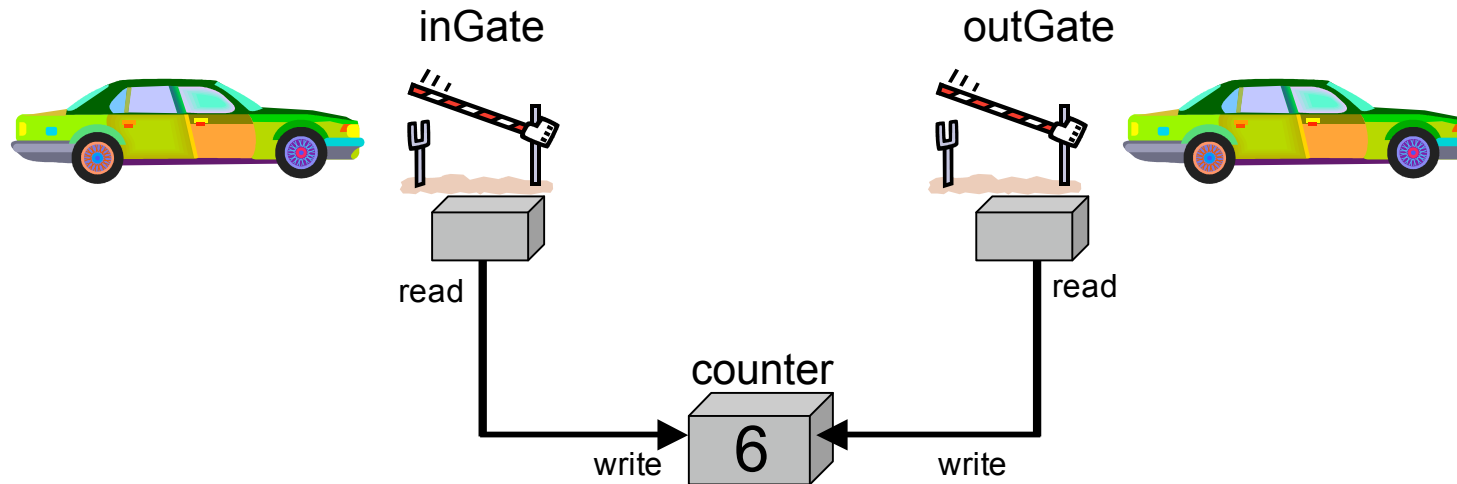


Java Virtual Machine

- Threads correspond to unsynchronized processes
- Shared objects correspond to shared variables, method variables to local variables
- Addition/subtraction of class variables are not atomic actions

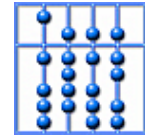


Example: Parking Lot

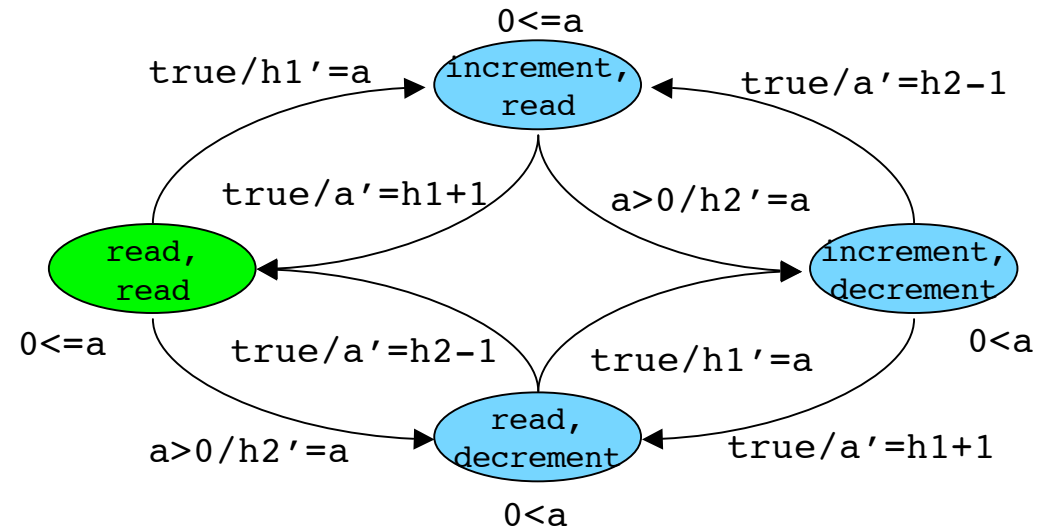
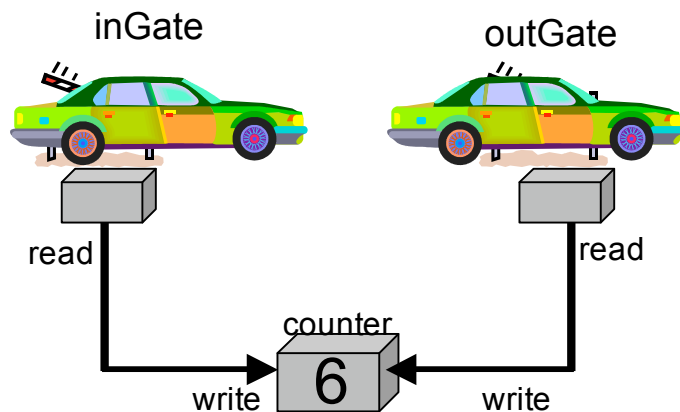


Execution trace of product automaton for situation

- Capacity.amount = 6
- inGate at capacity.decrement()
- immediately followed by
- outGate at capacity.increment()

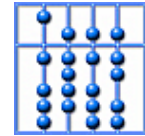


Example: Parking Lot



Execution traces of product automaton for situation (r=read,i=increment,d=decrement)

- Variant (r,r) → (i,r) → (r,r) → (r,d) → (r,r): capacity.amount = 6
- Variant (r,r) → (r,d) → (r,r) → (i,r) → (r,r): capacity.amount = 6
- Variant (r,r) → (i,r) → (i,d) → (r,d) → (r,r): capacity.amount = 5
- Variant (r,r) → (i,r) → (i,d) → (i,r) → (r,r): capacity.amount = 7
- Variant (r,r) → (r,d) → (i,d) → (r,d) → (r,r): capacity.amount = 5
- Variant (r,r) → (r,d) → (i,d) → (i,r) → (r,r): capacity.amount = 7



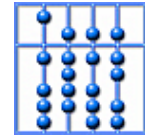
2.3 Issues: Interference and Race Conditions

Co-routines: Simple extension of imperative languages

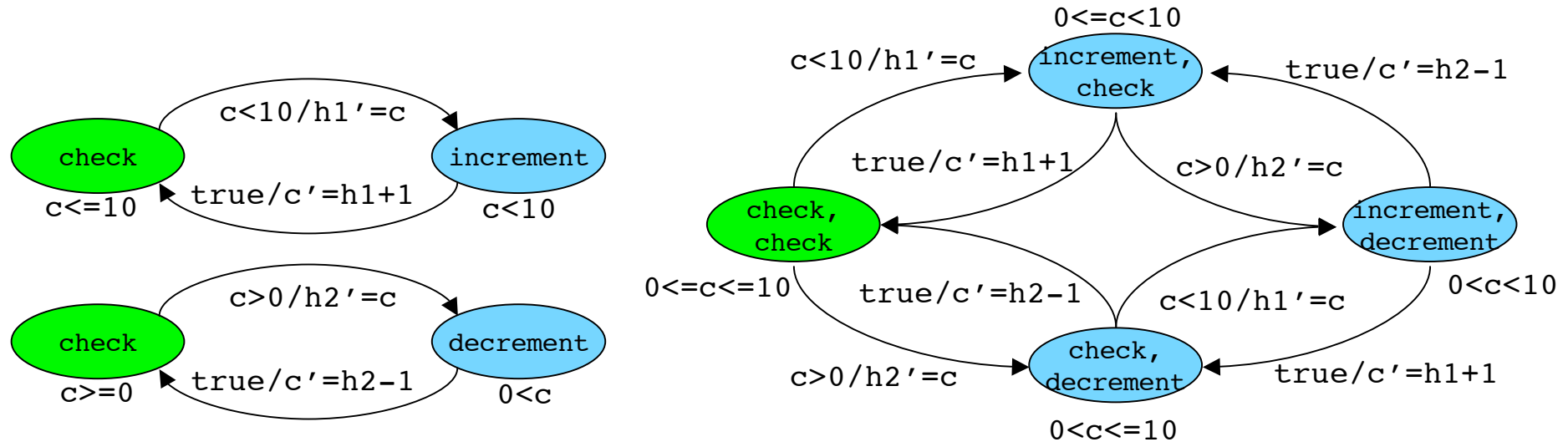
- Single process: Standard imperative program with control and data space
- Multi-process: Parallel composition of processes with shared data space

Weakness: Implicit interaction in co-routines

Issues: Interference, race condition



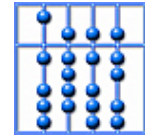
Issue: Interference



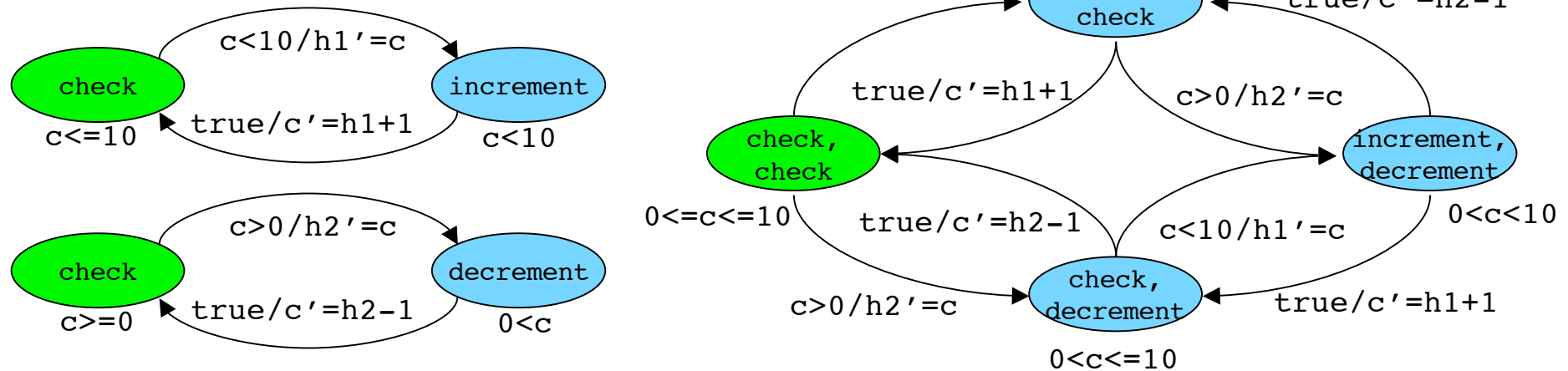
Problem: Difference between isolated and concurrent execution of a routine

Example: Observation traces

- Incrementing process in isolation:
(check, 7, 0) • (increment, 7, 7) • (check, 8, 7) •
- Incrementing process in combination:
(check, check, 7, 0, 0) • (increment, decrement, 7, 7, 7) • (check, decrement, 8, 7, 7) •
(check, check, 6, 7, 7) •



Issue: Interference



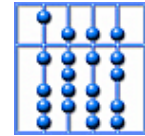
Notion: **Interference** between co-routines of a system

An observation made about a co-routine in isolation does no longer hold if the co-routine is executed in parallel with an interfering co-routine

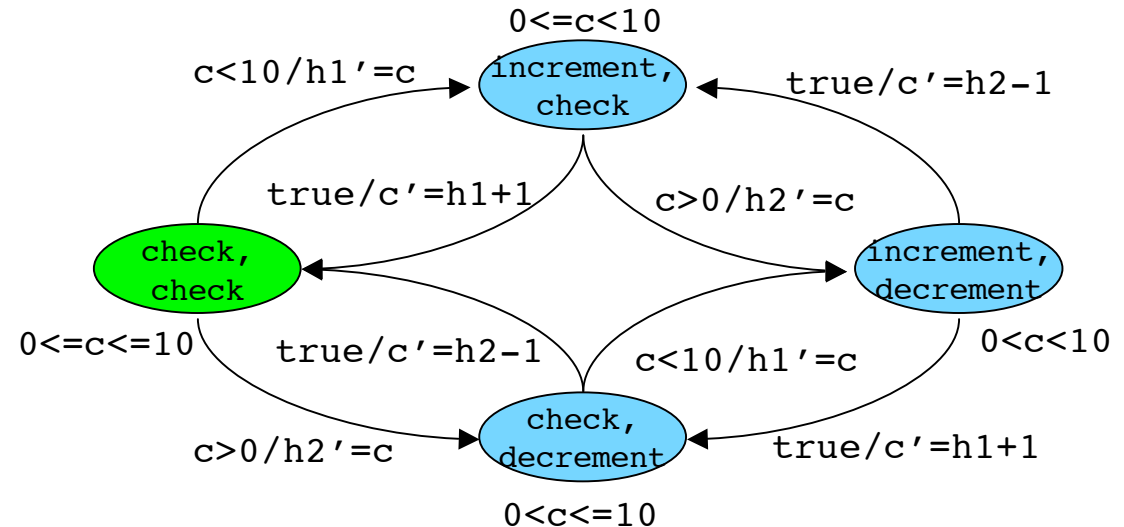
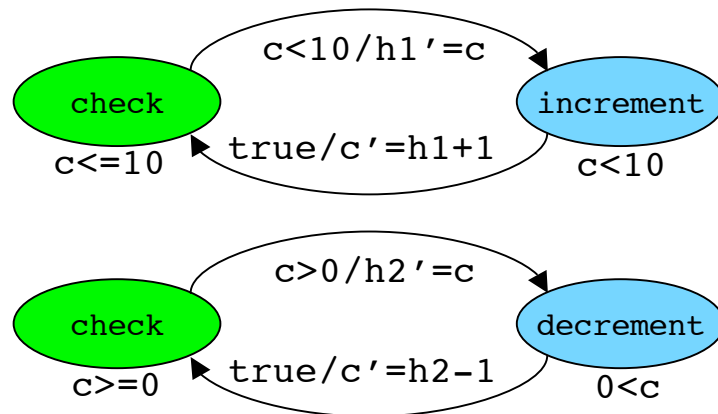
Example: Observation of increment routine: “The value of the counter variable increases.”

Relevance: Construction of a system

- Component developed and successfully conformance-tested in isolation
- Component not conformant when integrated into system



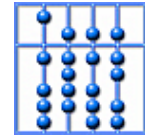
Issue: Race Condition



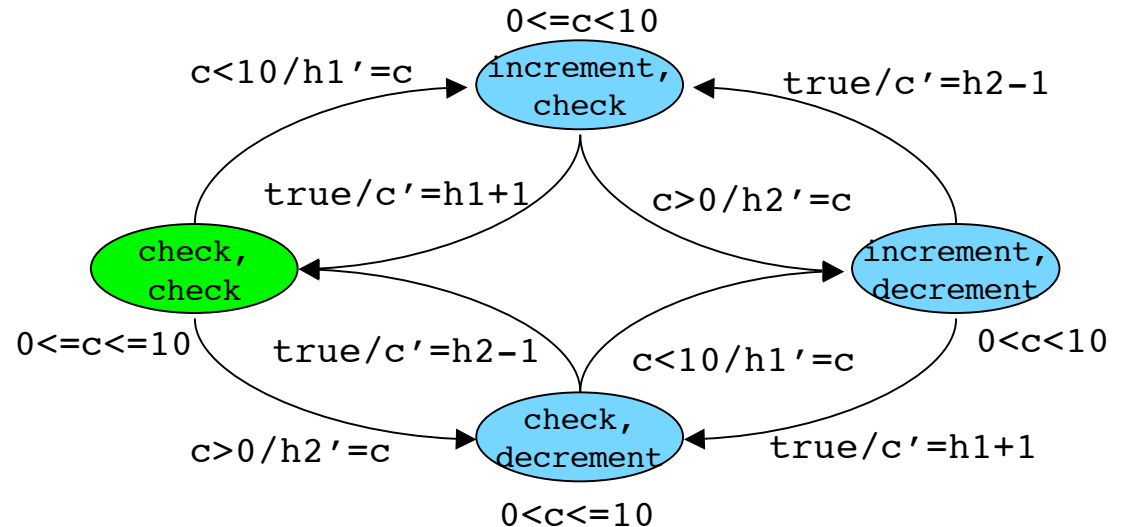
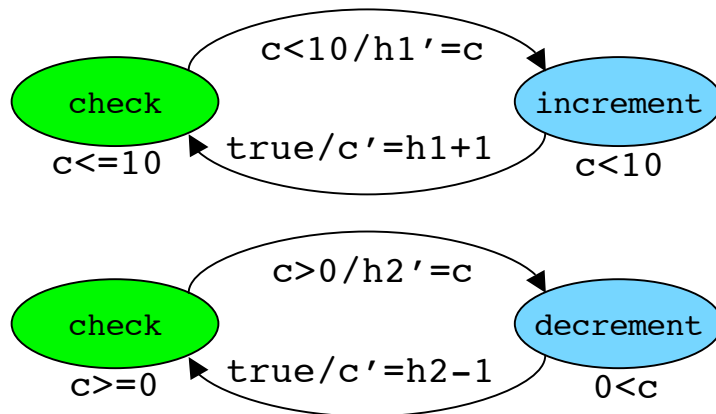
Problem: Different results depending of the relative speed of co-routines

Example: Observation traces of system of combined co-routines, each routine performing two actions

- Incrementing process is faster process:
(check, check, 7, 0, 0) • (increment, decrement, 7, 7, 7) •
(check, decrement, 8, 7, 7) • (check, check, 6, 7, 7) •
- Decrementing process in faster process:
(check, check, 7, 0, 0) • (increment, decrement, 7, 7, 7) •
(check, increment, 6, 7, 7) • (check, check, 8, 7, 7) •



Issue: Race condition



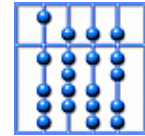
Notion: **Race condition** during execution of co-routines of a system

The results of a sequence of actions of a set of co-routines depends on the ordering of the sequence (i.e. the relative speed of the routines) independent of the environment

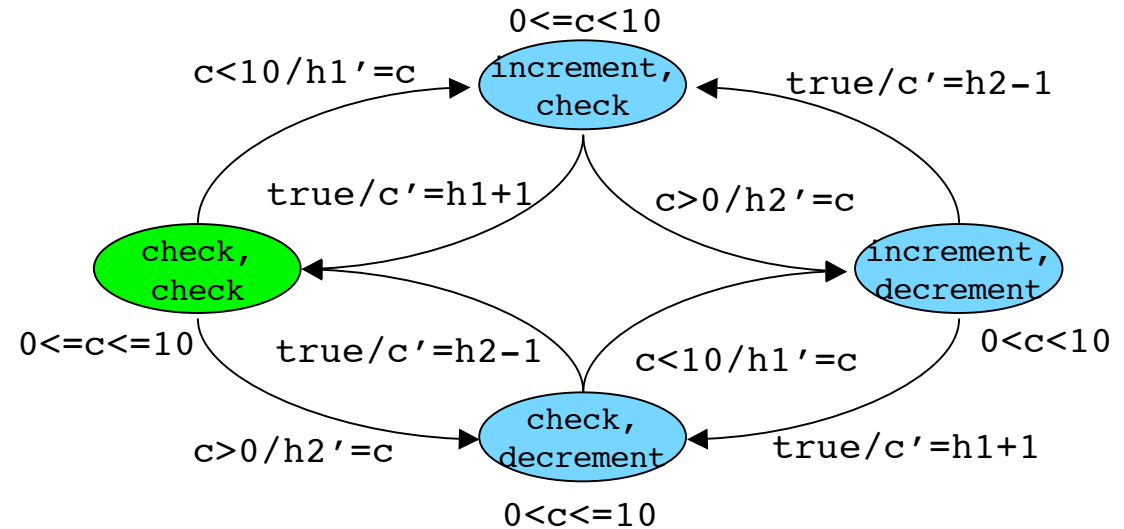
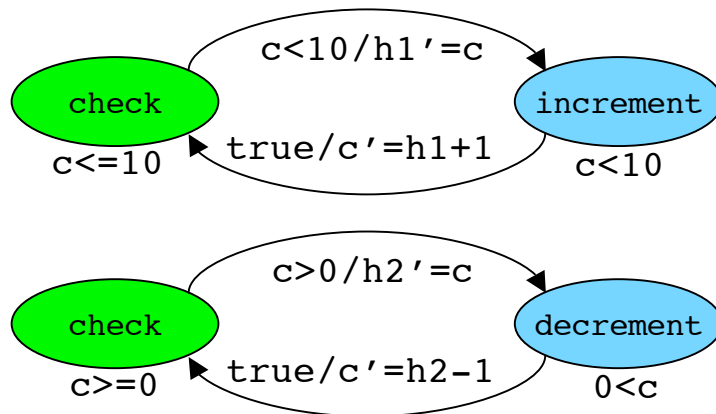
Example: “The value of the counter variable increases.”

Relevance: Implementation of a system

- Implementation developed and successfully conformance-tested on given platform
- Implementation not conformant when deployed to a different platform



3.4 Summary: Problems of Co-Routines

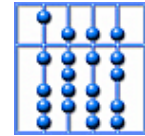


Issues:

- Interference: Observations of a co-routine in isolation do no longer hold when co-routine is composed with interfering co-routine
- Race condition: Observations of co-routines do no longer hold hold when re-ordering internal actions of the system

Consequence: Co-routines are not suitable concerning

- Modular development
- Platform-independent development



3.5 Questions

1. How can a queuing structure for a consumer/producer co-routine system be formalized using an extended transition system?
2. What are the strongest invariants for the states of the queue?
3. Is the model of co-routines compositional?
4. What is the main cause for interference?
5. What is the main cause for race conditions?
6. What mechanism can be used to avoid interference?
7. What mechanism can be used to avoid race conditions?
8. How can extended transition systems be restricted/extended to support a suitable component model based on variable communication?
9. How should a queue look like in the component based approach?
10. What is a characteristic property of the observations of a component using variable-based communication?