# Distributed Computing

José Orlando Pereira

Grupo de Sistemas Distribuídos
Departamento de Informática
Universidade do Minho

2009/2010

# Asynchronous systems

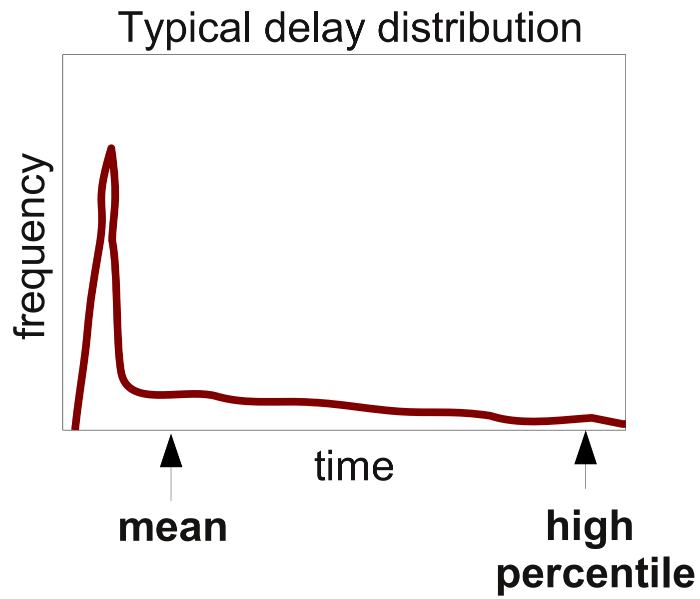
- Assume no bounds on:
  - clock drift
  - processing time
  - message passing time
- Motivated by real world considerations:
  - Load and processor scheduling
  - Network delays
  - ...

# Asynchronous systems

- Without loss of generality, assume a reliable fully connected network
- Relax the synchronous system:
  - Unbounded message loss
  - Large/unknown graph diameter
  - Dynamic graph
- Each of the resulting models is equivalent to an asynchronous system:
  - The universal system model

# Asynchronous systems

- Tight synchronous limits are dangerous:
  - Low coverage, expensive systems
- Large synchronous limits are not useful:
  - Round time proportional to high percentile delay
  - Taking advantage of synchrony causes a very large penalty
- Solutions for asynchronous systems have better performance:
  - Round time proportional to mean delay



# I/O Automata

- Very general model:
  - Describes also non-distributed and even non-concurrent systems
- Powerful tools:
  - Composable specifications
  - Hierarchical specifications
- Very widespread use in DS research

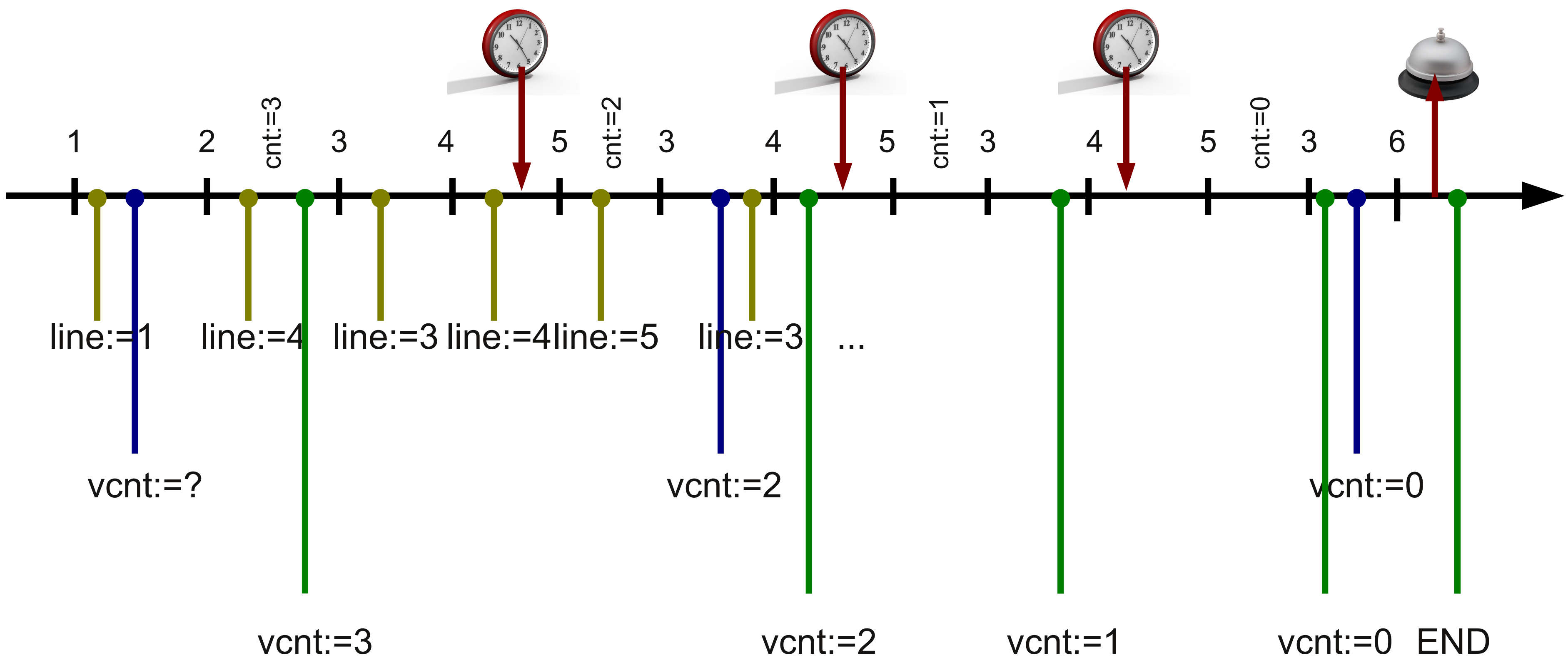
# Sample computation

- An alarm clock program:

```
main:                 // line 1
    cnt:=3            // line 2
    while cnt>0:      // line 3
        sleep 1s      // line 4
        cnt := cnt-1  // line 5
    ring              // line 6
```

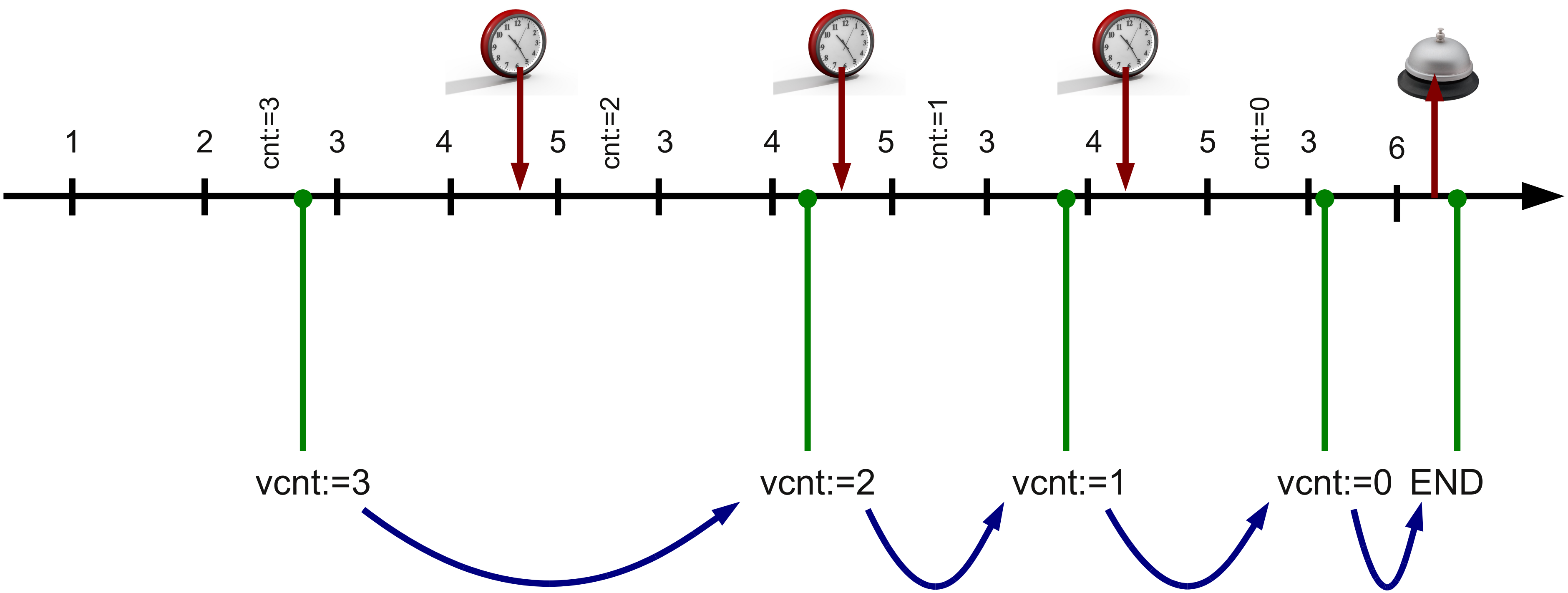
# Observation

- Select model variables and periodically observe the system:



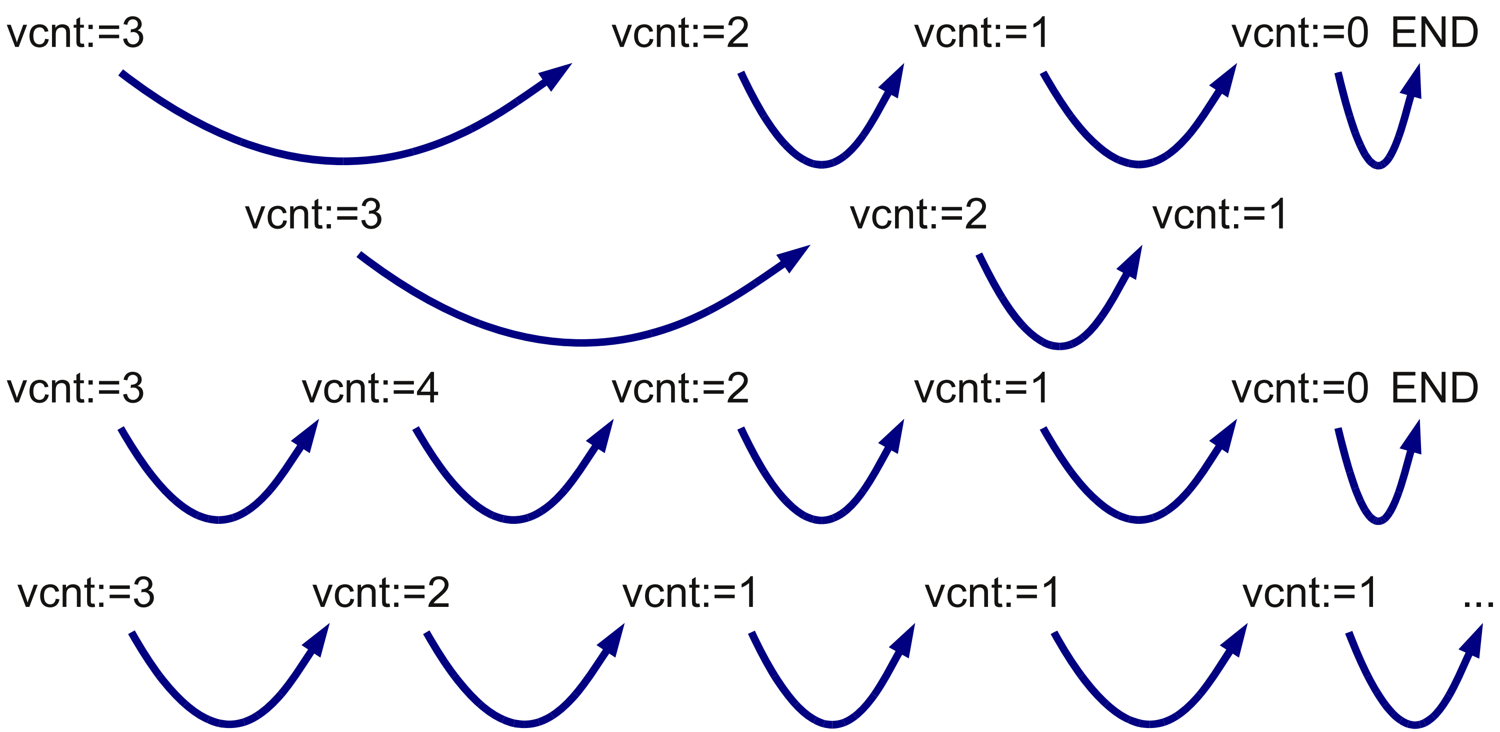
# Abstraction

- Choose observation that conveys interface, not implementation:



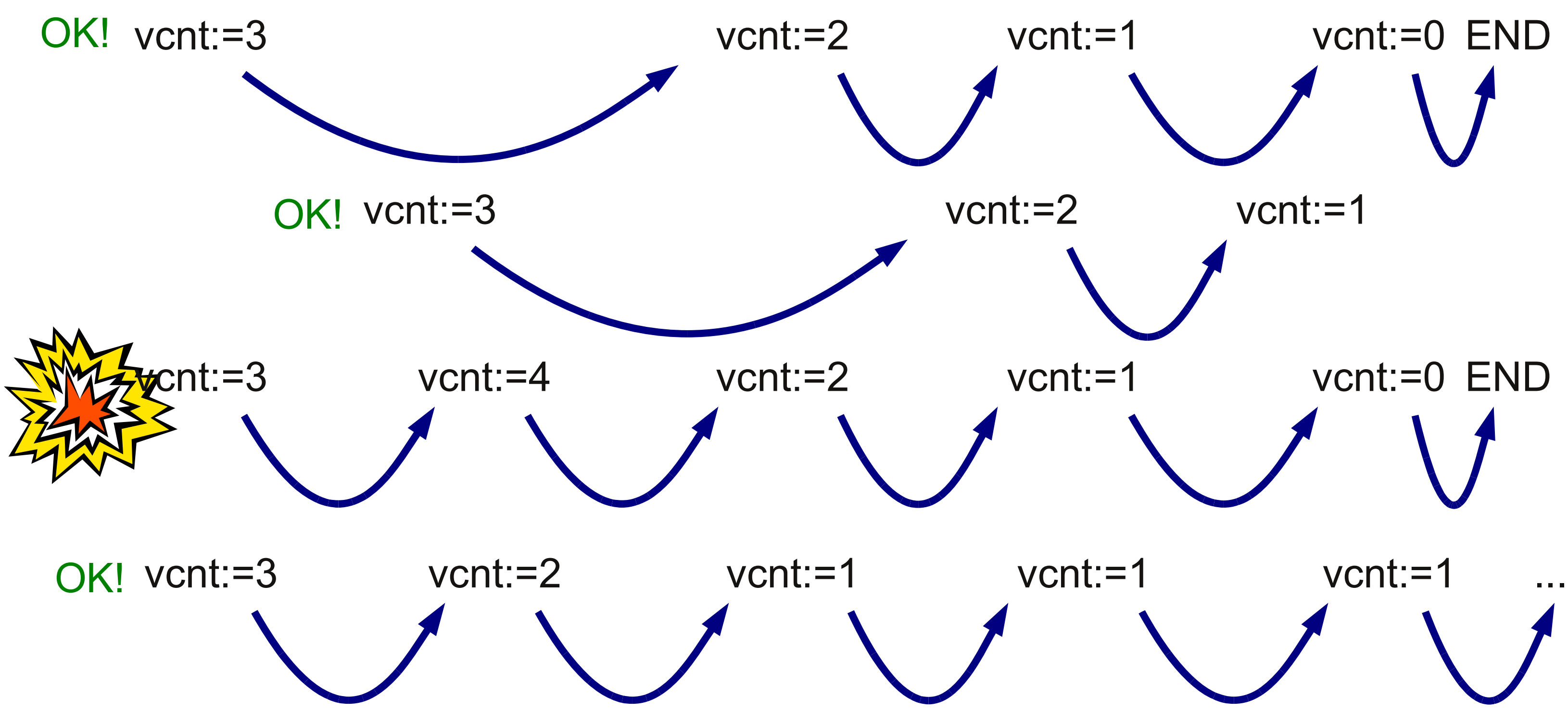
# Behaviors/Executions

- Consider all possible sequences of chosen atomic actions:



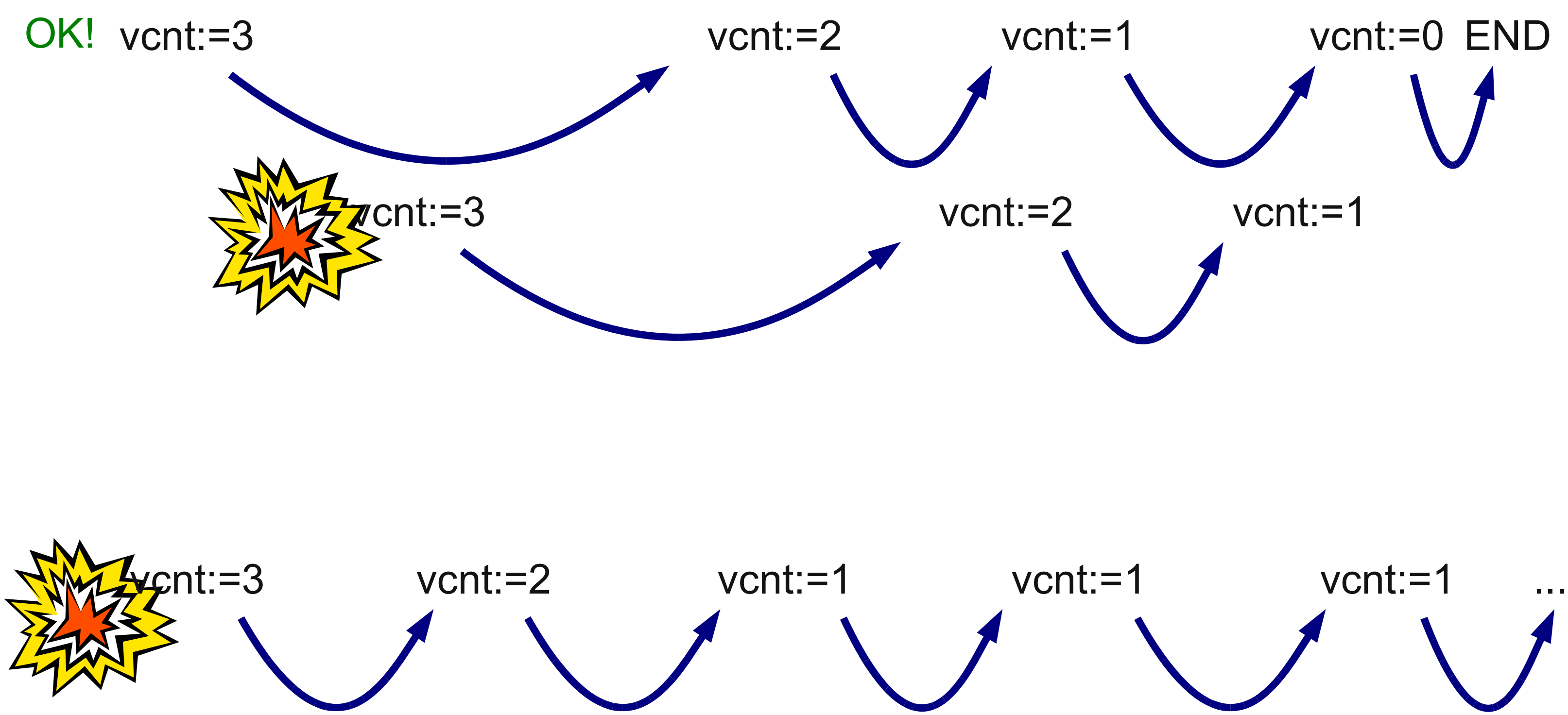
# Safety properties

- Nothing bad ever happens:



# Liveness properties

- Something good eventually[(*)] happens:




(*) eventually = inevitavelmente ≠ eventualmente

# Specification

- Specification is a set of allowable behaviors:

$$S=\left\{ vcnt{:=}3 \xrightarrow{\text{timeout}} vcnt{:=}2 \xrightarrow{\text{timeout}} vcnt{:=}1 \xrightarrow{\text{timeout}} vcnt{:=}0 \xrightarrow{\text{ring}} END \right\}$$

- An automaton provides a compact and practical representation
  - Infinite sets of behaviors

# Automaton definition

- An automaton A has five components:
  - sig(A), a triplet S of disjoint sets of actions:
    - in(S), the input actions
    - out(S), the output actions
    - int(S), the internal actions
  - states(A), a (possibly infinite) set of states
  - start(A), a non-empty subset of states(A)
  - trans(A), a subset of
    states(A) x acts(sig(A)) x states(A)
  - tasks(A), a partition of local(sig(A))

# Automaton definition

- Additional definitions:
  - ext(S) = in(S) U out(S)
  - local(S) = out(S) U int(S)
  - extsig(S) = (in(S), out(S), {})
- Short-hands:
  - ext(A) for ext(sig(A))
  - ...

# Transitions

- A transition is enabled in state s if there is some $\pi$,s' such that $(s,\pi,s') \in trans(A)$
- Input transitions are required to be enabled in all reachable states of A
- A state in which only input transitions are enabled is said to be quiescent

# Signature and State

- Input:
  - none
- Internal:
  - Timeout
- Output:
  - Ring

- States:
  - vcnt, integer, initially 3
  - END, boolean, initially false

# Transitions

- Timeout:
  - Pre-condition:
    - $\neg$END and vcnt>0
  - Effect:
    - vcnt := vcnt - 1
- Ring:
  - Pre-condition:
    - $\neg$END and vcnt = 0
  - Effect:
    - END := True

This is an equation,
not an attribution!

# Effects

- Effect equation:
  - vcnt := vcnt - 1
- Read this as:
  - “vcnt-after = vcnt-before – 1 and the state otherwise unchanged”
- Could be written as:
  - vcnt-after + 1 = vcnt-before
  - vcnt-before - vcnt-after = 1
  - ...

# Invariants

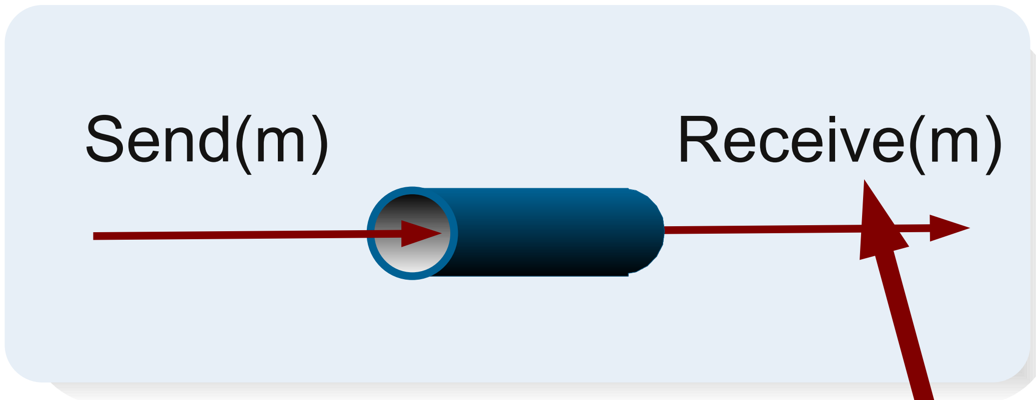
- Goal: Prove that always vcnt < 4 (safety!).
- Proof by induction:
  - Base step: True for all initial states?
    - 3<4: Yes!
  - Induction step: True for any next step?
    - Timeout transition:
      - vcnt-after = vcnt-before - 1
      - vcnt-before < 4<br>vcnt-after+1 < 4<br>vcnt-after < 3 < 4: Done
    - Ring transition:
      - always vcnt-after = vcnt-before = 0
      - 0<4: Done

# Trace properties

- A trace is the externally visible sequence of actions
- A trace property is a set of traces
- Proof strategy:
  - Add the trace as a variable to the state
  - Safety trace properties are then invariant assertions

# Example: Reliable channel



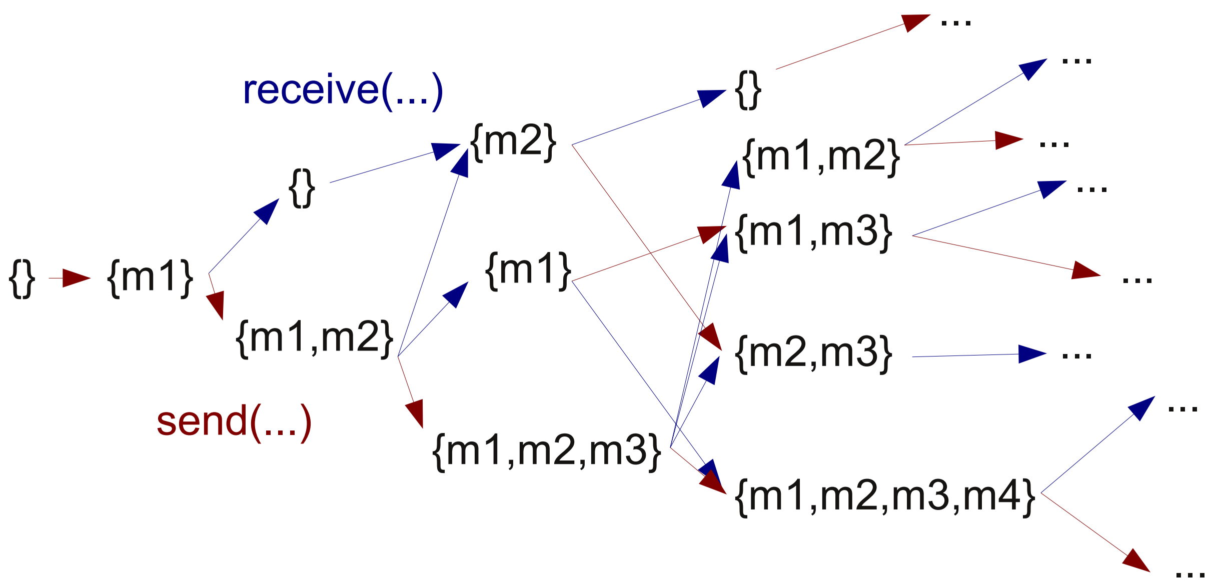

- Reliable channel:
  - Unordered
  - FIFO

Why *Receive(m)* and not *m := Receive()*?

# Example: Reliable channel

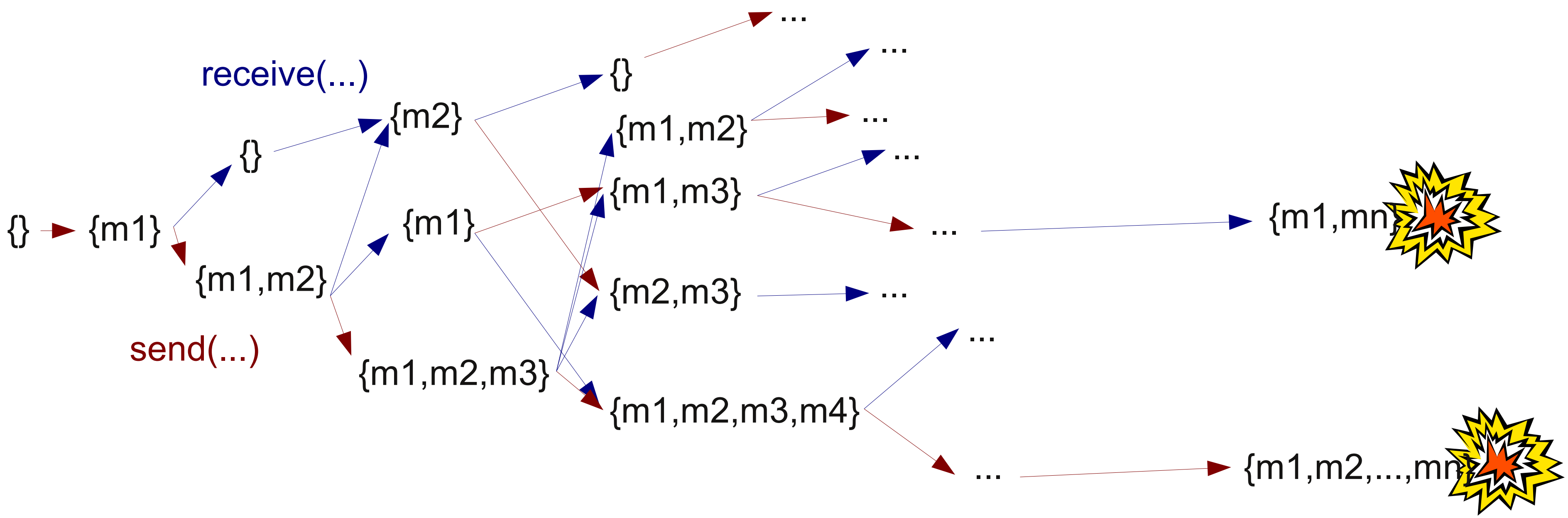
- State:
  - transit, bag of M, initially {}
- Send(m), $m \in M$:
  - Pre-condition:
    - True
  - Effect:
    - transit :=transit + {m}
- Receive(m), $m \in M$:
  - Pre-condition:
    - m in transit
  - Effect:
    - transit := transit - {m}

# Behaviors of a channel




- Concurrency is modeled by alternative enabled transitions:
  - Sender and receiver
  - Within the channel (reordering)

# Liveness and fairness

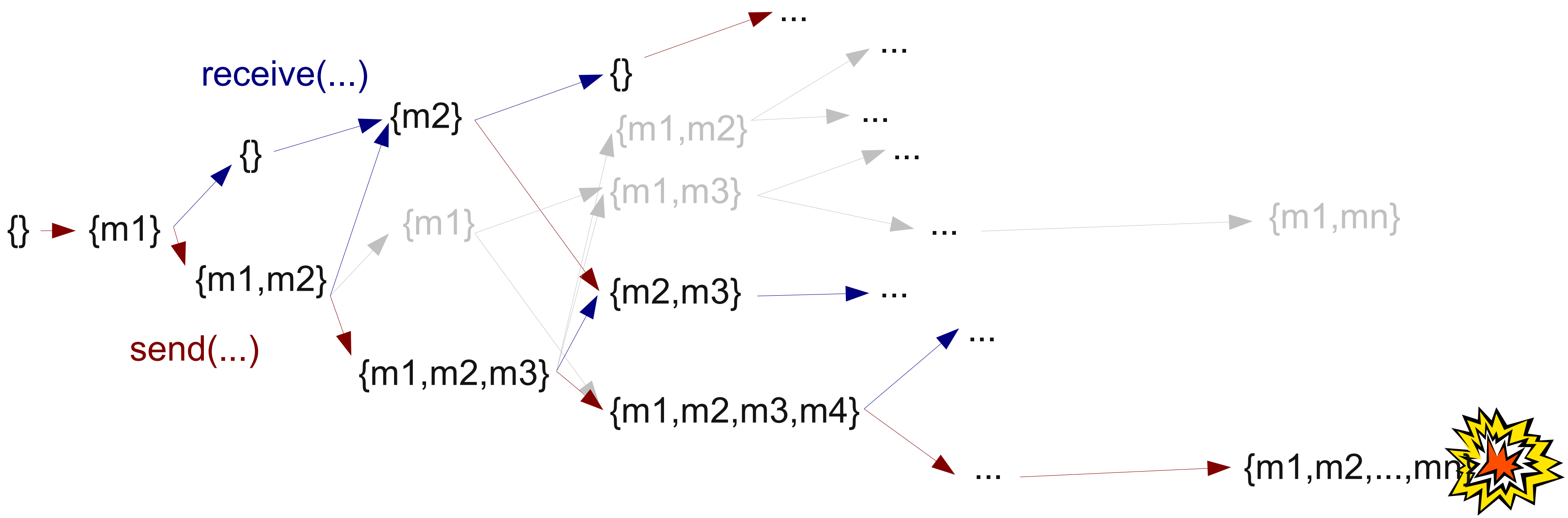



- Some behaviors do not satisfy liveness:
  - If m is sent, eventually m is received
- Some transitions don't get a fair chance to run:
  - receive(m1) and receive(m*)

# Fairness

- Partition transitions in tasks:
  - Tasks:
    - For all m: {receive(m)}
- Assume that no task can be forever prevented to take a step
- What about a FIFO reliable channel?

# Liveness and fairness



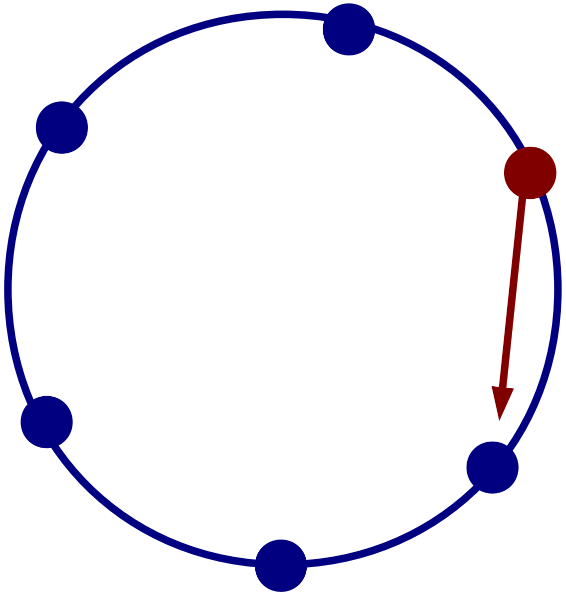

- FIFO order excludes a number of behaviors
  - Only executions with a finite number of receive(m) steps are unfair
- Fairness ensured by a single task:
  - {For all m: receive(m)}

# Example: FIFO channel

- State:
  - transit, seq. of M, initially <>
- Send(m), $m \in M$:
  - Pre-condition:
    - True
  - Effect:
    - transit :=transit+<m>
- Receive(m), $m \in M$:
  - Pre-condition:
    - m=head(transit)
  - Effect:
    - transit := tail(transit)
- Tasks:
  - {For all m: receive(m)}

# Example: Token ring

- Rotating token algorithm:



- Mutual exclusion?
- Deadlock freedom?

# Example: Token ring

- State:
  - n is the number of nodes
  - $token[0]=1$
  - $token[i]=0$, for $0<i<n$
- Move(i):
  - Pre-condition:
    - $token[i]=1$
  - Effect:
    - $token[i]:=0$
    - $token[(i+1) \bmod n]:=1$

# Example: Token ring

- Mutual exclusion:
  - There is at most one token in the ring (i.e. sum of token[i]≤1)
- Proof by induction:
  - Base step:
    - $\sum$token[i]=1 trivially true
  - Induction step:
    - $\sum$token-before[i]≤1⇒$\sum$token-after[i]≤1

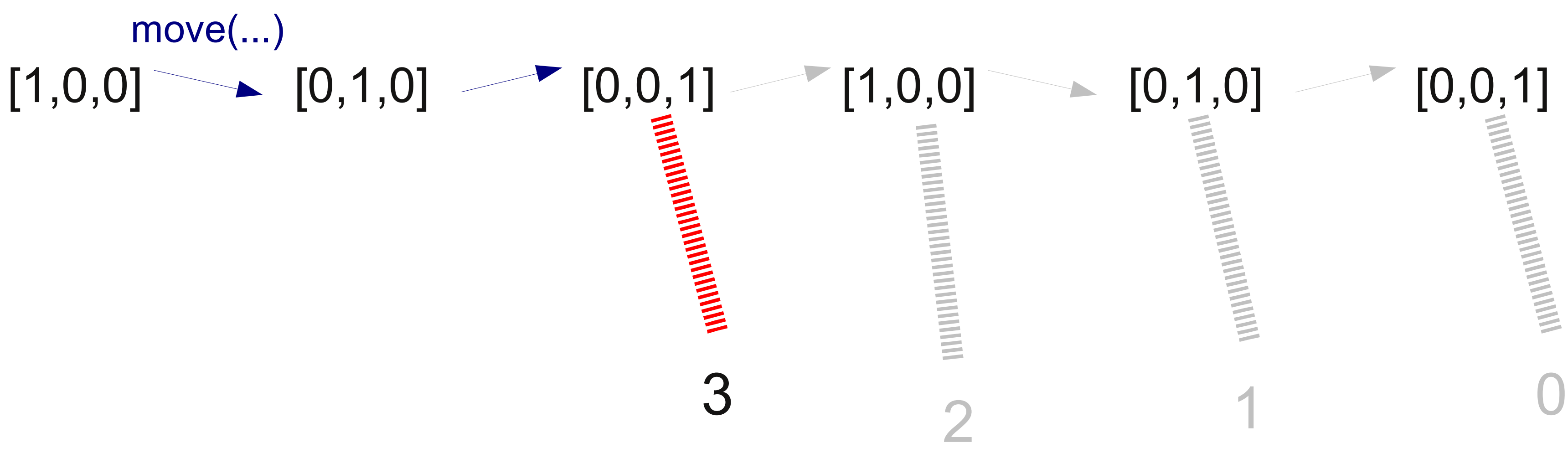
# Example: Token ring

- No starvation:
  - Eventually i gets the token at least k times
- Proof with a progress function:
  - Function from state to a well-founded set
  - Helper actions decrease the value
  - Other actions do not increase the value
  - Helper actions are taken until goal is met (i.e. enabled and in separate tasks)

Invariant assertion

# Progress function

- Define progress function f as:
  - Target is non-negative integers
  - Value is ((k-1) x n + i - 1) - length(trace)
- Example with n=3, k=2, and i=3:

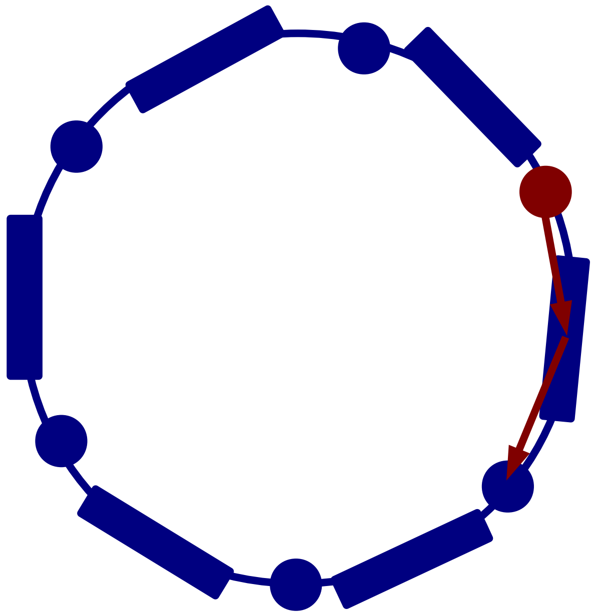


# Summary

- I/O Automata definition
  - Safety specification
  - Fairness specification
- Proof strategies for:
  - Invariants
  - Trace properties
    - Safety
    - Liveness
- How to apply to large and complex specifications?

# Example: Token ring with channels

- Refine the specification to include channels:



- Mutual exclusion?
- Deadlock freedom?

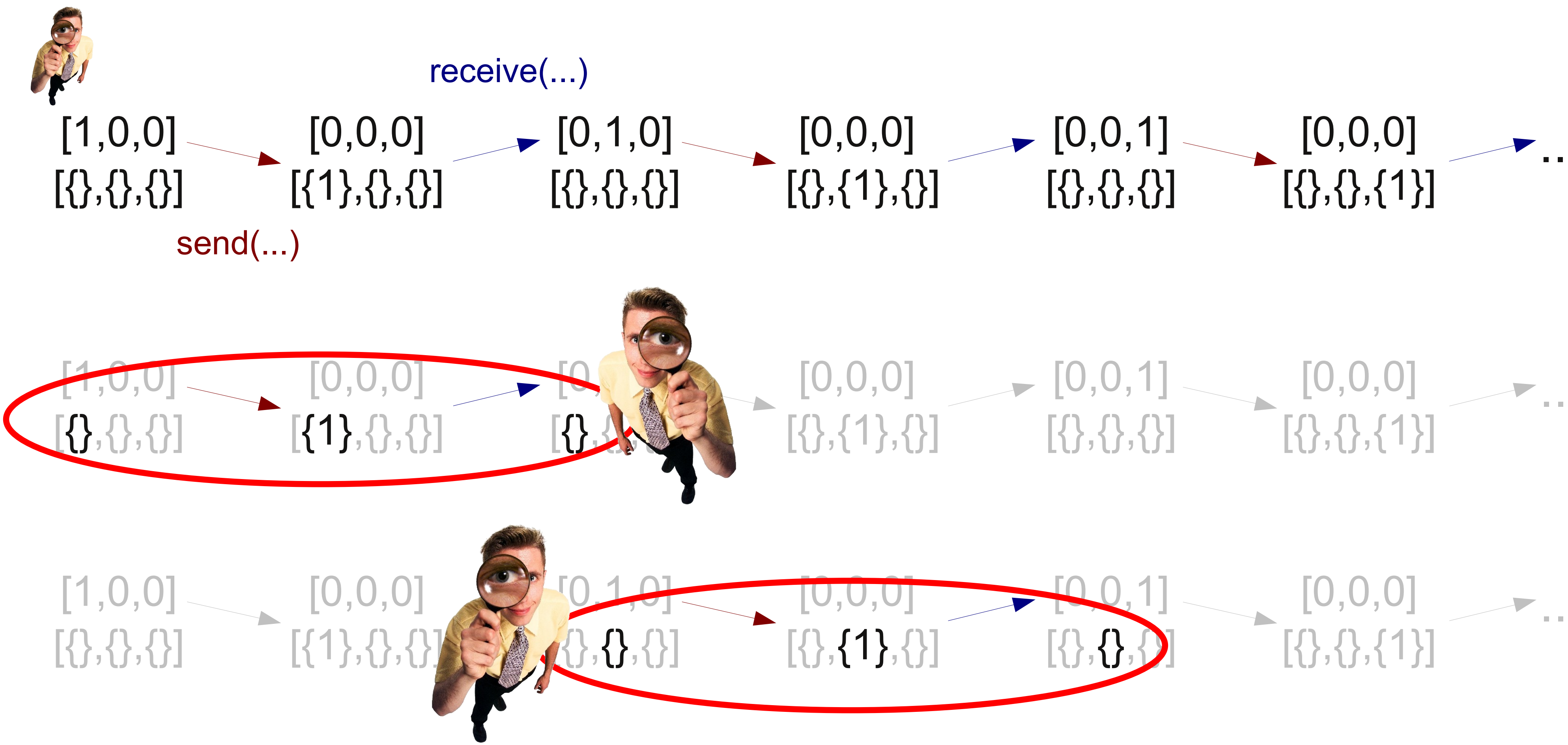
# Example: Token ring with channels

- Initially:
  - n is the number of nodes
  - token[0]=1
  - token[i]=0, for 0<i<n
  - transit[i]={}, for all i
- Send:
  - Pre-condition:
    - token[i]=1
  - Effect:
    - token[i]:=0
    - transit[i]:={1}
- Receive:
  - Pre-condition:
    - 1 in transit[i]
  - Effect:
    - token[(i+1)mod n]:=1
    - transit[i]:={}

# Example: Token ring with channels

- Proof of mutual exclusion?
- Seems to be true. But...
  - $\sum$token[i]≤1, with token=[1,0,0,...] and transit[0]={1}
  - after receive, $\sum$token[i]=2!
- Solution is to strengthen the invariant:
  - Prove by induction: $\sum$token[i]+$\sum$elems(transit[i])≤1
  - Then conclude $\sum$token[i]≤1
    (assuming that transit[i] not negative, easy to prove)

# Example: Token ring with channels




- One can observe valid executions of reliable channels embedded in the ring

# Composition

- Compatible automata:
  - Internal actions do not overlap with any other actions
  - Output actions are disjoint
  - No action is contained in infinitely many automata
- This allows:
  - Several input actions to overlap
  - Input actions to overlap with a single output action

# Composition

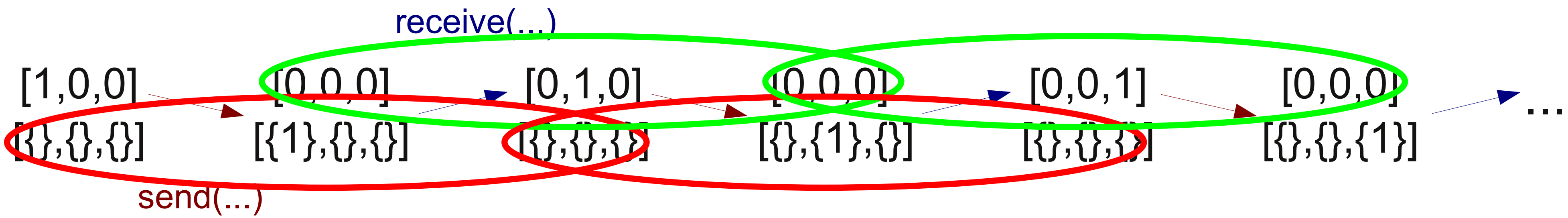
- A composition A with signature S from a set of Ai, with signature Si
- The state of the composed automaton A is:
  - state(A) = Π state(Ai)
  - start(A) = Π start(Ai)
- The signature of S is as follows:
  - out(S) = U out(Si)
  - int(S) = U int(Si)
  - in(S) = U in(Si) – out(S)
- Transitions and tasks likewise

# Example: A process

- State:
  - token, integer, initially 0
- Send(m), $m \in M$:
  - Pre-condition:
    - token = 1
  - Effect:
    - token := 0
- Receive(m), $m \in M$:
  - Pre-condition:
    - true
  - Effect:
    - token := 1

# Example: Composite token ring

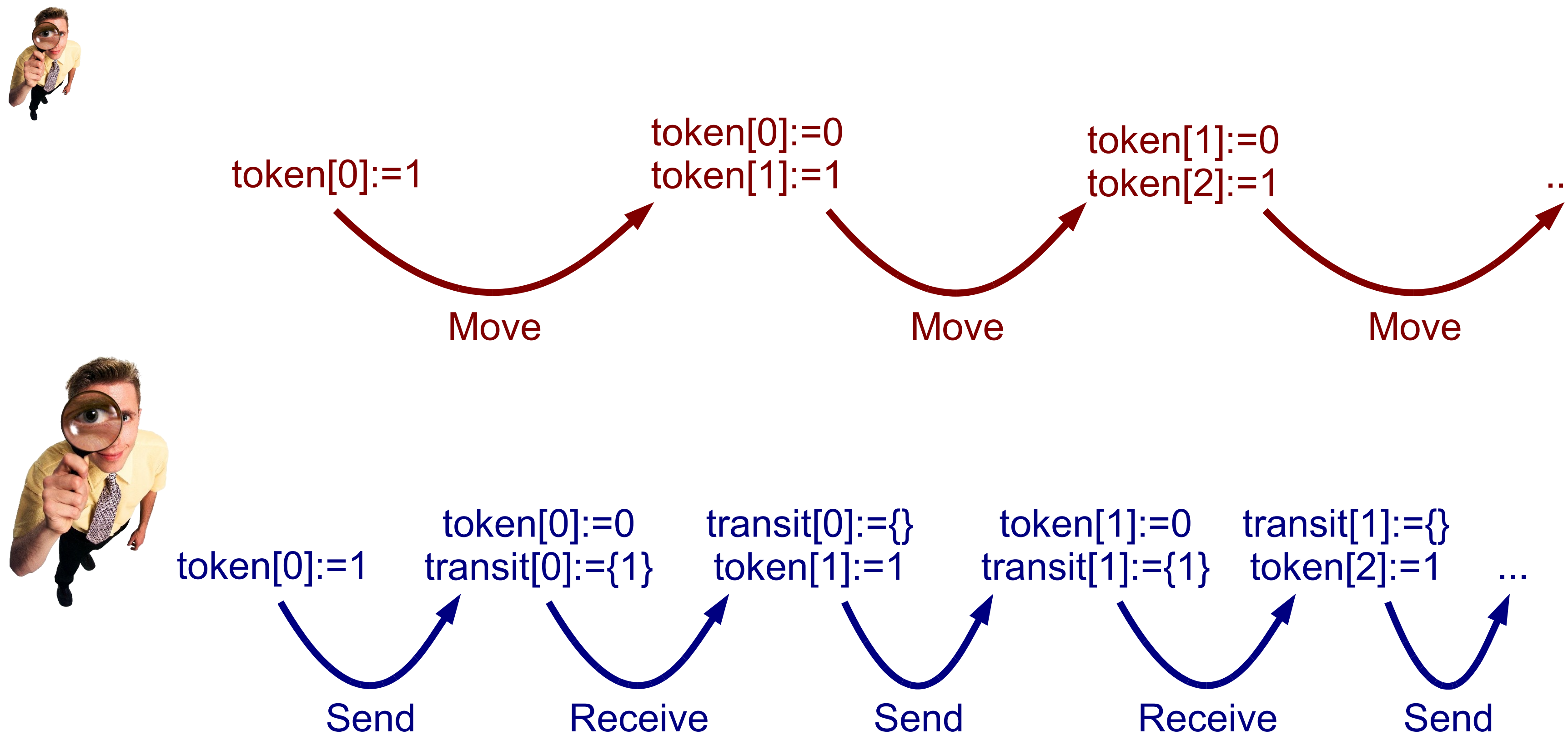
- send(m) is an input to a channel
  - overlaps with send(m) in a process
- receive(m) is an input to a process
  - overlaps with receive(m) in a channel



# Compositional reasoning

- A necessary condition for mutual exclusion in a ring is that the token is not duplicated while in transit
- Consider the following trace property:
  - For each receive(m) (i.e. lock), there is some corresponding send(m) (i.e. unlock)
- This property is true for each individual reliable channel
- Therefore it is true for the composed token ring

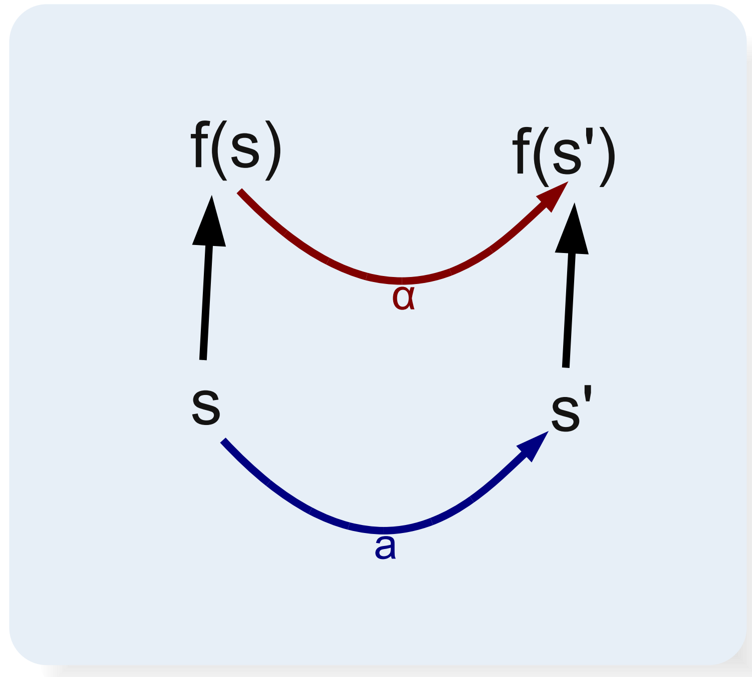
# Which level of abstraction?



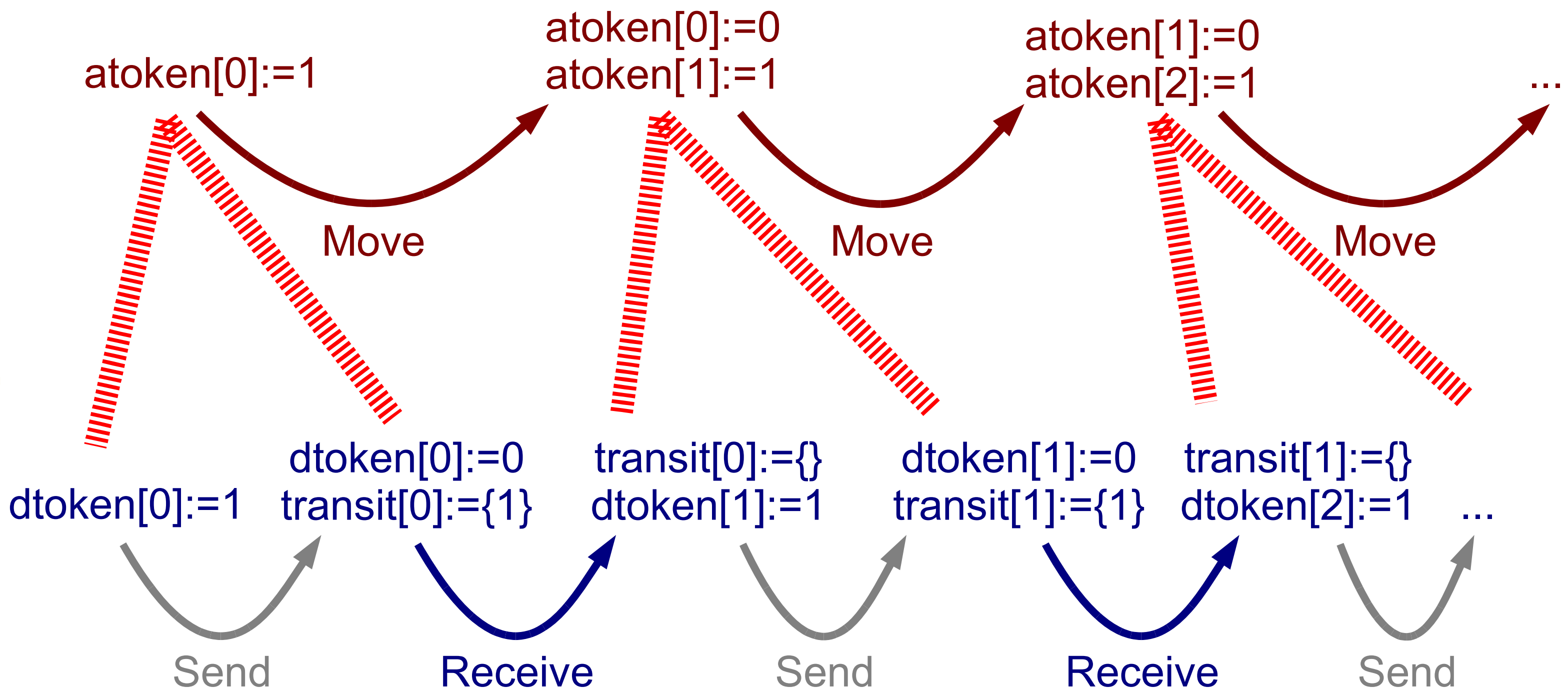

- Observations of the same system at different levels of abstraction
  - How to relate them?
  - Variable token is not observing the same thing!

# Simulation

- Map actions
- Map states:
  - f(detailed state) = abstract state
- Inicial states map
- Every detailed sequence a maps to an abstract sequence α



# Simulation

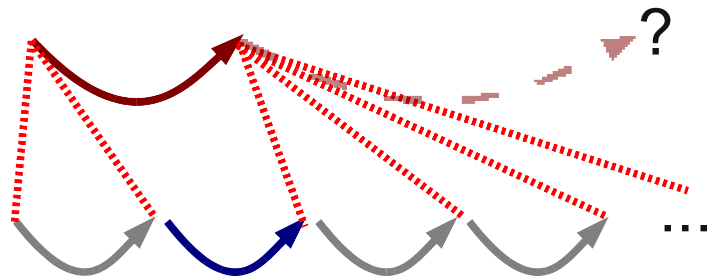



- Map <Receive> to <Move>, <Send> to <>.
- f: atoken[i] = dtoken[i] + transit[i]

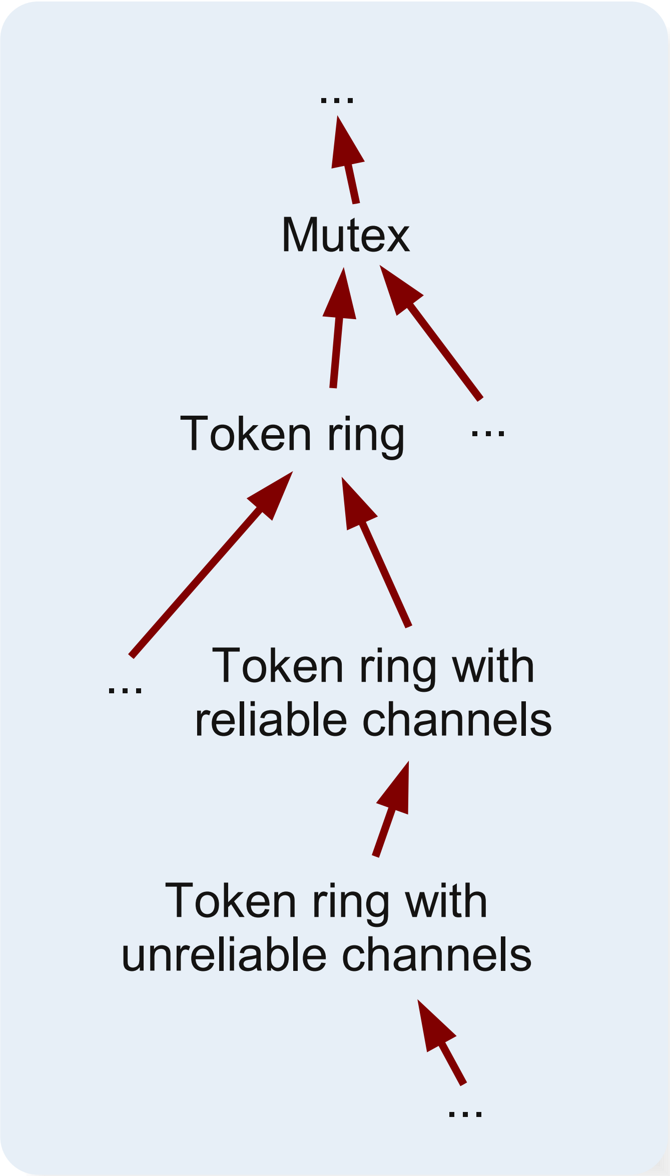
# Simulation

- If all detailed behaviors can be mapped to abstract behaviors, then:
  - A simulation proof exists
  - But may require an intermediate specification
- Simulation preserves safety properties
- Simulation does not necessarily preserve liveness properties:



# Refinement

- The goal is refinement of specifications
- Going up:
  - Understand similarities between different problems
- Going down:
  - Closer to the implementation (i.e. code)



# Summary

- Additional proof strategies:
  - Compositional reasoning
  - Simulation
- More:
  - N. Lynch. *Distributed Algorithms* (Ch. 8)