Distributed Computing

José Orlando Pereira

Grupo de Sistemas Distribuídos
Departamento de Informática
Universidade do Minho

2010/2011
Specifications and automata

- Specification is a set of allowable behaviors:

\[ S = \{ \text{vcnt}=3 \text{ timeout}, \text{vcnt}=2 \text{ timeout}, \text{vcnt}=1 \text{ timeout}, \text{vcnt}=0 \text{ ring} \} \]

- An automaton provides a compact and practical representation
An I/O automaton $A$ has five components:

- $\text{sig}(A)$, a triplet $S$ of disjoint sets of actions:
  - $\text{in}(S)$, the input actions
  - $\text{out}(S)$, the output actions
  - $\text{int}(S)$, the internal actions
- $\text{states}(A)$, a (possibly infinite) set of states
- $\text{start}(A)$, a non-empty subset of $\text{states}(A)$
- $\text{trans}(A)$, a subset of $\text{states}(A) \times \text{acts}(\text{sig}(A)) \times \text{states}(A)$
- $\text{tasks}(A)$, a partition of $\text{local}(\text{sig}(A))$
Transitions

- A transition is enabled in state $s$ if there is some $\pi, s'$ such that $(s, \pi, s') \in \text{trans}(A)$
- 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
Timeout:
- Pre-condition:
  - \( \neg \text{END} \) and \( vcnt > 0 \)
- Effect:
  - \( vcnt := vcnt - 1 \)

Ring:
- Pre-condition:
  - \( \neg \text{END} \) and \( vcnt = 0 \)
- Effect:
  - \( \text{END} := \text{True} \)

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`
  - "..."
Safe behaviors

- Enumerating safe behaviors:
  - Start with a behavior for each state $s$ in $\text{start}(A)$
  - For each transition $(s,a,s')$ in $\text{trans}(A)$ enabled for some state $s$ at the end of any known safe behavior:
    - Create a behavior with $(a,s')$ appended
  - Repeat (possibly, for ever...)
Safety properties

• Proof of safety properties:
  • Invariant proof by induction

• Strategies:
  • Strengthen the invariant
  • Include trace in state
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:**
    - \( vcnt\text{-after} = vcnt\text{-before} - 1 \)
    - \( vcnt\text{-before} < 4 \)
      - \( vcnt\text{-after}+1 < 4 \)
      - \( vcnt\text{-after} < 3 < 4 \): Done

  - **Ring transition:**
    - always \( vcnt\text{-after} = vcnt\text{-before} = 0 \)
    - \( 0 < 4 \): Done
Example: Reliable channel

- Reliable channel:
  - Unordered
  - FIFO

Why $\text{Receive}(m)$ and not $m := \text{Receive}()$?
Example: Reliable channel

- **State:**
  - transit, bag of M, initially \{\}

- **Send(m), m ∈ M:**
  - Pre-condition:
    - True
  - Effect:
    - transit := transit + \{m\}

- **Receive(m), m ∈ M:**
  - Pre-condition:
    - m in transit
  - Effect:
    - transit := transit - \{m\}
Concurrency is modeled by alternative enabled transitions:

- Sender and receiver
- Within the channel (reordering)
Some behaviors do not satisfy liveness:
- If \( m \) is sent, eventually \( m \) is received
- Some transitions don't get a fair chance to run:
  - \( \text{receive}(m_1) \) and \( \text{receive}(m^*) \)
Partition transitions in tasks:

Tasks:
- For all \( m \): \( \{ \text{receive}(m) \} \)

Assume that no task can be forever prevented from taking 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 ∈ M:**
  - Pre-condition: True
  - Effect: transit := transit + <m>

- **Receive(m), m ∈ 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) mod n]:=1
Example: Token ring

- Mutual exclusion:
  - There is at most one token in the ring (i.e. \( \sum \text{token}[i] \leq 1 \))

- Proof by induction:
  - Base step:
    - \( \sum \text{token}[i] = 1 \) trivially true
  - Induction step:
    - \( \sum \text{token-before}[i] \leq 1 \Rightarrow \sum \text{token-after}[i] \leq 1 \)
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
Define progress function $f$ as:
- Target is non-negative integers
- Value is $((k-1) \times n + i - 1) - \text{length(trace)}$

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
- \( \text{token}[0] = 1 \)
- \( \text{token}[i] = 0 \), for \( 0 < i < n \)
- \( \text{transit}[i] = \{\} \), for all \( i \)

Send:
- Pre-condition:
  - \( \text{token}[i] = 1 \)

Effect:
- \( \text{token}[i] := 0 \)
- \( \text{transit}[i] := \{1\} \)

Receive:
- Pre-condition:
  - 1 in \( \text{transit}[i] \)
- Effect:
  - \( \text{token}[(i+1) \mod n] := 1 \)
  - \( \text{transit}[i] := \{\} \)
**Example: Token ring with channels**

- **Proof of mutual exclusion?**
- **Seems to be true. But...**
  - \[ \sum_{i} \text{token}[i] \leq 1, \text{ with token} = [1, 0, 0, ...] \text{ and transit}[0] = \{1\} \]
  - after receive, \[ \sum_{i} \text{token}[i] = 2! \]
- **Solution is to strengthen the invariant:**
  - Prove by induction: \[ \sum_{i} \text{token}[i] + \sum \text{elems}(\text{transit}[i]) \leq 1 \]
  - Then conclude \[ \sum_{i} \text{token}[i] \leq 1 \]
    (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 $A_i$, with signature $S_i$

The state of the composed automaton $A$ is:
- $\text{state}(A) = \prod \text{state}(A_i)$
- $\text{start}(A) = \prod \text{start}(A_i)$

The signature of $S$ is as follows:
- $\text{out}(S) = \bigcup \text{out}(S_i)$
- $\text{int}(S) = \bigcup \text{int}(S_i)$
- $\text{in}(S) = \bigcup \text{in}(S_i) - \text{out}(S)$

Transitions and tasks likewise
Example: A process

- **State:**
  - token, integer, initially 0

- **Send(m), m∈ M:**
  - Pre-condition:
    - token = 1
  - Effect:
    - token := 0

- **Receive(m), m∈ 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:
  - \( f(\text{detailed state}) = \text{abstract state} \)
- Initial states map
- Every detailed sequence \( a \) maps to an abstract sequence \( \alpha \)
Map <Receive> to <Move>, <Send> to <>.

\[ f: \text{atoken}[i] = \text{dtoken}[i] + \text{transit}[i] \]
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:
The goal is refinement of specifications

Going up:
- Understand similarities between different problems

Going down:
- Closer to the implementation (i.e. code)
Conclusion

First goal achieved:
- I/O Automata
- Safety and liveness proofs
- Composition
- Refinement

Goals

- How do we make sure that algorithms are correct?
- Why are algorithms correct?