Egon Börger (Pisa) & Alexander Raschke (Ulm)

Communicating ASMs

illustrated by modeling monitoring network runs

Università di Pisa, Dipartimento di Informatica, boerger@di.unipi.it
Universität Ulm, Abteilung Informatik, alexander.raschke@uni-ulm.de

See Ch. 3.3 of Modeling Companion
http://modelingbook.informatik.uni-ulm.de

Goal of the lecture

- *define communicating ASMs*, i.e. multi-agent ASMs without shared memory whose actions are
  - either **local actions**, affecting only each agent's local state
  - or **inter-process communication** actions, i.e. sending/receiving msgs
- *illustrate their use* to model monitoring network runs (i.e. runs of concurrent ASMs which communicate with their neighbors) for
  - concurrent leader election
    - $\textsc{GraphLeadElect}$ (Exl.1)
  - termination detection of diffusing system runs
    - $\textsc{TerminationDetector}(\mathcal{D})$ (Exl.2)
    - $\textsc{TerminationDetector}(\textsc{GraphLeadElectDiffuse})$
  - concurrent (asynchronous) emulation of synchronous process runs
    - $\textsc{ConcurSyncEmulator}(Process, Edge)$ (Exl.3)

Definition of communicating ASMs

A system of *communicating* ASMs is defined as multi-agent ASM of components $p = (ag(p), pgm(p), mailbox(p))$ for $p \in Process$ where:

- the signatures are pairwise disjoint so that each agent has its own private state, also called *internal state* or *local state*,
- each agent is enriched by a $mailbox$ for incoming messages,
- each program $pgm(p)$ may contain, besides the usual ASM constructs, the abstract communication actions $\textsc{Send}(message)$ and $\textsc{Consume}(message)$ and the $Received(message)$ predicate.

Usually $Process$ is assumed to be finite (unless otherwise stated).

The notion of run is that of concurrent ASM runs. Due to the absence of shared locations (besides $mailbox$ which p shares with the communication medium), in such asynchronous runs assume wlog that

each step is an atomic ASM $\textsc{Read}\&\textsc{Write}$ step

including independent actions $\textsc{Send}$To/$\textsc{Receive}$From a $mailbox$.

Keeping communication actions abstract

We use communication constructs which deliberately abstract from communication channels. Their intended interpretation is as follows:

- $\textsc{Send}(m, \textbf{to } q)$ means to transfer the message $(m, \textbf{from self}, \textbf{to } q)$ to the communication medium whose job is to deliver it to $mailbox(q)$
- $Received(msg)$ **iff** $msg \in mailbox(\textbf{self})$
  - i.e. msg has been delivered to its destination by an interaction bw communication medium and receiver or some delegate, etc.
    - leaving to specify when/how to retrieve msgs from the mailbox
- $\textsc{Consume}(msg) = \textsc{Delete}(msg, mailbox(\textbf{self}))$
- $mailbox$ (or $inbox$, $outbox$) treated as set, unless otherwise stated
  - e.g. as multiset, FIFO-queue, priority queue, etc.

The components of a $msg = (m, \textbf{from } p, \textbf{to } q)$ are extracted by functions $payload(msg)$ (msg content), $sender(msg)$, $receiver(msg)$.

Some variations of delivery

Depending on properties of the communication medium:

- *Immediate reliable communication*: every message sent in one 'step' (e. g. in a synchronous round) is in the receiver's mailbox at the beginning of the next step (e. g. in the next synchronous round).
- *Reliable communication*: every message sent in one 'step' eventually arrives in the receiver's mailbox.
- *Eventually reliable communication* (asynchronous computation model) where either no message is lost or where multiple delivery attempts (message repetition) are performed, assuming that at least one of them eventually succeeds.
- *Lossy uncorrupted communication* (in the asynchronous model), where messages can get lost but not corrupted.
- *Lossy corrupted communication* (in the asynchronous computation model), where messages can get lost or be delivered corrupted.

A notational abbreviation

if $Received(msg(params))$ **then** $M(msg(params))$

abbreviates:

if thereissome $msg(params) \in mailbox$ **then**

choose $m \in \{msg(params) \mid msg(params) \in mailbox\}$ **in** $M(m)$

When the order of messages is relevant, we still abstract notationally from the order-reflecting $next$ function, which retrieves the next message from the $mailbox$. In that case

if $Received(msg(params))$ **then** $M(msg(params))$

stands for the following rule:

if $m = next(mailbox)$ **and forsome** $params\ \ m = msg(params)$

then $M(m)$

Disjoint signature assumption

- Communicating ASMs $(a, M, mailbox(a)), (b, N, mailbox(b))$ may have the same program $M = N$.
  - To guarantee disjoint locations (read: local states) we assume all function symbols f as implicitly parameterized ('instantiated') by the executing agents a, b in the form f_a, f_b.
    - This parameterization to partition states is used in full generality by what we call ambient ASMs (see Ch.4).
- Communicating ASMs may have input or output locations, but those are used only for providing input or output from/to the environment (if any) and not for inter-process communication (in case the environment is not seen as an additional process).

Exl.1: Concurrent Leader Election Requirements

Typical *LeaderElectionReq*uirements:

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.

Corresponding signature elements and constraints:

- Finite connected graph $(Process, Edge)$ (static) with sets $Neighb(p)$ $-$ set of neighbors q linked to $p \in Process$ by $(p, q) \in Edge$
- linear order $<$ (static) of $Process$es (NB. processes used as nodes)
- $mailbox(p)$, shared with communication medium, for each $p \in Process$; no other shared locations
- $\textsc{Send}(msg, \textbf{from } p, \textbf{to } q)$ implies $q \in Neighb(p)$
- $msg \in mailbox(p)$ implies $p \in Neighb(sender(msg))$

Design idea for LeaderElectionReq

A typical idea to design such an algorithm (see Lynch 1996, Sect.15.2):

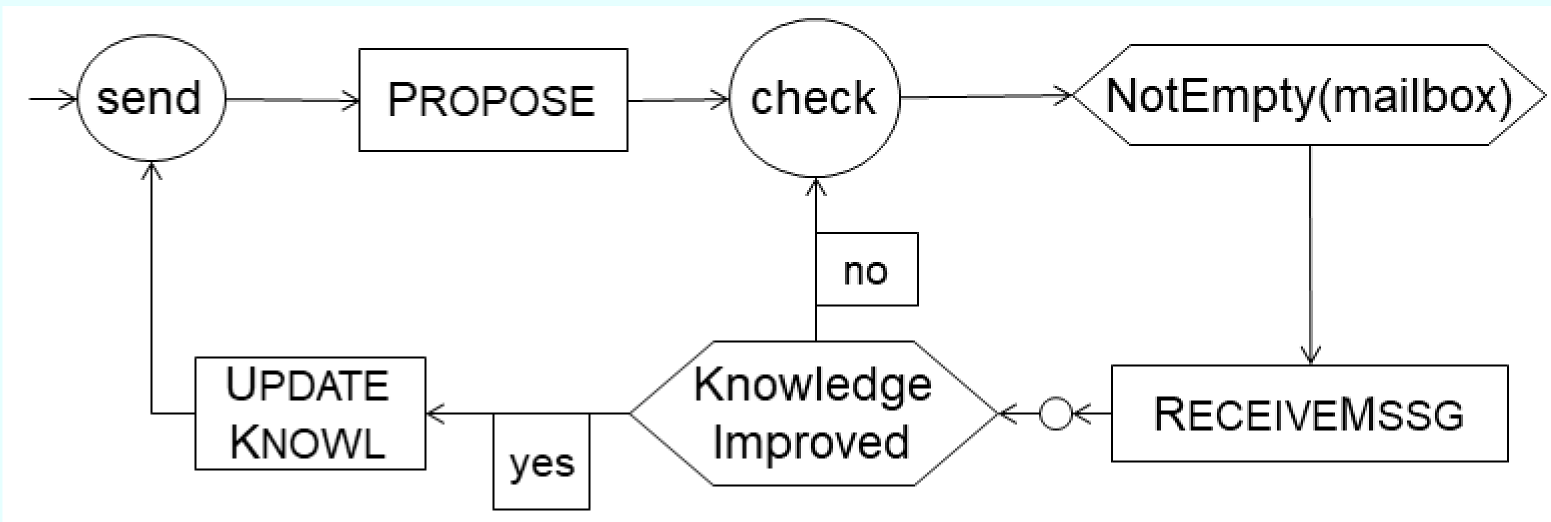
- *StepReq*. Every process maintains a record, say $cand$, of the greatest process it has seen so far, initially its own. It alternates between:
  - sending $cand$ to all its neighbors
  - updating $cand$ in case its knowledge has been improved by receiving a message (say $curMsg$) with larger value from some neighbor.

Corresponding additional signature elements for each $p \in Process$, all three controlled by p:

- $mode \in \{send, check\}$
- $cand \in Process$
- $curMsg$

A system GraphLeadElect of communicating ASMs

We use process names $p \in Process$ as agent name and equip each p with an instance of the communicating ASM program LeadElect defined by the following control state ASM which implements the above design idea:[1]



$$\text{GraphLeadElect} = (M_p)_{p \in Process}$$

where $M_p = (p, \text{LeadElect}_p, mailbox_p)$

[1] Figure © 2003 Springer-Verlag Berlin Heidelberg, reprinted with permission

Components and predicates of LeadElect

Initialization by:

$mode = send$ **and** $cand =$ **self** **and** $mailbox = \emptyset$

$\textsc{Propose} =$ **forall** $q \in Neighb$ $\textsc{Send}(cand,$ **to** $q)$

$\textsc{ReceiveMsg} =$

choose $msg \in mailbox$

$curMsg := payload(msg)$ -- NB. msg payload is a process

$\textsc{Consume}(msg)$

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

$\textsc{UpdateKnowl} = (cand := curMsg)$ -- 'increase' of $cand$

NB. $mode, cand, mailbox, Neighb, curMsg$ are instantiated (read: implicitly parameterized) by the executing process **self**.

Termination property of GraphLeadElect

Correctness Lemma. In every properly initialized concurrent GraphLeadElect run with reliable communication, fair and not infinitely lazy components—i. e. every enabled process will eventually make a move and every $msg \in mailbox$ will eventually be chosen by ReceiveMsg—eventually for every $p \in Process$ holds:

- $cand = max(Process)$ w. r. t. $<$ (everybody 'knows' the leader).
- $mailbox = \emptyset$ (there is no more communication)
- $mode = check$

Proof. Follow in the given run R the propagation of $cand = Max$

- which holds in the initial state S_0 of R for $Max = max(Process)$

via Propose-steps to $Neighb$ors along paths through which any given p is reachable from Max.

Formally we proceed by induction on the minimal path-length n connecting Max to p.

$\textsc{GraphLeadElect}$ correctness proof

Decompose R into initial segments $InitSegm_0 = [S_0]$ and $InitSegm_n = [S_0, S_n]$ (of minimal length for $n > 0$) such that:

- for each $n > 0$ and each $p \neq Max$ that is reachable from Max by a path of minimal length n, before reaching S_n process p did:
  - have a msg with $payload(msg) = Max$ in its $mailbox$
  - choose a msg with payload Max to $\textsc{ReceiveMssg}$, $\textsc{UpdateKnowl}$ by Max and $\textsc{Propose}$ Max to its $Neighb$ors

Lemma. S_n is well-defined for each n and eventually $S_n = S_{n+1}$.

Proof. For $n = 1$ holds $p \in Neighb(Max)$ so that eventually Max will $\textsc{Propose}$ to p a msg with $payload(msg) = Max$ so that p eventually chooses msg to $\textsc{ReceiveMssg}$, to $\textsc{UpdateKnowl}$ by Max and to $\textsc{Propose}$ Max to its $Neighb$ors. For $n > 1$, for p (if there is some, othwise $S_n = S_{n+1}$) apply induction hypothesis to a q that is reachable from Max by a path of minimal length n with $p \in Neighb(q)$.

GraphLeadElect correctness proof (Cont'd)

Corollary 1. For all $n > 0$, Max and every p that is reachable from Max by a path of length $\leq n$:

- has $cand = Max$ in state S_n and maintains it in the rest of R
- Sends in R no msg neither in nor after state S_n
- in or in states after S_n is in $mode = check$ when its $mailbox = \emptyset$

Proof 1. Use that since Propose is unguarded, in R every $p \in Process$ always returns eventually to $mode = check$.

Corollary 2. For some k, the algorithm reaches S_k from where eventually it reaches a final state S of R in which every $p \in Process$ is in $mode = check$ and has an empty $mailbox$.

Proof 2. Follows from the finiteness of the graph so that for some $k > 0$ every $p \neq Max$ is reachabe from Max by a path of length $\leq k$.

On the meaning of ‘let every process know the leader’

Problem with the interpretation of *FunctionalReq*uirement by:

eventually every $\textsc{GraphLeadElect}$-run terminates in a state where for every $p \in Process$ $cand = Max$ holds.

- every $p \neq Max$ can recognize eventually that it is not the leader, namely when its $cand$ assumes a value $cand \neq p$
- no process knows when the run terminates, so no p (not even Max) can recognize when its $cand$ has the correct leader value $cand = Max$

A solution:

- we refine $\textsc{GraphLeadElect}$ to a ‘diffusing’ concurrent ASM $\textsc{GraphLeadElectDiffuse}$
- we define for every diffusing system $\mathcal{M}$ $\textsc{TerminationDetector}$ rules which permit to monitor without any central control every $\mathcal{M}$-run to recognize when this run terminates

ExI.2: Termination Detector for Diffusing Computations

A diffusing system (Dijkstra/Scholten 1980) is a communicating ASM for which each of its runs satisfies the following:

- the run starts in a state where all system components are *quiescent* (read: no step is enabled so that no local action can happen), say in $mode = idle$
- the run is started by the environment which enables exactly one system component by sending a message to this component to become the $master$ of the run—only once until the run terminates (if it terminates at all)
- every other system component can be enabled in the run only by receiving a message from some system component
- the run terminates if it reaches a state in which all components are again quiescent, say in $mode = idle$

NB. An ASM M is quiescent in a state if the disjunction of all rule guards of M is false in this state.

Termination detection idea for diffusing system runs

Let $\mathcal{M} = (m)_{m \in Machine}$ be a diffusing communicating ASM.

Idea: Since components are assumed to be enabled only by receiving some message (assuming reliable msg passing), namely

- either a 'monitor' msg, say $Start$, from the environment
  – sent by a $\textsc{BeginEndShell}(\mathcal{M})$ env program
- or an '$\mathcal{M}$-internal' msg from another component

it suffices to monitor acknowledgements of $\mathcal{M}$-internal msgs:

- require for every sent $\mathcal{M}$-internal msg an acknowledgement
- check, when a component becomes quiescent, that for every sent $\mathcal{M}$-internal msg an acknowledgement has been received

Then the $master$'s role is to monitor entering and exiting the first of the resulting spanning trees.

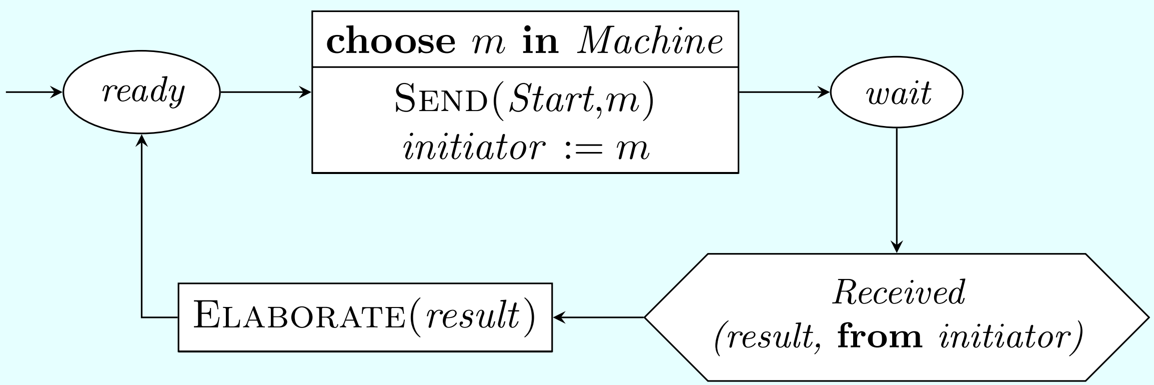
Environment program $\textsc{BeginEndShell}(\mathcal{M})$

Let $\mathcal{M} = (m)_{m \in Machine}$ be a diffusing communicating ASM. We extend it to a machine $\textsc{TerminationDetector}(\mathcal{M})$ by

- adding an env program $\textsc{BeginEndShell}(\mathcal{M})$ to trigger $\mathcal{M}$-runs
- extending each $pgm(m)$ to $pgm(m)^*$ by components
  - which react to an env trigger and monitor send/receive actions

Define $\textsc{BeginEndShell}(\mathcal{M})$ as the following control state ASM:

- located at a distinguished graph node without incoming edges[2]



[2] Figure © 2018 Springer-Verlag Berlin Heidelberg, reprinted with permission

Master Start/Termination rules

$\textsc{Start}_{\mathit{Diffuse}} =$

$\quad$ **if** $status(\textbf{self}) = idle$ **and** $Received(Start, \textbf{from } env)$ **then**

$\quad\quad status(\textbf{self}) := master$

$\quad\quad \textsc{Start}(\textbf{self})$ -- assumed to enable **self**

$\quad\quad \textsc{Consume}(Start, \textbf{from } env)$

$\textsc{Terminate}_{\mathit{Diffuse}} =$

$\quad$ **if** $status(\textbf{self}) = master$ **and** $quiescent(\textbf{self})$

$\quad\quad$ **and** $AllAcksArrived(\textbf{self})$ **then**

$\quad\quad\quad status(\textbf{self}) := idle$

$\quad\quad\quad \textsc{Send}(computationResult, env)$ -- payload of termination msg

NB. Initially each component is assumed to be in $status = idle$, with empty $mailbox$ and without msgs in the communication medium.

Spanning tree requirements

- *SpanningTreeBuildReq*. When a non-master component becomes enabled, namely by receiving in $status = idle$ an $\mathcal{M}$-internal (a non-ack) message, it designates the message sender—a neighbor—as its parent in a new spanning tree ($\textsc{EnterSpanningTree}$). Any further received $\mathcal{M}$-internal message is immediately acknowledged (also by the master).
- *ConvergeCastReq*. When a non-master component becomes quiescent and has received an ack for every $\mathcal{M}$-internal message it had sent out, it will $\textsc{ExitSpanningTree}$ and send an ack to its parent (thus acknowledging the msg which triggered building this spanning tree).
- *TerminationReportReq*. When the master becomes quiescent and all the $\mathcal{M}$-internal messages it had sent—some of which enabled other components—have been acknowledged, then it will report to the environment that the computation did terminate.

NB. Components may be enabled & disabled multiple times during a run.

Rules to monitor sending $\mathcal{M}$-internal messages

Each m has for each other m' a counter $msgToBeAckBy(m')$ of the number of $\mathcal{M}$-internal msgs sent by m to m' but not yet acknowledged by m'.

- $msgToBeAckBy(m')$ is increased at each $\textsc{Send}(msg, \textbf{to } m')$

  $\textsc{MonitorSentMsg}(msg, \textbf{to } receiver) =$ -- for $\mathcal{M}$-internal msg
  $\quad \textsc{Increase}(msgToBeAckBy(receiver))$
- $msgToBeAckBy(m')$ is decreased when $Received(ack, \textbf{from } m')$

  $\textsc{MonitorAckMsg} =$ -- decrease counter of expected acks
  $\quad$ **if** $Received(ack, \textbf{from } receiver)$ **then**
  $\qquad \textsc{Decrease}(msgToBeAckBy(receiver))$
  $\qquad \textsc{Consume}(ack, \textbf{from } receiver)$

Spanning tree rules to monitor msg acknowledgement

$\textsc{MonitorReceivedMsg}(msg,$ **from** $sender) =$ -- $\mathcal{M}$-internal msg

if $status($**self**$) = idle$

then $\textsc{EnterSpanningTree}(sender)$ -- get enabled

else $\textsc{Send}(ack, sender)$ -- immediate ack if already woken up

$\textsc{Consume}(msg,$ **from** $sender)$

where $\textsc{EnterSpanningTree}(sender) =$ -- NB. no ack sent

$status($**self**$) := treeNode$ $\quad parent($**self**$) := sender$

$\textsc{ExitSpanningTree} =$ **if** $status($**self**$) = treeNode$ **and**

$quiescent($**self**$)$ **and** $AllAcksArrived($**self**$)$ **then**

$\textsc{Send}(ack, parent($**self**$))$ -- Ack msg which created spanning tree

$parent($**self**$) :=$ **undef** $\quad status($**self**$) := idle$

where $AllAcksArrived($**self**$)$ **iff**

forall $m \in Machine\ msgToBeAckBy_{\mathbf{self}}(m) = 0$

$\textsc{TerminationDetector}(\mathcal{M})$ **for diffusing** $\mathcal{M} = (m)_{m \in Machine}$

$$\textsc{BeginEndShell}(\mathcal{M})$$
$$(pgm(m)^*)_{m \in Machine}$$

where $pgm(m)^*$ is obtained by adding to $pgm(m)$ the following rules:

$\textsc{Start}_{Diffuse}$ -- just once if triggered by the environment

$\textsc{MonitorSentMsg}(msg, \textbf{to}\ receiver)$ -- record expected acks

in parallel to any occurrence of $\textsc{Send}(msg, \textbf{to}\ receiver)$ in $pgm(m)$

$\textsc{MonitorAckMsg}$ -- when ack arrives decrease expected acks

$\textsc{MonitorReceivedMsg}(msg, \textbf{from}\ sender)$ -- wake up or do ack

in parallel to each occurrence of $\textsc{Receive}(msg, \textbf{from}\ sender)$

or action triggered by $Received(msg, \textbf{from}\ sender)$ in $pgm(m)$

$\textsc{ExitSpanningTree}$ -- only quiescent tree nodes if $AllAcksArrived$

$\textsc{Terminate}_{Diffuse}$ -- only master if quiescent & $AllAcksArrived$

$\textsc{TerminationDetector}(\mathcal{M})$ **Lemma**

In every $\textsc{TerminationDetector}(\mathcal{M})$ run every state satisfies:

- For every machine $m \neq initiator$: $status = idle$ iff $parent = \mathbf{undef}$ and in that case (read: if m is not a node of the spanning tree) m has not to wait for any message to be acknowledged (formally expressed $msgToBeAckBy(m')_m = 0$ for each $m' \in Machine$).
- The machines linked by a $parent$ path to the root $initiator$ form a spanning tree of all machines with $status \neq idle$.

$==================================$

The initialization is defined s.t. for every $m \in Machine$:

- m quiescent in $status(m) = idle$ with empty $mailbox(m)$
- no sent but not yet received message in the communication medium and no to-be-acknowledged msg
  – i. e. for each $m' \in Machine$ holds $msgToBeAckBy(m')_m = 0$)
- no parent defined ($parent(m) = \mathbf{undef}$)
- initial mode is $ready$

Applying $\textsc{TerminationDetector}$ to $\textsc{GraphLeadElect}$

To make $\textsc{GraphLeadElect}$ diffusing, every p which—by having received a $Start$ msg (initially from the environment)—is enabled, namely by updating $mode(p) = terminated$ to $mode = send$, will also $\textsc{StartNeighb}$ors and $\textsc{PropagateStart}$

- so that eventually the leader Max enters $mode = send$.

Therefore $Start$ is treated as $\textsc{GraphLeadElect}$-internal msg so that all $Neighb$ors of p $\textsc{MonitorReceivedMsg}(Start, \textbf{from } p)$.

- in $\textsc{Start}_{Diffuse}$ refine $\textsc{Start}_{\textbf{self}}$ to:

  $mode := send$ -- enabling **self**

  $\textsc{StartNeighb}$ -- defined as **forall** $q \in Neighb$ $\textsc{Send}(Start, \textbf{to } q)$

- to $\textsc{LeadElect}$ add $\textsc{PropagateStart}$, defined by

  if $Received(Start, \textbf{from } p)$ **and** $p \in Process$ **then**

      if $mode = terminated$ **then** $\textsc{Start}(\textbf{self})$ -- enable **self**

      $\textsc{Consume}(Start, \textbf{from } p)$

Refinement to GraphLeadElectDiffuse

Refine $\textsc{Terminate}_{\mathit{Diffuse}}$ to trigger a final round which resets each component to $\mathit{mode} = \mathit{terminated}$ (for the next diffusing run).

- $\textsc{Terminate}_{\mathit{Diffuse}} =$

  if $\mathit{status} = \mathit{master}$ **and** $\mathit{quiescent}$ **and** $\mathit{AllAcksArrived}$ **then**

      if $\mathit{mode} = \mathit{check}$ **then** $\textsc{Terminate}$ -- launch termination round

      if $\mathit{mode} = \mathit{terminated}$ **then**

          $\mathit{status} := \mathit{idle}$ $\quad$ $\textsc{Send}(\mathit{computationResult}, \textbf{to } \mathit{env})$

  where $\textsc{Terminate} =$

      $\textsc{TerminateNeighb}$ -- **forall** $q \in \mathit{Neighb}$ $\textsc{Send}(\mathit{Stop}, \textbf{to } q)$

      $\mathit{mode} := \mathit{terminated}$ -- initialize mode for next diffusing run

- Stop (like Start) is treated as GraphLeadElect-internal msg.
- To LeadElect add PropagateTermination, defined by:

  if $\mathit{Received}(\mathit{Stop}, \textbf{from } p)$ **then**

      if $\mathit{mode} \neq \mathit{terminated}$ **then** $\textsc{Terminate}$

      $\textsc{Consume}(\mathit{Stop}, \textbf{from } p)$

Termination detection lemma for $\textsc{GraphLeadElectDiffuse}$

Let $T = \textsc{TerminationDetector}(\textsc{GraphLeadElectDiffuse})$.
Every diffusing run of T eventually terminates and does $\textsc{Elaborate}(Max)$ (refining $computationResult$ to $cand$).

More precisely: Let R be any diffusing run of T (with reliable message passing and without infinitely lazy components).

R eventually terminates and does $\textsc{Elaborate}(Max)$ when the system of $\textsc{GraphLeadElectDiffuse}$ machines, started by an $initiator$ in a quiescent state, after the second $\textsc{Terminate}_{Diffuse}$ step of the $initiator$ enters again a quiescent state, where all components have $status = idle$ and $mode = terminated$.

Proof. Follows from the termination of $\textsc{GraphLeadElect}$ runs and from the above explained behavior of the monitoring components of $\textsc{TerminationDetector}$. For details see ModelingBook pg.125.

Exl.3: Concurrent emulation of synchronous processes

Goal. Emulation of synchronous runs of a network $(Process, Edge)$ of communicating processes by concurrent runs of a network

$$\textsc{ConcurSyncEmulator}(Process, Edge) = (Process^*, Edge^*)$$

(also called $\textsc{LocalSyncTransformer}$) of communicating processes, assuming immediate reliable communication bw neighbors.

$=========================$

A *round* in a synchronous run is characterized by each $p \in Process$

- reading its mailbox only at the beginning of the round
- sending messages only at the end of the round, msgs which in the next round are in the receivers' mailbox
- performing otherwise only non-communication local actions

Wlog we abstract from the possible sequence of local actions, treating them as one atomic step, and assume that in each round, each process sends exactly one (possibly empty) message to each of its neighbors.

Characterization of synchronous $Process$ runs

Let $(Process, Edge)$ be a network of communicating ASMs p with agent $ag(p)$ executing $pgm(p)$ using a $mailbox(p)$ for immediate reliable communication with the $Neighb$ors of p (defined via $Edge$).

The synchronous runs of $(Process, Edge)$ can be described as the runs of the following (highly parallel) ASM. In each step it performs one step (communication and local actions) of each of its components.

$\textsc{SyncNet}(Process, Edge) =$

forall $p \in Process$

$pgm(p)$

$\textsc{IncreaseRound}$ -- i. e. $curRound := curRound + 1$

Each round corresponds to one step of $\textsc{SyncNet}(Process, Edge)$ so that $curRound$ works as step counter.

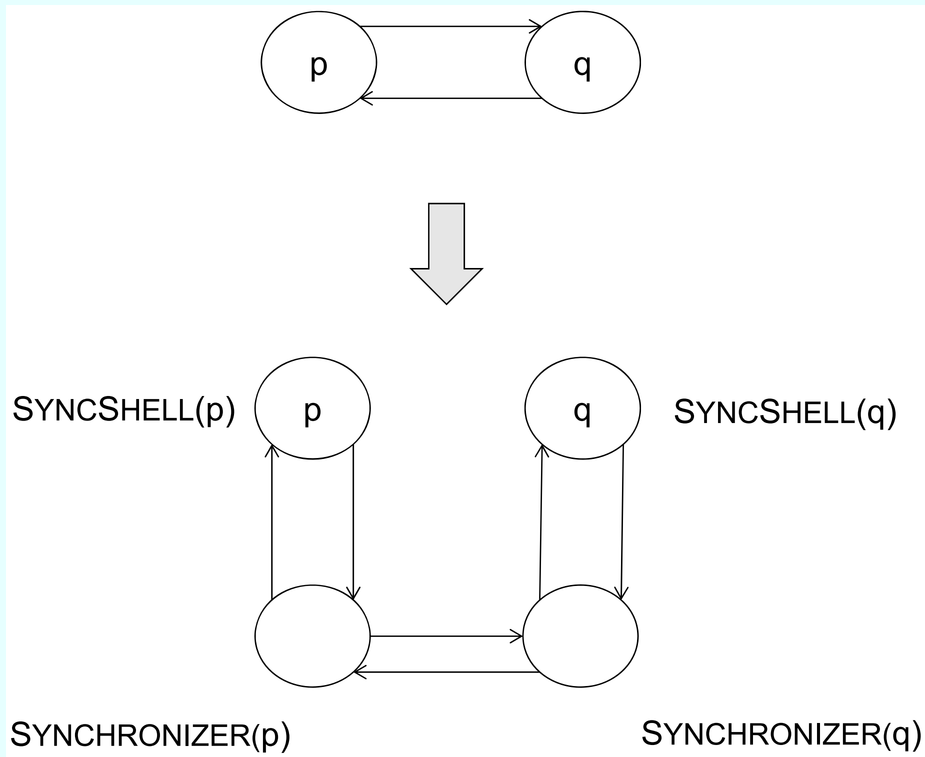
Concurrent emulation of synchronous processes

Idea (Lynch 1996, Sect.16.2, here generalized from interleaved processes to concurrent communicating ASMs):

- associate each process $p \in Process$ with a $synchronizer(p)$ which for each round r synchronizes
  - each round-r-step of p with a round-r-step of all its $Neighb$ors
  - the $Process$-internal communication in round r, i.e. between p and its $Neighb$ors (via $Edge$)
- replace $pgm(p)$ by $\textsc{SyncShell}(p)$ which
  - simulates one step of $pgm(p)$ when $ReadyForNextRound_p$
  - $\textsc{Suspends}$ p until it has $ReceivedAllMsgsFor$ round $r+1$ and the $synchronizer(p)$ sends a $resume$ msg (after p and all its $Neigb$ors $MadeOneStep$ in round r)

Graph $Edge^*$ of the concurrent synchronization emulator

$\rightarrow$ indicates a communication line between neighbors.[3]



NB. $pgm(p)$ is changed to $\mathrm{SyncShell}(p)$ whose round-r-steps are synchronized, by an agent with pgm $\mathrm{Synchronizer}(p)$, with those of (the SyncShell instances of) all $Neigh$bors of p (in $Edge$).

[3] Figure © 2003 Springer-Verlag Berlin Heidelberg, reprinted with permission

$\mathit{Process}^*$ of the concurrent synchronization emulator

For each process p

- its $pgm(p)$ is replaced by a program $\textsc{SyncShell}(p)$ which refines the communication actions of p
- a synchronization controller $\textsc{SyncCtl}(p)$ is added which synchronizes the round actions of all processes (including their communication)

$\mathit{Process}^* =$

$$\mathit{Process}(pgm(p)/\textsc{SyncShell}(p)) \qquad \text{-- process refinement}$$

$$\cup\ \textsc{SyncCtl} \qquad \text{-- adding synchronization controller}$$

where

$$\textsc{SyncCtl} = (\textsc{SyncCtl}(p))_{p \in \mathit{Process}}$$

$$\textsc{SyncCtl}(p) =$$

$$(\mathit{synchronizer}(p), \textsc{Synchronizer}(p), \mathit{mailbox}(\mathit{synchronizer}(p)))$$

$\textsc{SyncShell}(p)$ for the concurrent synchronization emulator

if $ReadyForNextRound_p$ **then**

$pgm(p)^*$ -- simulate one step of $pgm(p)$

$\textsc{Suspend}(p)$

else

if $Received(resume,$ **from** $synchronizer(p))$ **then**

$\textsc{Resume}(p)$

where

$ReadyForNextRound_p$ **iff**

$Resumed(p)$ **and** $ReceivedAllMsgsFor(curRound_p, p)$

$\textsc{Suspend}$ and $\textsc{Resume}$ actions

$\textsc{Suspend}(p)$ sets $WaitingForNextRoundTick(p)$ to true, informs the $synchronizer(p)$ that p has performed its $curRound_p$ step and prepares itself for the next round (by an $\textsc{Increase}(curRound_p)$).

$\textsc{Suspend}(p) =$ -- making p not $ReadyForNextRound$

  $WaitingForNextRoundTick(p) := true$ -- reset by $\textsc{Resume}$

  $\textsc{InformAboutStep}(p)$ -- step performed by $pgm(p)^*$

  $\textsc{Increase}(curRound_p)$ -- preparing for next round $curRound_p + 1$

$\textsc{Resume}(p) =$ -- upon receiving $resume$ msg

  $WaitingForNextRoundTick(p) := false$

  $\textsc{Consume}((resume,$ **from** $synchronizer(p))$

where

  $\textsc{InformAboutStep}(p) =$

    $\textsc{Send}(stepInfo(p, curRound_p),$ **to** $synchronizer(p))$

  $Resumed(p)$ **iff** $WaitingForNextRoundTick(p) = false$

Simulation of $pgm(p)$-steps by $pgm(p)^*$-steps

Simulating a $pgm(p)$-step consists in performing this step except for $\textsc{Send}$ing $Process$-internal msgs (analogously for receiving)

- together with the $curRound_p$ information
- not to their receiver $q \in Neighb(p)$ but to the $synchronizer(p)$

where

- $\textsc{Synchronizer}(p)$ does $\textsc{ForwardMsgsSentBy}(p, curRound_p)$ to $synchronizer(q)$
- $\textsc{Synchronizer}(q)$ will $\textsc{PassMsgsSentTo}(q, r)$ to q

$pgm(p)^* = pgm(p)$ replacing

$\textsc{Send}(m, \textbf{to } q)$ -- communication with round info via $synchronizer$

by $\textsc{Send}((m, curRound_p, \textbf{to } q), \textbf{to } synchronizer(p))$

$Received(m, \textbf{from } q)$ by

$Received((m, curRound_p - 1, \textbf{from } q), \textbf{from } synchronizer(p))$

-- NB. m received in round r has been sent in round $r - 1$

The two SYNCHRONIZER roles

- FORWARDMSGSSENTBY process p in a round-r-step, namely to the $synchronizer(q)$ of each $Neighb$or q of p. These msgs can be:
  – monitor msgs: $stepInfo$ and $resume$ msg
  – a $Process$-internal $mgs \in ProcessMsg$, exchanged between processes via a SEND(msg) in some $pgm(p)$ with $p \in Process$
- Check when to CLOSEROUND r for p by
  – passing to p in one blow all $ProcessMsg$s which have been sent in round r to p—$Received$ by $synchronizer(p)$ from $synchronizer(q)$ of some $q \in Neighb(p)$—to be $Received$ by p in round $r+1$
  – waking up p (by a $resume$ msg) and proceeding to the next round

SYNCHRONIZER$(p) =$

let $r = curRound($**self**$)$

FORWARDMSGSSENTBY(p, r)

CLOSEROUND(r, p)

$\textsc{CloseRound}(r, p)$ **rule of** $\textsc{Synchronizer}(p)$

if $\mathit{Received}(\mathit{stepInfo}(p, r),$ **from** $p)$ -- p made a round-r-step

and (**forall** $q \in \mathit{Neighb}(p)$ $\mathit{MadeOneStep}(q, r))$

and $\mathit{ReceivedAllMsgsToPassTo}(p, r)$ **then**

$\textsc{PassMsgsSentTo}(p, r)$ -- msgs to-be-received in round $r + 1$

$\textsc{WakeUp}(p)$ -- i.e. $\textsc{Send}(\mathit{resume},$ **to** $\mathit{syncShell}(p))$

where

$\mathit{MadeOneStep}(q, r)$ **iff** -- NB. $r = \mathit{curRound}_q$

$\mathit{Received}(\mathit{stepInfo}(q, r),$ **from** $\mathit{synchronizer}(q))$

$\mathit{ReceivedAllMsgsToPassTo}(p, r)$ **iff**

forall $q \in \mathit{Neighb}(p)$ -- q has sent a msg to p in round r

forsome m $\mathit{Received}((m, r,$ **to** $p),$ **from** $\mathit{synchronizer}(q))$

NB. p assumed to send per round to each neighbor exactly one msg.

Components of $\textsc{CloseRound}(r, p)$

$\textsc{PassMsgsSentTo}(p, r) =$ -- all msgs p receives in round $r+1$

forall $q \in Neighb(p)$ **forall** -- all msgs of type $ProcessMsg$

$((m, r, \textbf{to } p), \textbf{from } synchronizer(q)) \in mailbox \cap ProcessMsg$

$\textsc{Send}((m, r, \textbf{from } q), \textbf{to } p)$

$\textsc{Consume}(m, r, \textbf{from } q)$

This rule is executed when $ReceivedAllMsgsToPassTo(p, r)$ is true. Correspondingly we can define for $ReadyForNextRound_p$:

$ReceivedAllMsgsFor(r+1, p)$ **iff**

forall $q \in Neighb(p)$ **forsome** m -- q has sent some msg in round r

$Received((m, r, \textbf{from } q), \textbf{from } synchronizer(p))$

$ReceivedAllMsgsFor(0, p) = true$ -- by initialization

-- If $Received(m)$ in round $r+1$, m has been sent in round r

Initial states & runs of $\textsc{ConcurSyncEmulator}(\mathit{Process}, \mathit{Edge})$

Initial states satisfy:

- $\mathit{curRound} = 0$ and $\mathit{mailbox} = \emptyset$ for each $p \in \mathit{Process}$ (same for $\textsc{SyncNet}(\mathit{Process}, \mathit{Edge})$) and for each $\textsc{SyncCtl}(p)$
- $\mathit{ReadyForNextRound}_p = \mathit{ReceivedAllMsgsToPassTo}(p) = \mathit{true}$ and $\mathit{ReceivedAllMsgsFor}(0, p) = \mathit{true}$ for each $p \in \mathit{Process}$.

The concurrent emulation refines each p-step of one $\textsc{SyncNet}(\mathit{Process}, \mathit{Edge})$ round-r-step into:

- a first $\textsc{SyncShell}(p)$ step—when $\mathit{ReadyForNextRound}_p$ holds for r
- followed by message forwarding steps performed by $\mathit{synchronizer}(p)$ and the $\mathit{synchronizer}$s of its neighbors q
- followed by a $\textsc{CloseRound}(r, p)$ step—when also all neighbors of p made a round-r-step and the messages sent in round r to p have been received by the $\mathit{synchronizer}(p)$
- concluded by the second $\textsc{SyncShell}(p)$ round-r-step

$\textsc{ConcurSyncEmulator}(\mathit{Process}, \mathit{Edge})$ **correctness property**

Let R_s be any run of $\textsc{SyncNet}(\mathit{Process}, \mathit{Edge})$ and R_c any concurrent $\textsc{ConcurSyncEmulator}(\mathit{Process}, \mathit{Edge})$ run, both properly initialized and started with equal values in same-named locations.

Then for every process p the following holds for every round number $r = \mathit{curRound}_{\textsc{SyncNet}(\mathit{Process}, \mathit{Edge})} = \mathit{curRound}_p$:

- when $ag(p)$ starts its round r in R_s resp. in R_c, the runs are in corresponding states $\mathit{state}_{R_s}(r), \mathit{state}_{R_c}(p, r)$ with same values
  - in same-named p-locations
  - for the payload and destination of corresponding $\mathit{ProcessMsg}$s
    - $(m,$ **to** $q)$ resp. $(m, \mathit{curRound}_p,$ **to** $q)$

    sent by $ag(p)$ in round r to (similarly $\mathit{Received}$ from) its neighbor q.

NB. Same-value property in corresponding states means

$$\mathit{state}_{R_s}(r) \downarrow \Sigma_p = \mathit{state}_{R_c}(p, r) \downarrow \Sigma_p$$

Proof: induction on r (see ModelingBook pg. 131 for details).

References

- E. Börger and A. Raschke: Modeling Companion for Software Practitioners. Springer 2018
  http://modelingbook.informatik.uni-ulm.de
  The book bibliography provides exact references to the literature from where the three examples are taken.
- E. Börger and R. Stärk: Abstract State Machines. Springer 2003.

Copyright Notice