Egon Börger (Pisa)

Leader Election

Ground models and their refinement to CoreASM executable models

Dipartimento di Informatica, Università di Pisa, Italy
boerger@di.unipi.it

See Modeling Companion Ch. 4.3 (refinement variations of the PROXY programming pattern)

4 examples

We illustrate using ASMs for various leader election requirements

- to proceed from natural language requirements to precise models
  - a ground model one can justify to be faithful to the requirements
- to include additional requirements by ASM refinements
  - which one can prove to be correct wrt the ground model

The requirements consider the following backgrounds (data structures) and computation models:

- synchronous leader election in a *ring*
- synchronous leader election in a *connected graph* with known diameter
- *asynchronous* leader election in a connected graph
- asynchronous leader election in a connected graph with computation of a *minimal path* to leader

ExI.1: Synchronous leader election in a ring: requirements

... LCR algorithm in honor of Le Lann, Chang, and Roberts ... uses only *unidirectional communication* and does not rely on knowledge of the size of the *ring*. Only the leader performs an output. The algorithm uses only *comparison operations* ...
(Quote from N. Lynch: Distributed Algorithms (1996), Sect. 3.3)

LCR algorithm (informal):

Each process *sends* its identifier around the ring. When a process *receives* an incoming identifier, it compares that identifier to its own. If the incoming identifier is greater than its own, it keeps passing the identifier; if it is less than its own, it discards the incoming identifier; if it is equal to its own, the process declares itself the leader. (ibid.)

From English to ASM: States of LCR (signature)

- finite static set of $p \in Process$es ('agents') of cardinality > 1
- static *linear order* $< \subseteq Process \times Process$ (for comparison operations)
- static *ring structure* (for unidirectional communication)
  - e.g. bijective function $rightNeighb : Process \rightarrow Process$ s.t. $rightNeighb(p) \neq p$
- each p can send msgs to its $rightNeighb$ and receive msgs from its left neighbor in the ring

Each agent p has:

- a $mailbox$, here also called $Proposals$, assumed to be initially empty
- an output location $leader$, initially $leader = unknown$

Agents are assumed to be synchronized in rounds: we *model one round as one step of a parallel ASM* SyncLcr.

Msg delivery is assumed to be reliable: msgs sent in round r are received in round $r + 1$.

From English to ASM: LCR algorithm

Each process sends its identifier around the ring. When a process receives an incoming identifier, it compares that identifier to its own. If the incoming identifier is greater than its own, it keeps passing the identifier; if it is less than its own, it discards the incoming identifier; if it is equal to its own, the process declares itself the leader.

This is reflected by successive machine steps:

- to first PROPOSE itself as a potential leader by sending—here once—(an id of) **self** as leader info to the respective neighbor:

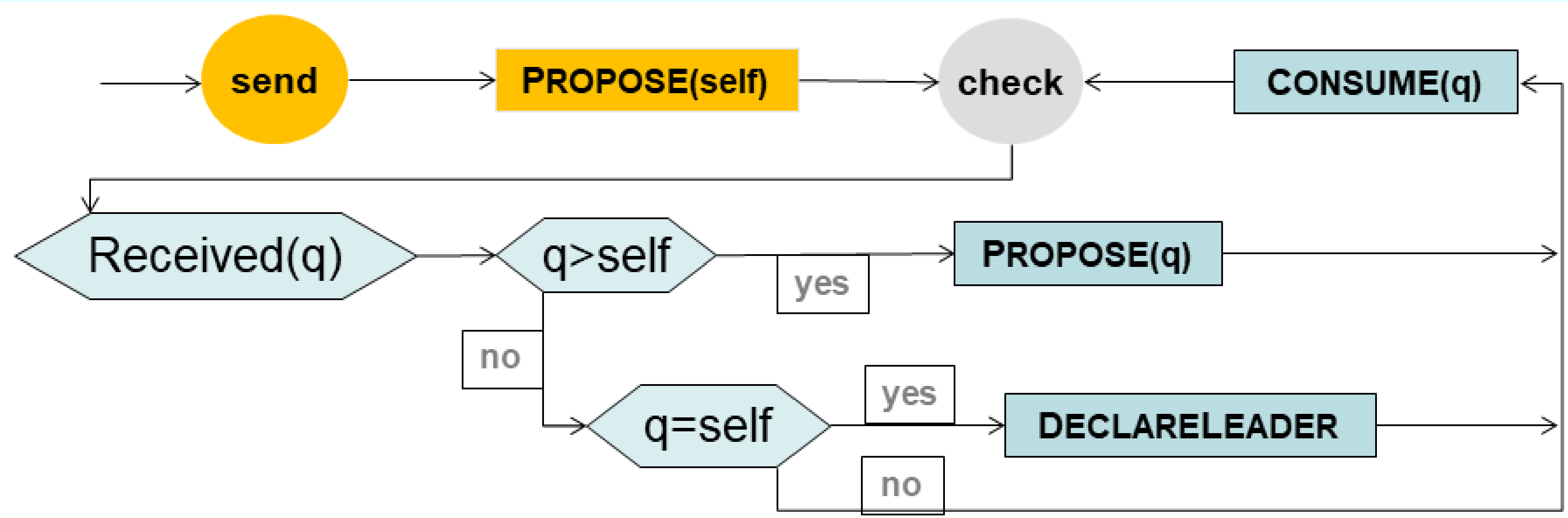
  $\textsc{Propose}(\textbf{self}) = \textsc{Send}(\textbf{self}, \textbf{to } rightNeighb)$

- to then repeatedly CHECK&UPDATELEADERKNOWLedge, on the basis of the received msg, until $leader \neq \mathbf{undef}$:

  if $Received(q)$ **then**
  if $q >$ **self** **then** $\textsc{Propose}(q)$ -- forwards local leader info
  if $q =$ **self** **then** $leader :=$ **self** -- declares itself the leader
  $\textsc{Consume}(q)$

Flowchart definition of LCR

$\textsc{SyncRingLeaderElect} = \textbf{forall } p \in \mathit{Process}\ \textsc{Lcr}(p)$



where

$\textsc{Propose}(p) = \textsc{Send}(p, \textbf{to } \mathit{rightNeighb}(\textbf{self}))$

$\textsc{DeclareLeader} = (\mathit{leader} := \textbf{self})$

NB. For conceptual economy, $p \in \mathit{Process}$ are used as 'identifiers'.

SyncRingLeaderElection correctness

Correctness Property. Each SyncRingLeaderElection run, started in an initial state, leads in finitely many steps to (a state where)

- $leader_{max(Process)} = max(Process)$ ('output') and
- $leader_q = unknown$ ('no output') for every other q

with all processes in mode $check$ and empty mailbox $Proposals$.

Proof by induction. Use that in each round after the first Propose(**self**) step, each time a process p receives a Proposed message $q < p$, this msg is not forwarded and the overall number of sent msgs, which never increases, decreases at least by one. See Lynch p.29-30.

ExI.2: Sync leader election in a connected graph

The signature (background) of $\text{L}\textsc{cr}$ is adapted as follows:

- finite *directed graph* $(Process, Edge)$, assumed to be connected
  − i.e. there is a path from each $p \in Process$ to each $q \in Process$
- *communication only bw neighbors* (i.e. along edges): $Neighb(p)$ is the set of the processes p is linked to by a directed edge outgoing p
- let *diameter* $= max\{distance(p, q) \mid p, q \in Process\}$ (derived location), $distance(p, q) =$ length of shortest path from p to q. $diameter$ is a static location every process can read.

Requirement (Lynch 4.1): Every process should eventually set its $status$ to $Leader$ or $NonLeader$, only $max(Process)$ to $Leader$ wrt the linear order $<$ of $Process$es. Initially $status = unknown$.

Algorithmic idea for sync leader election in connected graphs

Idea (Lynch 4.1): each $p \in Process$

- keeps its current (local) leader knowledge in a location *cand* (‘greatest process seen so far’)
- in synchronized rounds CHECKs the received $Proposals$ (of current $cand$s of its $Neighbor$s) to UPDATELEADERKNOWL
  - which includes to PROPOSE the updated $cand$ value to its $Neighbor$s
- the algorithm will stop after $diameter$ rounds
  - again for simplicity we model rounds by parallel ASM steps

Extend signature by:

- controlled location *cand*, for each p, denoting the local leader knowledge which p sends to its $Neighbor$s; initially $cand_p = p$
- counter *round*, initially $round = 0$

Initially $cand_p = p$, $round = 0$, $Proposals = \emptyset$

Reusing LCR components for connected graphs

Reuse PROPOSE and CHECK&UPDATELEADERKNOWL:

- CHECK&UPDATELEADERKNOWL is adapted to
  - check a mailbox with possibly multiple msgs, from all neighbors
    - instead of only one msg from the left ring neighbor
  - update local leader knowledge $cand$, by $max(\{cand\} \cup Proposals)$, and forward it
    - instead of forwarding a larger identifier, received from the left ring neighbor, to the right ring neighbor
  - DECLARELEADER sets $status$ of $leader = max(Process)$ to $Leader$ and $status$ of each $p \neq leader$ to $NonLeader$
- PROPOSE is adapted to SEND
  - the updated $cand$ value (updated on the basis of $Proposals$)
  - to each $q \in Neighb$
    - instead of one received identifier to the $rightNeighb$

Definition: SyncGraphLeaderElect

if $round < diameter$ **then**
  CHECK&UPDATELEADERKNOWL
  INCREMENT($round$)
if $round = diameter$ **then** DECLARELEADER

CHECK&UPDATELEADERKNOWL =
  if $Proposals \neq \emptyset$ **then**
    let $q = max(\{cand\} \cup Proposals)$ -- choose greatest element
      $cand := q$ PROPOSE(q) EMPTY($Proposals$)
PROPOSE(q) = **forall** $p \in Neighb$ SEND(q, **to** p)
DECLARELEADER =
  if $p = cand$ **then** $status := Leader$ **else** $status := NonLeader$
  INCREMENT($round$) -- added to stop the run

Correctness of SyncGraphLeaderElection

Correctness Statement (rephrased from Lynch, Theorem 4.1. p.53): In SyncGraphLeaderElection runs, within $diameter$ rounds, $max(Process)$ outputs $status = Leader$ and each other process $status = NonLeader$.

The proof uses the following **Lemma**:

forall $r \leq diameter$ **forall** $p, q \in Process$
if $distance(p, q) \leq r$ **then** $cand_q \geq p$ holds after r steps

Proof of the lemma by induction on r.

By the lemma, for each q after $diameter$ steps holds:

$$cand_q \geq max(Process) \geq_{by\ definition} cand_q$$

NB. For a run illustration see Lynch, Course 6.852, Fall 2013, Lect.2, slides 28-39

ExI.3: Async Leader Election in connected graphs

PlantReq. Consider a network of finitely many linearly ordered $Process$es without shared memory, located at the nodes of a directed connected graph and communicating asynchronously with their neighbors (only).

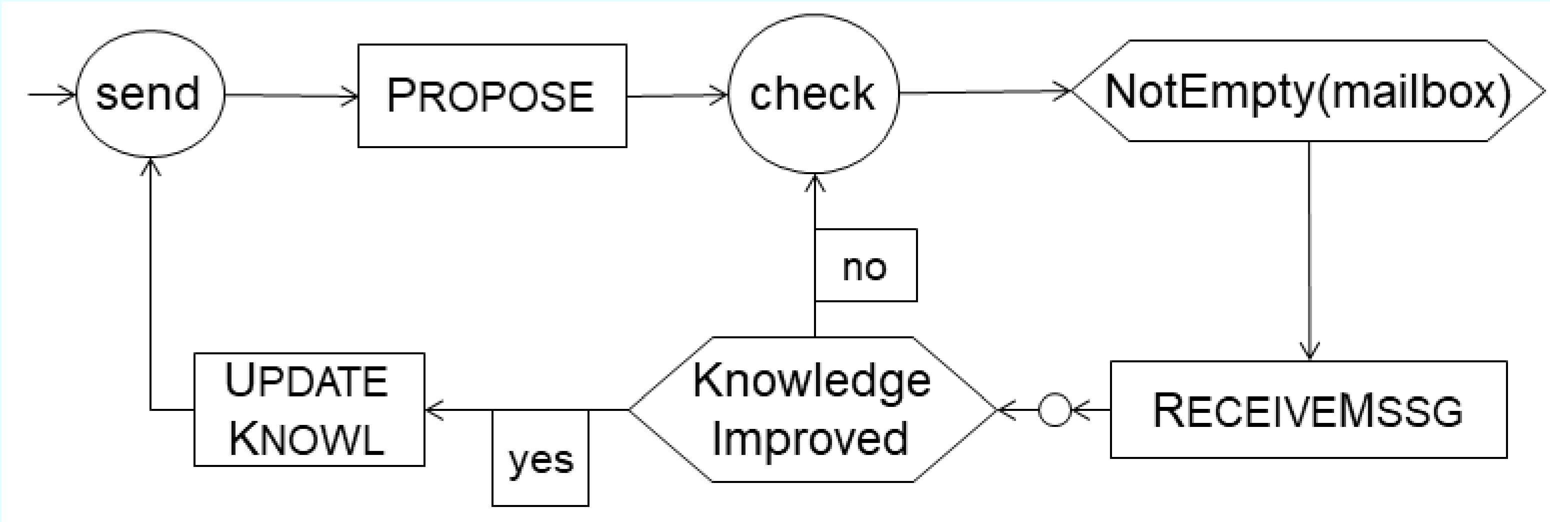
FunctionalReq. Design and verify a distributed algorithm whose execution lets every process know the leader.

Algorithmic idea:

StepReq. Every process p maintains its leader knowledge in a record, say $cand$ (also written $cand_p$), denoting the greatest process it has seen so far (initially itself). The process p alternates between:

- SEND $cand_p$ to all its $Neighb$ors
- RECEIVEMSG q from some neighbor and UPDATEKNOWL in case p's $KnowledgeImproved$ by the received leader information q being larger than $cand_p$.

LeadElect flowchart for AsyncGraphLeaderElect



$\text{PROPOSE} =$ **forall** $q \in Neighb$ $\text{SEND}(cand,$ **to** $q)$[1]

$\text{RECEIVEMSG} =$ **choose** $q \in mailbox$ -- check msgs one by one

$curMsg := q$

$\text{CONSUME}(q)$

$KnowledgeImproved$ iff $curMsg >$ **self**

$\text{UPDATEKNOWL} = (cand := curMsg)$

[1] Figure © 2003 Springer Berlin-Heidelberg, reused with permission.

AsyncGraphLeaderElection Behavior Property

let $\textsc{AsyncGraphLeaderElect} =$

$(p, \textsc{LeadElect}_p, mailbox_p)_{p \in Process}$

In properly initialized concurrent $\textsc{AsyncGraphLeaderElect}$ runs, with reliable communication and without infinitely lazy components,

- i.e. every enabled process will eventually make a move

eventually for every $p \in Process$ holds:

- $cand = max(Process)$ (everybody 'knows' the leader wrt $<$)
- $mailbox = \emptyset$ (there is no more communication)
- $mode = check$

AsyncGraphLeaderElection Behavior Property: Proof

Index the elements of $Process$ with order-reflecting increasing indeces $p_0 < p_1 < \ldots < p_{Max}$.

Consider any run and any $p \in Process$.

Each UpdateKnowl-step of p in the run decreases the discrepancy $Max - index(cand_p)$ between the real $leader$ and the leader knowledge $cand_p$ of p.

When the discrepancy becomes 0 for every $p \in Process$ the claim follows (by induction).

ExI.4: Async Leader Election with minimal path computation

Additional requirement (BellmanFord algorithm in Lynch 4.3):

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 (via such a neighbor)

Algorithmic idea: add to $cand$ a $nearNeighbor$ with minimal $distance$ to the leader candidate

Additional signature for each process:

- $nearNeighb \in Process$ (initially $nearNeighb =$ **self**)
- dynamic location $distance \in Distance$ (initially $distance = o$) with static $distance(a, b)$ function for neighbors a, b

Refine LeadElect components by path info

Messages become triples $(cand, nearNeighb, dist)$ whose components are retrieved by functions $fst, snd, third$:

$\text{Propose} =$ **forall** $q \in Neighb$

$\quad \text{Send}((cand, nearNeighb, distance + distance(\textbf{self}, q)), \textbf{to}\ q)$

$KnowledgeImproved$ **iff** $fst(curMsg) >$ **self** **or**

$\quad fst(curMsg) =$ **self** **and** $third(curMsg) < distance$

$\text{UpdateKnowl} =$

$\quad cand := curMsg$

$\quad nearNeighb := snd(curMsg)$

$\quad distance := third(curMsg)$

NB. This is a pure data refinement of operations and predicates. The refinement type (1,1) leaves the program control flow (the above flowchart for LeadElect) unchanged.

AsyncGraphMinPathToLeader **Behavior Property**

Define AsyncGraphMinPathToLeader to be the same as AsyncGraphLeaderElect but with refined LeaderElect components.

Enrich the AsyncGraphLeaderElection behavior property by:

... eventually for every $p \in Process$ holds:

- $cand = max(Process)$
- $distance =$ *minimal distance of a path from agent to* $leader$
- $nearNeighbor =$ *a neighbor on a minimal path to the* $leader$ (except for $leader$ where $nearNeighbor = leader$)
- $mailbox = \emptyset$
- $mode = check$

Proof of AsyncGraphMinPathToLeader Behavior

Use an induction as for SyncGraphLeaderElect, adding a *side induction on $distance$*.

The side induction works when a process p checks a $curMsg$ from a neighbor q which

- proposes as leader $cand$ but with $dist < distance(p)$

Then p learns a shorter path to (the up to now best known $cand$ for the) $leader$, going through q as new $nearNeighb$.

- NB. compositional proof method resulting from conservative ASM refinement concept (incremental modular extension)

CoreASM refinement

For a refinement to CoreASM and some characteristic runs see the CoreASM code, developed in May 2011 by Julian Lettner (FH Hagenberg, Austria). See

- https://github.com/CoreASM/coreasm.core/wiki/Examples
- the Modeling Companion website http://modelingbook.informatik.uni-ulm.de.

Observation

Interpretation of 'every process knows the leader' as 'eventually $cand = max(Process)$' is not a solution that really satisfies in the context of distributed (truly concurrent) computation

- processes do not know *when* they know the leader

Ring background structure helps to detect termination.

Exercise: Refine LEADELECT with the graph background structure to a machine which also detects the termination of the asynchronous run in case the number of participating processes is known to each process.

References

- Ephraim Korach, Shay Kutten, Shlomo Moran. *A Modular Technique for the Design of Efficient Distributed Leader Finding Algorithms*. ACM Transactions on Programming Languages and Systems 12 (1), 1990
- An event-B development appeared in:
  J.-R.Abrial, D. Cansell, D. Mery: *A mechanically proved and incremental development of IEEE 1394 tree identify protocol*. Formal Aspects of Computing 14 (2003) 215-227
- Nancy A. Lynch, *Distributed Algorithms*. Morgan Kaufmann, San Franciso 1996. See also material for course 6.852 at MIT, Fall 2013
- E. Börger and A. Raschke: *Modeling Companion for Software Practitioners*. Springer 2018
  See website `http://modelingbook.informatik.uni-ulm.de`

Copyright Notice