Asynchronous Multi-Agent ASMs

An Introduction

Egon Börger

Dipartimento di Informatica, Universita di Pisa
http://www.di.unipi.it/~boerger

For details see Chapter 2 (Asynchronous Multi-Agent ASMs) of:

E. Börger, R. Stärk

Abstract State Machines

A Method for High-Level System Design and Analysis

Springer-Verlag 2003

For update info see AsmBook web page:

http://www.di.unipi.it/AsmBook

- Definition of Async ASMs
- Small Examples
  - Mutual Exclusion
    - Dining Philosophers
    - Multiple Read One Write
  - Network Algorithms
    - Master Slave
    - Consensus
    - Load Balance
    - Leader Election
    - Echo
    - Phase Synchronization
- Simple Properties of Async ASMs
- References

Recalling Basic ASMs: computation step

- Basic ASM: a finite set of rules, executed by a single agent self, of form
  - If Condition Then Updates
- A computation step of a basic ASM:
- In the current state (well-defined global structure) S:
  - determine all the fireable rules in S (s.t. Cond is true in S)
  - compute all expressions t_i,t occurring in updates $f(t_1,\ldots,t_n):=t$
    - this includes reading of the monitored functions, supposed to have well-defined stable values for each execution step
    - execute simultaneously all these function updates

Recalling Basic ASMs: next state

- The updating yields a next state S', which is well-defined by the updates - and by possible changes of the monitored function values, thought of as resulting from implicit actions of the environment
  - environment actions are assumed to be "located" with respect to a machine step in such a way that their result is ready when the machine is going to perform its next step (implicit environment-controller separation principle)

Distributed Runs: State Stability Problem

- In runs of asynchronously cooperating independent agents, the possible incomparability of moves - coming with different data, clocks, moments and duration of execution - makes it difficult
  - to uniquely define a global state where moves are executed
  - to locate changes of monitored fcts in the ordering of moves
- The **coherence condition** in the definition of asynchronous multi-agent ASM runs below postulates well-definedness for a relevant portion of state in which an agent is supposed to perform a step, thus providing a notion of "local" stable view of "the" idealized state in which an agent makes a move.

Async ASM Runs: Environmental Changes

- Changes of monitored functions can be made explicit as resulting from moves of one or more environment agents.
  - Such **monitored moves** - moves of “unkown” agents, as opposed to the **controlled moves** of the explicitly given agents - are often thought of as implicitly given, e.g. by speaking of “time moments” in which changes of monitored functions take place.

Defining Asynchronous Multi-Agent ASMs

- An **Async ASM** is a family of pairs (a, ASM(a)) of
  - agents a $\in$ Agent (a possibly dynamic set)
  - basic ASMs ASM (a)
- A **Run** of an async ASM is a partially ordered set (M, $<$) of "moves" m of its agents s.t.:
  - finite history: each move has only finitely many predecessors, i.e. {m' | m' $<$ m } is finite for each m$\in$ M
  - sequentiality of agents: for each agent a $\in$ Agent, his moves {m | m$\in$ M, a performs m} are linearly ordered by $<$
  - coherence: each (finite) initial segment X of (M, $<$) has an associated state σ (X) – think of it as the result of all moves in X with m executed before m' if m<m'– which for every maximal element m$\in$ X is the result of applying move m in state σ(X-{m})

Linearizing Initial segments of runs of an async ASM

- **Linearization Lemma.** Let X be a finite initial segment (downward closed subset) of a run of an async ASM. Each linearization of X yields a run with the same final state.
- Proof: follows from the coherence condition

- Definition & Simple Properties of Async ASMs
- Small Examples
  - Mutual Exclusion
    - Dining Philosophers
    - Multiple Read One Write
  - Network Algorithms
    - Master Slave
    - Consensus
    - Load Balance
    - Leader Election
    - Echo
    - Phase Synchronization
- Exercises and References

See Reisig 1998, Sect. 10

Exclusive use of shared resources: problem statement

- Goal: Design a distributed algorithm that
  - allows every interested agent to eventually use some shared resource in exclusive use
  - prevents two users to use the shared resource simultaneously
- Agent: set of agents each equipped with
  - resource: Resource (different agents may have the same resource)
- owner: Resource $\rightarrow$ Agent recording the current user of a given resource

Exclusive use of shared resources: single agent view

- The algorithm has to manipulate updates of owner in such a way that every attempt by an agent to become owner of his resource will eventually succeed
- Every single agent would like to execute alternately the following two rules:

```
if owner(resource) = none
    then owner(resource):= self
```

```
if owner(resource) = self
    then owner(resource):= none
```

Conflicts in using resource shared among different agents are risolved by defining a partial order among possibly conflicting moves in an (appropriately initialized) run of an async ASM. Realizing such an order reflects appropriate scheduling and priority policies.

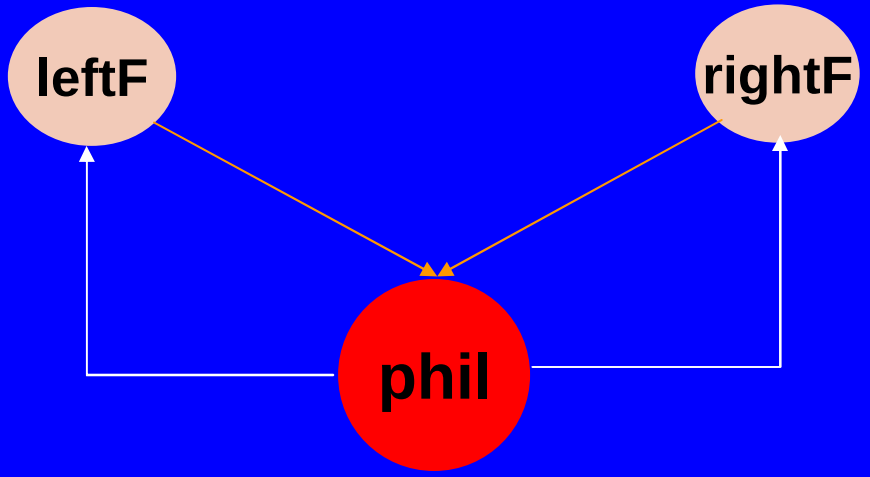
Dining Philosophers
illustrating exclusive use of shared resources

Shared resources represented as "forks" used by "philosophers"

Resource assigning function F: depends on the neighbourhood in the underlying graph

e.g. each agent may have one left and one right neighbour sharing the resource:

Fork = (leftF,rightF) with elementwise defined owner




```
if owner(leftF) = owner(rightF) = none
    then owner(leftF):= self
         owner(rightF):= self
```

```
if owner(leftF) = owner(rightF) = self
    then owner(leftF):= none
         owner(rightF):= none
```

- Definition of Async ASMs
- Small Examples
  - Mutual Exclusion
    - Dining Philosophers
    - Multiple Read One Write See Reisig 1998, Sect. 24
  - Network Algorithms
    - Master Slave
    - Consensus
    - Load Balance
    - Leader Election
    - Echo
    - Phase Synchronization
- Simple Properties of Async ASMs
- References

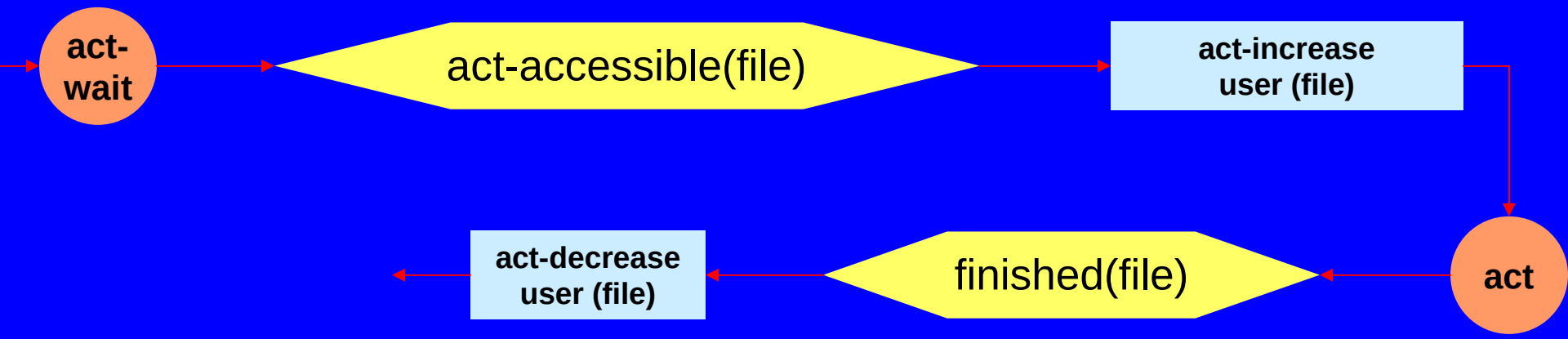
Multiple Reads One Write : problem statement (non-exclusive and exclusive resource sharing)

- Goal: Design a distributed algorithm allowing at each step one agent to start a read or a write operation in a given file, up to max-read>0 simultaneous reads but only 1 write (not overlapping with any read)
  - Below we define agent rules for read/write access to files which in a distributed run of an async ASM (of agents all of which are equipped with those rules) reach the requested goal
    - NB. In the example we do not define the partial order for such distributed runs.

Multiple Reads One Write : Agent Signature

- Agent : set of agents which are allowed to access files for read/write operations
  - finished:File $\rightarrow$ Bool indicating whether an agent has finished his current file operation
- File: set of files equipped with
  - user : Nat indicating the number of agents which are currently reading or writing the file
  - max-read,max-write maximal number of agents allowed to simultaneously use the file for reading/writing

Multiple Reads One Write: agent rules for act=read/write



act-accessible(file) ᵒ
user (file) < max-act (file)

max-write(file) = 1
write-in/decrease user(file) ᵒ
user (file):= user(file) ± 4

max-read (file) = 4
write-in/decrease user(file) ᵒ
user (file):= user(file) ± 1

If instead of 1 one wants to allow multiple simultaneous file access attempts,
max-read becomes a **cumulative** counter to have the expected overall effect

read-accessible (file) becomes user (file) + newUsers ≤ max-act (file)
where newUsers indicates the number of users attempting to access the file for reading

- Definition of Async ASMs
- Small Examples
  - Mutual Exclusion
    - Dining Philosophers
    - Multiple Read One Write
  - Network Algorithms
    - Master Slave Agreement See Reisig 1998, Sect. 30 (Fig.30.1), 75
    - Consensus
    - Load Balance
    - Leader Election
    - Echo
    - Phase Synchronization
- Simple Properties of Async ASMs
- References

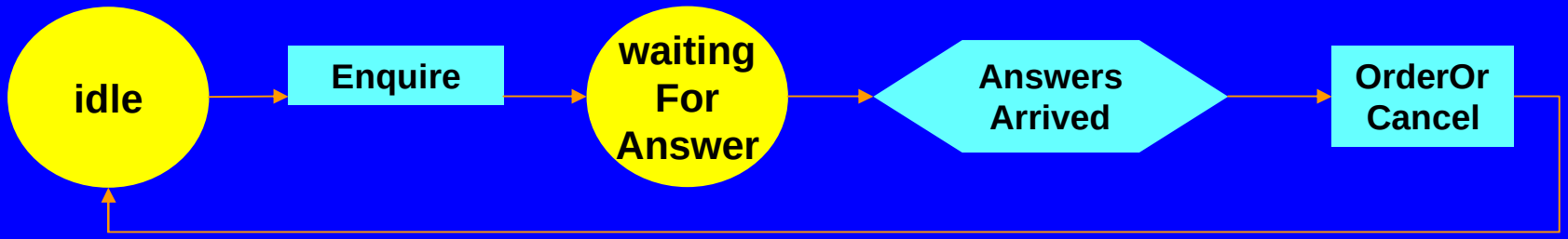
Master/Slave Agreement : problem statement

- Goal: Design a distributed algorithm for a master launching orders to slave agents, to be confirmed by them and to be executed iff all of them confirm to the master the acceptance of the order.
- Algorithmic Idea:
  - the master enquires about a job to be launched and then waits for answers from the slaves
  - the slaves answer the enquiry and then wait for the order or the cancellation of the launched job from the master
  - the master orders or cancels the job depending on whether all slaves answered to accept it or not
- Eventually the master becomes idle, with all slaves either idle too or executing the accepted job.

Master/Slave Agreement : Agent Signature

- master: a distinguished agent
  - order: Order external function yielding jobs to be sent to the slaves
  - ctl_state: {idle, waitingForAnswer}
- Slaves: a set of agents equipped with
  - asked: {true,false} recording whether a job request has been launched by the master to the slave
  - answer : {accept,refuse,undef} recording whether a job request has been acceped by the slave
  - ctl_state:{idle, waitingForOrder, busy}
- Initially ctl_state = idle, order = answer =undef, asked = false
- NB. Abstraction from message passing: functions asked, answer, order shared in writing resp.reading among slaves and master

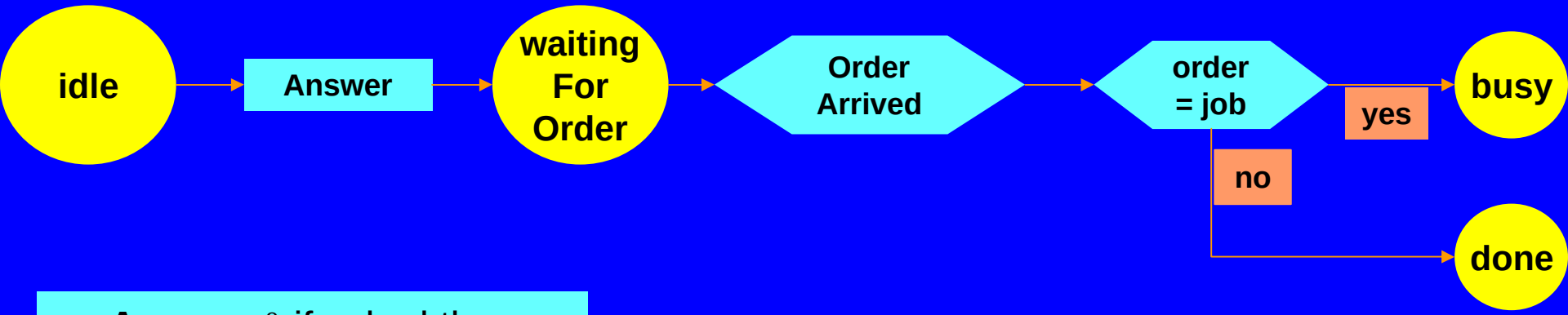
Master/Slave Agreement ASMs



Enquire ° **forall** s ∈ Slave s.asked := true

Answers Arrived ° **forall** s ∈ Slave
s.answer ∈ {accept, refuse}

OrderOrCancel °
clear answer
if for some s ∈ Slave s.answer = refuse
then order:=cancel
else order:=job



Answer ° if asked then
choose r ∈ {accept, refuse}
answer:=r
asked:=false

OrderArrived ° order ∈ { job, cancel }

Master/Slave Agreement Correctness

- Proposition: In every run of a set of master and slaves, all equipped with the corresponding master/slave ASM, after the master has started an Enquiry, eventually the master becomes idle and
  - either all slaves become done or
  - all slaves become busy executing the job ordered by the master
- Proof. Follows by run induction.

- Definition of Async ASMs
- Small Examples
  - Mutual Exclusion
    - Dining Philosophers
    - Multiple Read One Write
  - Network Algorithms
    - Master Slave
    - Consensus See Reisig 1998, Sect. 35 (Fig. 35.1), 80
    - Load Balance
    - Leader Election
    - Echo
    - Phase Synchronization
- Simple Properties of Async ASMs
- References

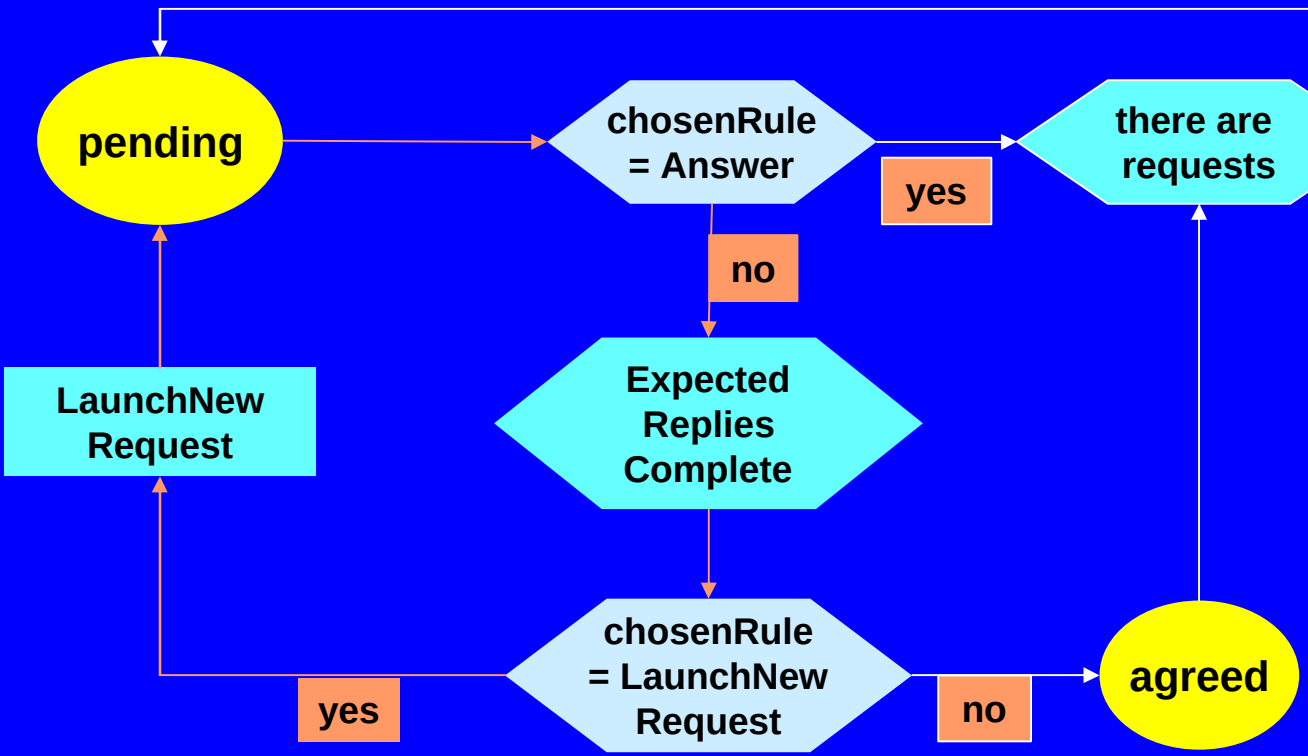
Consensus in Networks: problem statement

- Goal: Design a distributed algorithm for reaching consensus among homogeneous agents of finite connected networks (using only communication between neighbors, without broker or mediator, with abstract requests and answers).
- Algorithmic Idea: every agent (node) may
  - launch a request to his neighbors and wait for the replies
  - agree with the replies received from his neighbors
  - reply to requests received from his neighbors
- until all agents agree (maybe never)

Consensus ASM: Agent Signature

- Agent : finite connected set (of nodes)
- Each agent equipped with:
  - neighb $\subseteq$ Agent (external function)
  - Request $\subseteq$ {RequestFrom(n)|n$\in$ neighb } (controlled function)
  - Reply $\subseteq$ {ReplyFrom(n)|n $\in$ neighb } (controlled function)
  - ctl_state : {pending, agreed}
- Initially ctl_state=pending, Request = empty, Reply = full
- chosenRule to control non-deterministic choice between LaunchNewRequest and Answer

Consensus Control State ASM



ExpectedRepliesComplete
- Reply = full

Answer -

forall r ∈ Request **do**
delete r from Request
send answer for r

LaunchNewRequest -
Reply := empty
BroadcastRequest

send answer for RequestFrom(n) -
insert ReplyFrom(self) into Reply(n)

Variation: **choose** r ∈ Request **...**

BroadcastRequest - **forall** n ∈ neighb
insert RequestFrom(self) into Request(n)

Async Consensus ASM : Stability property

- Proposition: In every distributed run of agents equipped with the consensus ASM, if the run terminates, then for every agent holds:
  - ctl_state = agreed
  - Reply = full
  - Request = empty
- Proof (assuming that every enabled agent will eventually make a move): follows from the definition of LaunchNewRequest, Answer.
  - When Reply=full at agent, then there is no RequestFrom(agent) in Request(n) for any n ∈ neighb(agent)

- Definition of Async ASMs
- Small Examples
  - Mutual Exclusion
    - Dining Philosophers
    - Multiple Read One Write
  - Network Algorithms
    - Master Slave
    - Consensus
    - Load Balance See Reisig 1998, Sect. 37 (Fig.37.1), 82
    - Leader Election
    - Echo
    - Phase Synchronization
- Simple Properties of Async ASMs
- References

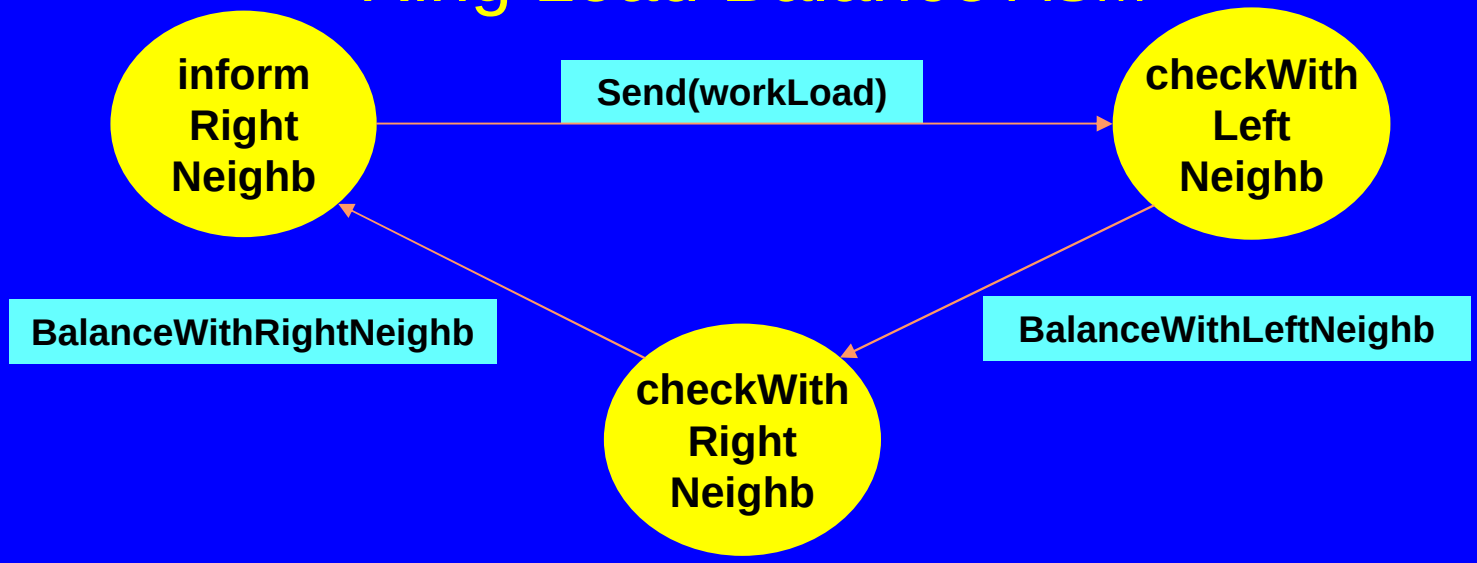
Ring Load Balance: problem statement

- Goal: Design a distributed algorithm for reaching workload balance among the agents of a ring (using only communication between right/left neighbors)
  - keeping message passing mechanism and effective task transfer abstract, assuming fixed number of agents and total workload
- Algorithmic Idea: every agent (ring node)
  - alternately sends
    - a workload info mssg to his right neighbor
    - a task transfer mssg to his left neighbor, transfering a task to balance the workload with the left neighbor
  - updates his workload to balance it with his right neighbor
- so that eventually the difference between the workload of two nodes becomes ≤ 1
  - NB. Sending workload info mssg precedes task balancing

Ring Load Balance ASM: Agent Signature

- Agent : a ring of agents equipped with:
  - leftNeighb, rightNeighb: Agent (external functions)
  - workLoad: the current workload
  - neighbLoad: current workload of leftNeighb
  - transferLoad: workload transfered by rightNeighb
  - ctl_state : {informRightNeighb, checkWithLeftNeighb , checkWithRightNeighb }
- Initially
  - ctl_state = informRightNeighb
  - neighbLoad = transferLoad = undef

Ring Load Balance ASM



BalanceWithRightNeighb ° If arrived(transferLoad) then accept task from rightNeighb

BalanceWithLeftNeighb ° If arrived(neighbLoad) then transfer task to leftNeighb

accept task from rightNeighb °
workLoad:= workLoad + transferLoad
transferLoad:= undef

arrived(l) ° l ≠undef

transfer task to leftNeighb °
let transfer = workLoad > neighbLoad **in**
leftNeighb.transferLoad:= transfer
workLoad:= workLoad – transfer
neighbLoad:= undef

Send(workLoad) °
rightNeighb.neighbLoad
:= workLoad

Async Ring Load Balance ASM : Correctness

- Proposition: In every distributed run of agents equipped with the ring load balance ASM, eventually the workload difference between two neighboring nodes becomes ≤ 1.
- Proof (assuming that every enabled agent will eventually make a move): induction on the wieght of run workload differences. Let w = total workLoad of all nodes, a= |Agent|.
  - Case 1: t|a. Then eventually workLoad(n)=w/a for every node n.
  - Case 2: otherwise. Then eventually the workLoad of some neighboring nodes will differ by 1.

- Definition of Async ASMs
- Small Examples
  - Mutual Exclusion
    - Dining Philosophers
    - Multiple Read One Write
  - Network Algorithms
    - Master Slave
    - Consensus
    - Load Balance
    - Leader Election See Reisig 1998, Sect. 32 (Fig.32.1/2), 76
    - Echo
    - Phase Synchronization
- Simple Properties of Async ASMs
- References

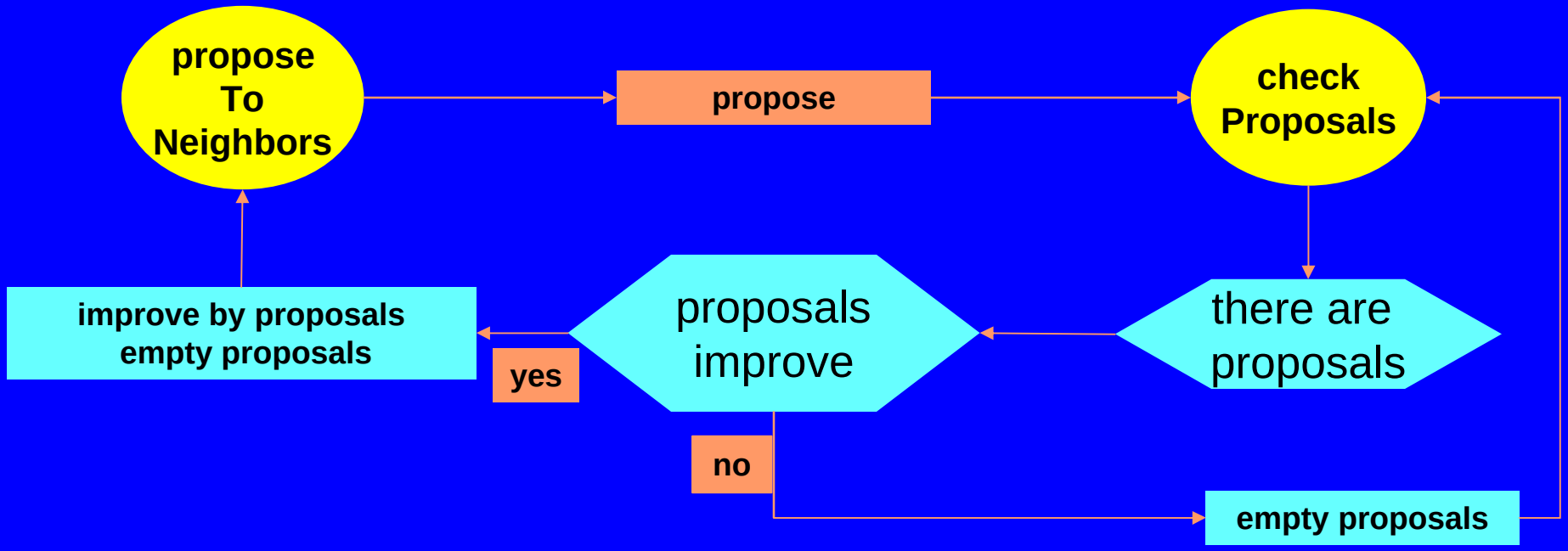
Leader Election: problem statement

- Goal: Design a distributed algorithm for the election of a leader in finite connected networks of homogeneous agents, using only communication (message passing) between neighbor nodes.
- Assumptions:
  - network nodes (agents) are connected & linearly ordered
  - leader = max (Agent) wrt the linear order <
- Algorithmic Idea: every agent (network node)
  - proposes to his neighbors his current leader candidate
  - checks the leader proposals received from his neighbors
    - upon detecting a proposal which improves his leader candidate he improves his candidate for his next proposal
- Eventually cand=max(Agent) holds for all agents

Leader Election: Agent Signature

- Agent : finite linearly ordered connected set
  - < linear order of Agent (external function)
    - leader = max (Agent) wrt the linear order <
- Each agent equipped with:
  - neighb ⊆ Agent (external function)
  - cand: Agent (controlled function)
  - proposals ⊆ Agent (controlled function)
  - ctl_state : {proposeToNeighbors, checkProposals}
- Initially ctl_state=proposeToNeighbors, cand=self, proposals = empty

Leader Election Control State ASM



propose ° forall n ∈ neighb insert cand to proposals(n)

proposals improve ° max (proposals) > cand

improve by proposals ° cand := max (proposals)

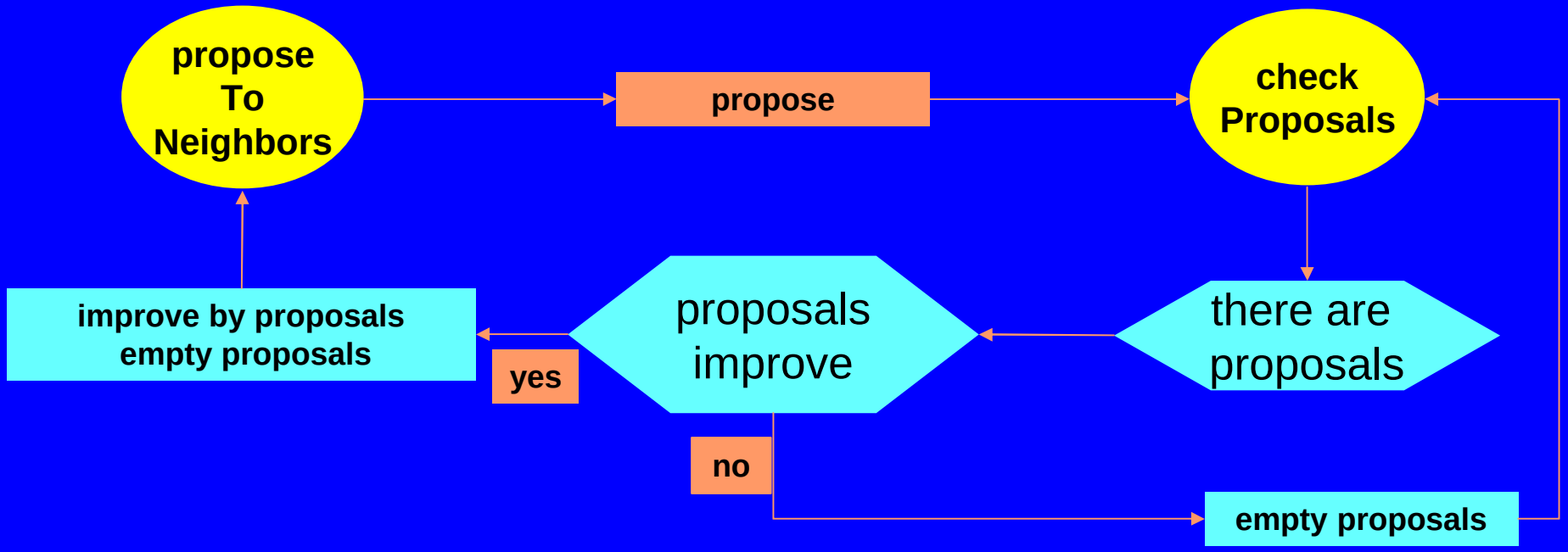
Leader Election: Correctness property

- Proposition: In every distributed run of agents equipped with the leader election ASM, eventually for every agent holds:
  - cand=max(Agent)
  - ctl_state = checkProposals
  - proposals = empty
- Proof (assuming that every enabled agent will eventually make a move): induction on runs and on $\Sigma\{\text{leader} - \text{cand}(n) \mid n \in \text{Agent}\}$
  - measuring “distances” of candidates from leader

Refining Leader Election: compute minimal path to leader

- Goal: refine the leader election algorithm to compute for each agent also a shortest path to the leader, providing
  - a neighbor (except for leader) which is closest to the leader
  - the minimal distance to the leader
- Idea: enrich cand and proposals by neighbor with minimal distance to leader:
  - nearNeighb: Agent
  - distance: Distance (e.g. = Nat $\cup\{\infty\}$)
  - proposals: PowerSet(Agent x Agent x Distance)
- Initially nearNeighbor = self distance =0

ASM Computing Minimal Path To Leader



propose ° forall n ∈ neighb insert (cand, nearNeighb, distance) to proposals(n)

proposals improve ° **let** m = Max (proposals) **in** m > cand
or (m=cand **and** minDistance(proposalsFor m)+1 < distance)

Max taken over agents

update PathInfo to m °
choose (n,d) with (m,n,d) ∈ proposals
d=minDistance(proposalsFor m) **in**
nearNeighb := n
distance := d+1

improve by proposals °
cand := Max (proposals)
update PathInfo to Max (proposals)

Minimal Path Computation: Correctness

- Proposition: In every distributed run of agents equipped with the ASM computing a minimal path to the leader, eventually for every agent holds:
  - cand=max(Agent)=leader
  - distance=minimal distance of a path from agent to leader
  - nearNeighbor = a neighbor of agent on a minimal path to the leader (except for leader where nearNeighbor=leader)
  - ctl_state = checkProposals
  - proposals = empty
- Proof (assuming that every enabled agent will eventually make a move): induction on runs and on $\Sigma\{\text{leader} - \text{cand}(n) \mid n \in \text{Agent}\}$ with side induction on the minimal distances in proposalsFor Max(proposals)

Exercises

- Refine the CHECK submachine of the leader election ASM by a machine which checks proposals elementwise. Prove that the refinement is correct.
  - Hint: See Reisig op.cit. Fig.32.1
- Adapt the ASM for the election of a maximal leader and for computing a minimal path wrt a partial order $\leq$ instead of a total order.
- Reuse the leader election ASM to define an algorithm which, given the leader, computes for each agent the distance (length of a shortest path) to the leader and a neighbor where to start a shortest path to the leader.
  - Hint: See Reisig op.cit. Fig.32.2

- Definition of Async ASMs
- Small Examples
  - Mutual Exclusion
    - Dining Philosophers
    - Multiple Read One Write
  - Network Algorithms
    - Master Slave
    - Consensus
    - Load Balance
    - Leader Election
    - Echo See Reisig 1998, Sect. 33, 77
    - Phase Synchronization
- Simple Properties of Async ASMs
- References

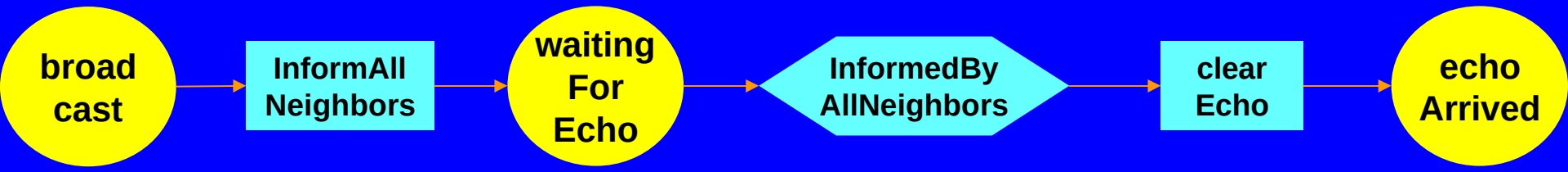
Echo Algorithm : problem statement

- Goal: Design a distributed algorithm that guarantees an initiator's message being broadcast and acknowledged ('echoed') through a finite connected network, using only communication between neighbors
- Algorithmic Idea:
  - the initiator (a distinguished node) broadcasts an info to all his neighbors and waits for their acknowledgements
  - every node which has been informed by some neighbor ('parent') node informs all his other neighbor nodes and waits for their acknowledgements to come back, to then forward his own acknowledgement back to the parent node

Echo ASM: Agent Signature

- Agent : connected graph of agents
  - initiator: Agent distinguished element with control states {broadcast, waitingForEcho, echoArrived} and special ASM
- Each agent is equipped with:
  - neighb $\subseteq$ Agent (external function)
  - informedBy : Agent $\rightarrow$ Bool (abstract message passing)
    - indicating whether a message (with info or acknowledgement) from a neighbor agent has been sent (arrived)
  - parent : Agent (building a spanning tree for acks)
    - yields a neighbor node who has sent a messg which is to be acknowledged, once all other neighbor nodes have acknowledged this mssg which has been forwarded to them
  - ctl_state:{listeningToInfo,waitingForAck,informed }
- Initially initiator in ctl_state = broadcast, for all other agents:
  - ctl_state = listeningToInfo, informedBy everywhere false, parent = undef

Initiator ASM for Echo algorithm



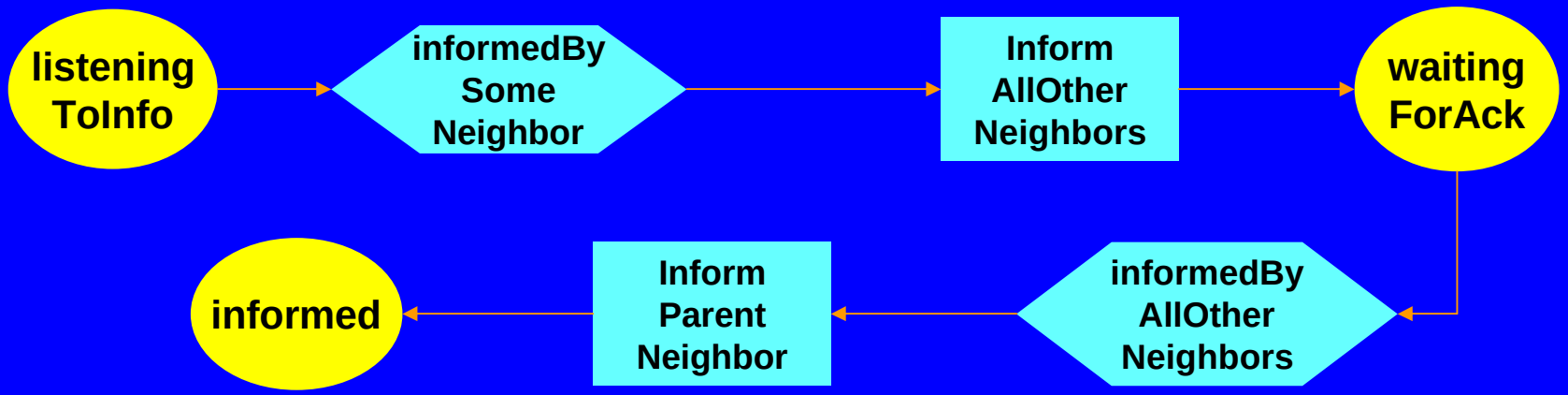
InformAllNeighbors ° **forall** u ∈ neighb u.informedBy(self):=true

InformedByAllNeighbors ° forall u ∈ neighb informedBy(u) = true

clearEcho ° forall u ∈ neighb informedBy(u) : = false

NB. Clear info when it gets redundant to prepare for later iterations

Echo Agent ASM for Agent ≠ Initiator



informedBySomeNeighbor ° for some u ∈ neighb informedBy(u) = true

InformAllOtherNeighbors °
choose u ∈ neighb **with** informedBy(u) **in**
forall v ∈ neighb-{u} v.informedBy(self) := true
informedBy(u):= false parent := u

informedByAllOtherNeighbors °
forall v ∈ neighb-{parent}
informedBy(v) = true

InformParentNeighbor °
parent.informedBy(self) := true
clearAcknowledgement

clearAcknowledgement °
forall u∈ neighb - {parent} informedBy(u) : = false
parent:= undef

Async Echo ASM (initiator & agents): Correctness

- Proposition: In every run of a set of agents including an initiator, all equipped with the corresponding echo ASMs:
  - the initiator terminates (termination)
  - the initiator terminates only when all other agents have been informed about its originally sent mssg (correctness)
- Proof. Follows by run induction from two lemmas.

Echo ASM (initiator and agents): Correctness

- Lemma 1. In every run of the async echo ASM, each time an agent executes InformAllOtherNeighbors, in the spanning tree of agents waitingForAck the distance to the initiator grows until leafs are reached.
  - Proof. By downward induction on echo ASM runs
- Lemma 2. In every run of the async echo ASM, each time an agent executes InformParentNeighbor, in the spanning tree the distance to the initiator of nodes with a subtree of informed agents shrinks, until the initiator is reached.
  - Proof. By upward induction on runs of the async echo ASM

Exercise

- Refine the asynch echo ASM to the case of a network with more than one initiator.
  - Hint: Use a pure data refinement, recording the initiator's identity when informing neighbors and letting the initiator wait for the echo to is own initiative.

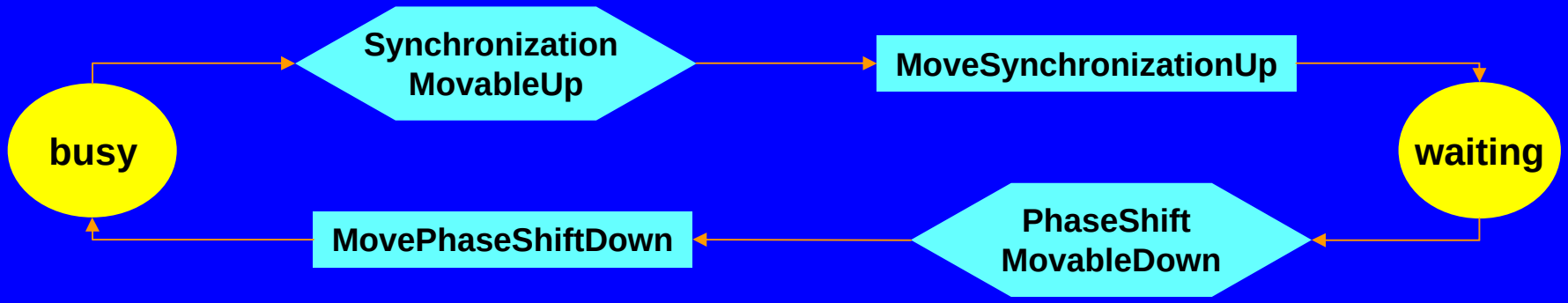
- Definition of Async ASMs
- Small Examples
  - Mutual Exclusion
    - Dining Philosophers
    - Multiple Read One Write
  - Network Algorithms
    - Master Slave
    - Consensus
    - Load Balance
    - Leader Election
    - Echo
    - Phase Synchronization See Reisig 1998, Sect. 36 (Fig.36.1), 81
- Simple Properties of Async ASMs
- References

Phase Synchronization : problem statement

- Goal: Design a distributed algorithm that guarantees synchronized execution of phases of computations, occurring at nodes of an undirected tree (connected acyclic network), using only communication between tree neighbors
- Algorithmic Idea: every agent (tree node) before becoming busy in a new phase has to
  - synchronize with its neighbor nodes, moving the synchronization up along the tree structure
  - wait until all agents are waiting for the phase increase to start
  - increase its phase - when the phase shift becomes movable down - and move the phase shift further down along the tree structure
    - until all nodes have become busy in the new phase

Phase Synchronization ASM: Agent Signature

- Agent : an undirected tree of agents equipped with:
  - neighb $\subseteq$ Agent (external function)
  - phase: the current phase
  - synchPartner, waitPartner: Agent
    yields the neighbor node an agent is synchronized with (in moving the synchronization up or down the tree) resp. waiting for (to reverse the previous upward synchronization downwards)
  - ctl_state : {busy, waiting }
- Initially
  - ctl_state = busy, phase = 0, synchPartner = waitPartner = undef



SynchronizationMovableUp ° ∃ y∈ neighb SynchronizationMovableUpTo(y)

for some neighb y all other neighbors synchronize with self in phase(self)

MoveSynchronizationUp ° **choose** y∈ neighb with SynchronizationMovableUpTo(y)

synchronize self with y in phase(self) MoveSynchronizationUpTo(y)

SynchronizationMovableUpTo(y) ° ∀z∈ neighb-{y}

z.synchPartner(phase(self)) = self

PhaseShiftMovableDown

waitPartner synchronizes with self in phase(self)

MoveSynchronizationUpTo(y) ° ∀z∈ neighb-{y}

z.synchPartner(phase(self)) := undef

self.synchPartner(phase(self)) := y, self.waitPartner(phase(self)) := y

MovePhaseShiftDown ° phase : = phase +1

∀z∈ neighb-{waitPartner} self.synchPartner(phase(self)) : = z

waitPartner.synchPartner (phase(self)):= undef

Phase Synchronization ASM : Correctness & Liveness

- Proposition: In every infinite run of the asynch phase synchronization ASM:
  - in any state any two busy agents are in the same phase (correctness property)
  - if every enabled agent will eventually make a move, then each agent will eventually reach each phase (liveness property)

Phase Synch: **PhaseShiftMovableDown Lemma**

- Lemma: For every phase p, whenever in a run a state is reached where for the first time

  u.waitPartner(p)=v synchronizes with u in phase p,

  every element of

  subtree(u,v) ∪subtree(v,u)∪{u,v}

  is waiting in phase p

  where subtree(x,y) = {n| n reachable by a path from x without touching y}

Proof of **PhaseShiftMovableDown Lemma**

- Proof of the lemma follows from the following two claims.
- Claim 1. When Synchronization is moved up in phase(u)=p from busy u to to v=u.synchPartner(p), the elements of subtree(u,v) are waiting, u becomes waiting, and they all remain so until the next application of a Shift rule "MovePhaseShiftDown".
  - Claim 1 follows by induction on the applications of the Synchronization rule.
    - n=1: true at the leaves
    - n+1: follows by induction hypothesis and Synchronization rule from

      subtree(u,v)= $\dot{E}_{i<n}$ subtree(u_i,u) $\cup$ {u_0,...,u_{n-1}} for neighb(u}= {u_0,...,u_n}
      with v= u_n (by connectedness)

Proof of **PhaseShiftMovableDown Lemma**

- Claim 2. For every infinite run and every phase p, a state is reached in which all agents are waiting in phase p and for some agent u its waitPartner in phase p synchronizes with u in phase p.
  - Follows by induction on p.

- Definition of Async ASMs
- Small Examples
  - Mutual Exclusion
    - Dining Philosophers
    - Multiple Read One Write
  - Network Algorithms
    - Master Slave
    - Consensus
    - Load Balance
    - Leader Election
    - Echo
    - Phase Synchronization
- Simple Properties of Async ASMs
- References

Analyzing Initial segments of runs of an async ASM

- Let m be a move of an agent in a run of an async ASM.
- Let Post(m) = {I| I initial segment & m is maximal in I}. Post (m) represents the multiplicity of states, resulting from the last move m and each depending via $\sigma(I) = \sigma(I-\{m\})$ from its "pre"-state.
- Let Pre(m) = {I-{m}| I $\in$ Post(m) } representing the multiplicity of states in which move m can be performed.
- **Pre(m)-Lemma.** Pre(m) has minimal and maximal elements:

  min Pre(m) ={m'|m' < m} max Pre(m) ={m'|not m' $\geq$ m}

  Pre(m) = {I| I initial segment & min Pre(m) $\subseteq$ I $\subseteq$ max Pre(m) }.

  Therefore Pre(m) is closed under union and intersection.

Characterizing Concurrent Moves in Runs of an Async ASM

- **Concurrent-Moves Lemma.** In runs of an async ASM, the concurrency (incomparability by the run order) of moves m,m' of agents is equivalent to any of the following:
- $\text{minPre}(m) \cup \text{minPre}(m') \in \text{Pre}(m) \cap \text{Pre}(m')$ both moves can come after all the moves which come before any of the two
- $\text{Pre}(m) \cap \text{Pre}(m') \neq \emptyset$
- There exist $I,J \in \text{Pre}(m)$ with $m' \in I-J$

  before one of the moves, the other may be executed or not
- Proof. If m,m' are concurrent, then $I = \text{minPre}(m) \cup \text{minPre}(m')$ is an initial segment and $\text{minPre}(m) \subseteq I \subseteq \text{maxPre}(m)$, so that $I \in \text{Pre}(m)$. By symmetry then $I \in \text{Pre}(m')$.
- Let $J \in \text{Pre}(m) \cap \text{Pre}(m')$ and set $I = J \cup \{m'\}$. Then $I \in \text{Pre}(m)$.
- Suppose $I,J \in \text{Pre}(m)$ with $m' \in I-J$. $m'<m$ would imply $m' \in \text{minPre}(m) \subseteq J$ and thus $m' \in J$. $m<m'$ would imply $m' \notin \text{maxPre}(m) \supseteq I$ and thus $m' \notin I$. Therefore m, m' are incomparable, i.e. concurrent.

Conditions for indisputability of terms

- **Sufficient indisputability condition.** If $t_{Pre(m)}$ is not indisputable, then there is a move m' concurrent with m which may change t.
  - Proof by contradiction. By the Pre(m)-lemma, for every $I \in Pre(m)$ one can reach $\sigma(I)$ from $\sigma(minPre(m))$ by applying moves concurrent to m. By assumption none of them may change t, so that $Val_{\sigma(I)}(t) = Val_{\sigma(minPre(m))}(t)$. That would mean that $t_{Pre(m)}$ is indisputable.
- **Necessary indisputability condition.** If there is a move m' concurrent with m which must change t, then $t_{Pre(m)}$ is not indisputable.
  - Proof . By the concurrent-moves lemma, there is a common initial segment $I \in Pre(m) \cap Pre(m')$. Then also $I \cup \{m'\} \in Pre(m)$, and since m' must change t, I and $I \cup \{m'\}$ have different values for t.

Stability and change of term values in initial segments

- Extending term evaluation from states to state sets:
  - $\mathrm{Val}_{\mathrm{Pre}(m)}(t) = c$ if all initial segments in Pre(m) have the same value c for t: $\forall\, I \in \mathrm{Pre}(m)$: $\mathrm{Val}_{\sigma(I)}(t) = c$

    We then say that $\mathrm{Val}_{\mathrm{Pre}(m)}(t)$ is indisputable, writing also that $t_{\mathrm{Pre}(m)}$ (or its value) is indisputable
  - $\mathrm{Val}_{\mathrm{Pre}(m)}(t)$ = undef otherwise
- Defn: move m may change the value of t iff m changes the value of t in some linearization of the given run, formally:

$$\text{for some } I \in \mathrm{Pre}(m)\text{: } \mathrm{Val}_{\sigma(I)}(t) \neq \mathrm{Val}_{\sigma(I \cup \{m\})}(t)$$

- Defn: m must change the value of t iff m changes the value of t in every linearization of the given run, formally:

$$\text{for every } I \in \mathrm{Pre}(m)\text{: } \mathrm{Val}_{\sigma(I)}(t) \neq \mathrm{Val}_{\sigma(I \cup \{m\})}(t)$$

Moves with indisputable different term values compare

- **Lemma.** If the values of $t_{Pre(m)}$ and of $t_{Pre(m')}$ are different but both indisputable, then m and m' are not concurrent but compare in the run order (i.e. m<m' or m'<m).
- Proof. If $t_{Pre(m)}$ and $t_{Pre(m')}$ are indisputable and m,m' are concurrent, then by the concurrent-moves lemma there is a common initial segment $I \in Pre(m) \cap Pre(m')$ so that $t_{Pre(m)} = Val_{\sigma(I)}(t) = t_{Pre(m')}$

Focused Terms

- Defn: A term t is focused in a given run if any move which may change its value must change it.
- Sufficient Condition for Focus. If the value of a term t may be changed only by one agent a, then it is focused.
- Proof. It suffices to show that $t_{Pre(m)}$ is indisputable
  - because then $t_{Post(m)}$ is also indisputable, and the assumption that only a may change the value of t implies: $t_{Post(m)}$ is different from $t_{Pre(m)}$ iff m changes t

  By the sufficient indisputability condition, $t_{Pre(m)}$ is indisputable because moves concurrent to m must be moves of other agents than a, therefore by assumption none of them may change t.
- Synchronization Lemma. For a focused term t, $t_{Pre(m)}$ is indisputable iff , in the given partial order, m compares with all moves which change the value of t.

  Proof: follows from the indisputability and focus conditions.

References

- W.Reisig: Elements of Distributed Algorithms
  Springer-Verlag 1998
- Y. Gurevich and D. Rosenzweig: Partially Ordered Runs: A Case Study.
  - In: Abstract State Machines. Theory and Applications. (Ed. Y.Gurevich et al). Springer LNCS 1912, 2000, 131-150
- E. Börger, R. Stärk: Abstract State Machines. A Method for High-Level System Design and Analysis Springer-Verlag 2003, see http://www.di.unipi.it/AsmBook