Egon Börger (Pisa)

Design and analysis of distributed algorithms

with Abstract State Machines

A comparison with Petri nets

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

Invited Lecture at ABZ'2016. See Springer LNCS 9675 (pg. 3-34)[1]

[1] The ASM figures below are taken from there, © 2016 Springer-Verlag, reused with permission

Theme of this lecture

Compare *technically*, sine ira et studio, the use of Petri nets (PNs) and of Abstract State Machines (ASMs) for

- *modeling* distributed algorithms
- *analysing* properties of distributed algorithms

using *concrete characteristic examples*.

This endeavour turned out to be an exercise in building appropriate

communicating ASM ground models

for requirements which are formulated in natural language.

Procedure for concrete comparison of PNs and ASMs

For evaluation fairness we base the comparison on:

- W. Reisig, Elements of Distributed Algorithms (1998)[2] for
  - *notion of Petri net*, defined there to "provide the expressive power necessary to model elementary distributed algorithms adequately, retaining intuitive clarity and formal simplicity" (p.VII)
  - *"choice of* small and medium size distributed *algorithms"* proposed as representative "for a wide class of distributed algorithms" which "can help the practitioner to design distributed algorithms" (p.V)
  - we select however the *'Advanced System Models'* from op.cit. (instead of the elementary ones) to make the comparison more meaningful
- Nancy A. Lynch, Distributed Algorithms (1996) for requirements (where not taken directly from Reisig's book)

[2] The PN figures are taken from there, © 1998 Springer-Verlag Berlin Heidelberg, reused with permission

The declared goal of modeling and verification

"*help the practitioner* to design distributed algorithms" (p.V)

"*retaining intuitive clarity and formal simplicity*" (p.VII) resp.

"*make intuitive statements and conclusions transparent and precise*, this way *deepening the reader's insight* into the functioning of systems" (p.143)

Therefore, when below we compare PN/ASM-based proofs it is

- NOT about machine supported (automatic or interactive) mechanical model checking or theorem proving
  - such additional value requires additional effort!
- but about traditional mathematical proofs deviced for human readers
  - to support the practitioner's intuitions and his understanding that and why an algorithm does what the requirements ask it to do

Assuming abstract communication medium

The algorithms describe communicating $Processes$, each equipped with its $mailbox_{\textbf{self}}$ and the following actions/predicates:

$\textsc{Send}(msg, \textbf{to}\ p)$ -- *deliver* msg *to mailbox* of p

-- triggers communication medium to $\textsc{Insert}(msg, mailbox_p)$

$Received(msg) = (msg \in mailbox)$ -- *msg has been delivered*

$\textsc{Consume}(msg) = \textsc{Delete}(msg, mailbox)$

Abstract $\textsc{Insert}$ action at the receiver process happens

- immediately in the synchronous (reliable) model
- eventually in the asynchronous (reliable) model

This abstracts from introducing explicit communication channels

- $Chan_{p,q}$ linking $outMailbox(p)$ to $Chan$ input and $Chan$ output to $inMailbox(q)$

Flooding leader election over connected graphs (Lynch 4.1)

Requirement: Design and verify an algorithm such that

- over a finite *connected directed graph* $(Process, Edge)$
- using *communication only bw Neighb*ors

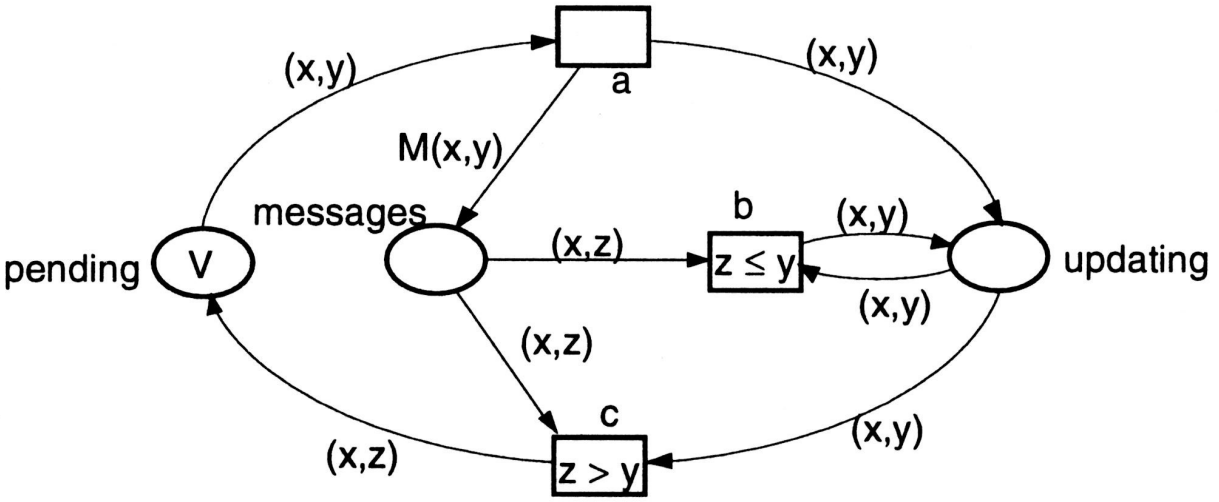
eventually every process knows the leader $max(Process)$ (wrt linear order $<$ of $Process$es) and the algorithm terminates.

Algorithmic idea: repeatedly each process p *alternates*

- to first PROPOSE its current leader knowledge $cand$ (greatest process seen so far, initially $cand_p = p$) SENDing it to its $Neighb$ors
- then to CHECK the $Proposals$ ($Received$ in the initially empty $mailbox = Proposals$) whether they IMPROVE *its* $cand$ value

until everybody knows the leader.

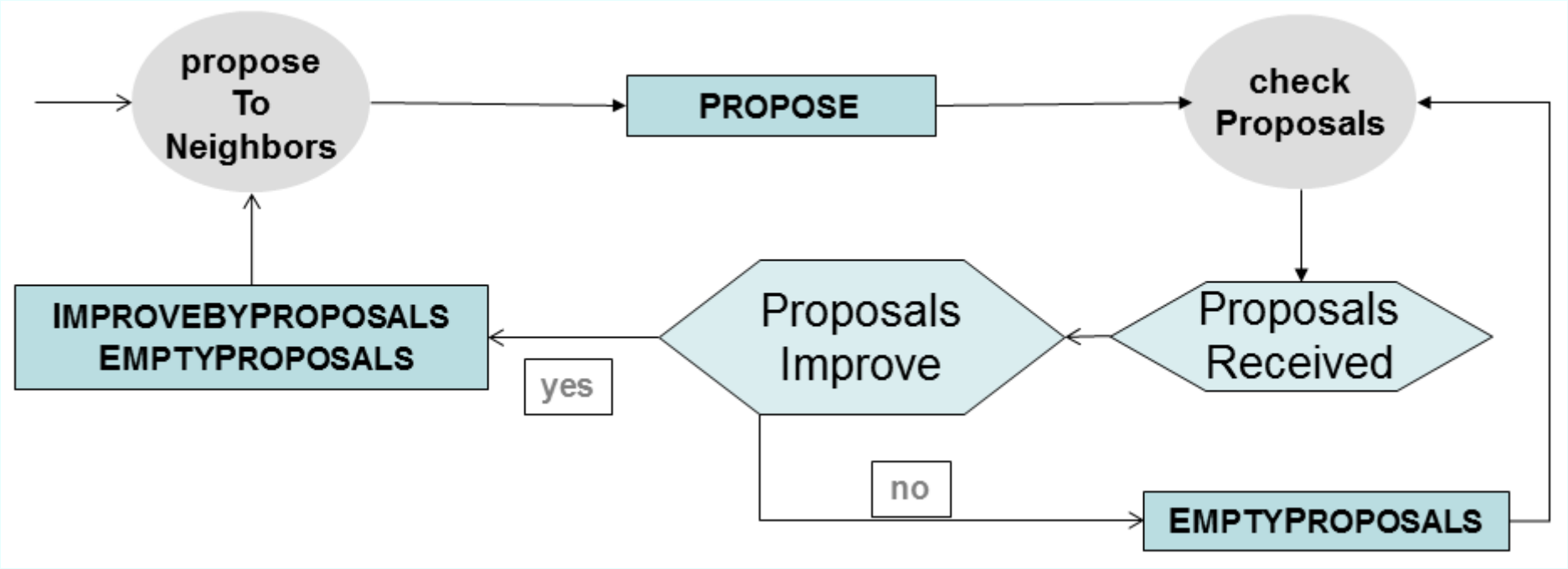
Petri net encoding of FloodingLeadElect



sort	site	var	x, y, z : site
sort	state : site × site		$x,y \in U \rightarrow x\, W^*\, y$
const	U : set of sites		$W_1 \cup W_2 = U$
const	V,W : set of states		$V = \{(u,u) \mid u \in U\}$
	≤ : total order on U		$M(x,y) = W(x) \times \{y\}$
fct	M : state → set of states		

Source: W. Reisig, Elements of Distributed Algorithms. Fig.32.1

From English to ASM: 2-phase FloodingLeadElect



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

$ProposalsImprove = (max(Proposals) > cand)$

$\textsc{ImproveByProposals} = (cand := max(Proposals))$

Each family of $(p, \textsc{FloodingLeadElect})$ forms a concurrent ASM.

Comparison reveals Petri net idiosyncrasy (1)

low level token-based encoding of objects/agents & of actions as token manipulations buries intuitive meaning of single agents' actions

- PN implementation of **forall** $q \in Neighb$ $\textsc{Send}(cand, \textbf{to}\ q)$
  - deletes any token (x, y) (read: a process x with leader candidate y) from place $pending$
  - adds the set $M(x, y) = W(x) \times \{y\}$ of tokens to place $messages$
    - $W(x)$ encodes the logical expression '**forall** $q \in Neighb$': instead of communication medium forwarding candidate msg y into neighbors' local $mailbox$es it is x which moves around all its neighbors (coupled with y) to place $messages$ (a global mailbox)!
  - adds token (x, y) to place $updating$
    - passing token (x, y) from $pending$ to $updating$ encodes $mode$ update for x from $send$ ($proposeToNeighbors$) to $receive$ ($checkProposals$): why should x move & y be dragged along?
- *initialization* $cand =$ **self** encoded by token set V in place $pending$

Comparison reveals Petri net idiosyncrasy (2)

global overall process view

- *obstructs separation of concerns*: in one PN initialization and actions of each single process instance are detailed explicitly
  - moving x around (for $\leq$) where only cand/msg y/z needed
  - instead of describing them locally, separately for parameterized single processes Propose, CheckProposals, ImproveByProposals to keep models small (easy to understand and analyse)
- *obfuscates architectural system view* (communication structure)
- *burdens net layout with background elements*
  - instead of separating background and control concerns, here the graph $(Process, Edge)$ with $<$ and $Neighb$orhood relation

Consequence: lack of support for componentwise and stepwise refinable design/analysis of concurrent processes with multiple component types: *complicates implementation*

Comparison reveals Petri net idiosyncrasy (3)

- visualization helpful mostly for control flow, resorting to *less clear encoding of data* flow in PN:
  - alternating bw $pending$ and $updating$ explicitly visualized
  - only 'indirect' (implicit) visibility of checking $Proposals$
    - PN describes it elementwise : a low-level implementation!

    compared to the (via max function) conceptually high-level integration of checking $Proposals$ into the ASM flowchart

Nevertheless, PN diagrams are considered as defining the algorithm:

- "The hurried reader may just study the pictures." (Reisig p.V)
- "Petri nets are a graphical notion and at the same time a precise mathematical notion". This is generally considered among "the most important properties" of PNs (Desel et al. 2001)

But *large unstructured diagrams*, without appropriate subdiagram concept, tend to become *difficult to understand and analyse*.

Verifying correctness of FLOODINGLEADELECT

- **Proposition**: In every concurrent run of $Process$es, each equipped with program FLOODINGLEADELECT, if every enabled process will eventually make a move, eventually for every $p \in Process$ holds:
  - $cand = max(Process)$ (everybody knows the leader)
  - mailbox $Proposals = \emptyset$ (no more communication)
  - $mode = checkProposals$ (quiet mode)
- **Proof**: induction on runs and on the sum of the differences $diff(max(Process), cand)$ until this sum becomes 0

Main step: each time a process p (e.g. $max(Process)$) PROPOSEs itself *to a smaller neighbor* $q \in Neighb_p$, next time that neighbor checks it discovers that its $ProposalsImprove$ and *increase* $cand_q$ (wrt $<$) yielding a decrease of $diff(max(Process), cand_q)$.

Compare with technically involved (formalistic, hard to follow) 'proof graph' based Petri net verification in Reisig pg. 258-260.

Master/Slave agreement protocol requirements

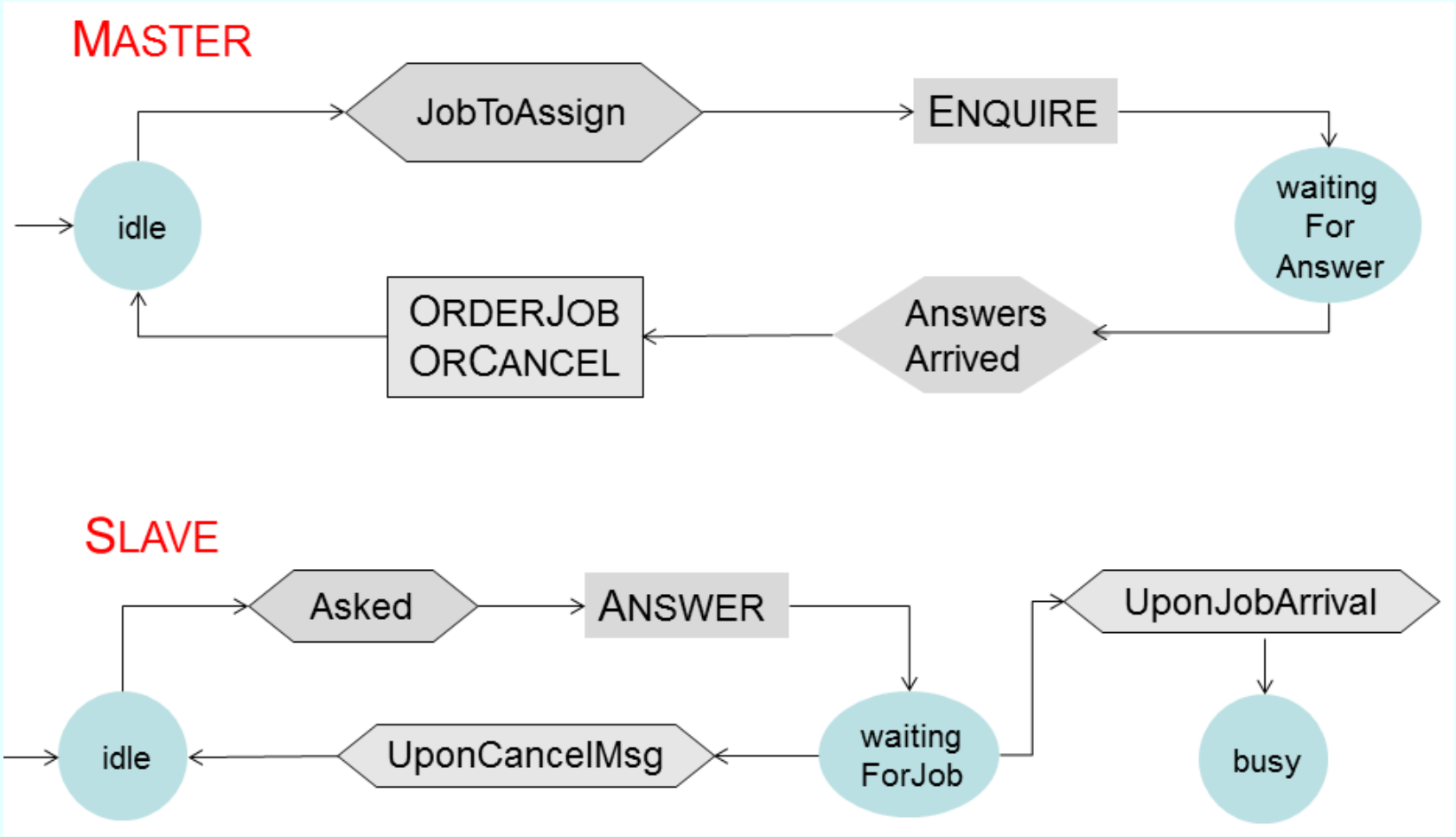
Requirement: Algorithm for a master having a $JobToAssign$ to slave agents, job that will be executed by all slaves only if all of them have confirmed to the master to accept to execute and will be canceled if some of them refuses execution. (op.cit. Sect.30)

Algorithmic Idea:

- the *master* (when it has a $JobToAssign$) starts by sending an inquiry to each slave (ENQUIREs) and then waits for the answers
  - *each slave* (when $Asked$) ANSWERs to accept or refuse
- if all $AnswersArrived$ the *master* sends to each slave a msg to
  - ORDERJOB in case all slaves accepted
  - ORCANCEL the request (otherwise)
    - *slaves change mode* $UponJobArrival$ or $UponCancelMsg$

Goal: eventually the master becomes idle, with either all slaves idle too or all slaves busy (executing the accepted job).

From English to ASM: MASTER/SLAVE programs



From English to ASM: Master/Slave states (signature)

- $master$: a distinguished agent with locations:
  - $job \in Job$ for the job to assign to the slaves
  - $JobToAssign$, a trigger predicate signalling that the $master$ should start trying to assign the job
  - $mode \in \{idle, waitingForAnswer\}$, initially $mode = idle$
  - program $\textsc{Master}$
- $Slaves$: a set of agents with locations:
  - $mode \in \{idle, waitingForJob, busy\}$, initially $mode = idle$
  - program $\textsc{Slave}$

From English to ASM: Master predicates and actions

$\textsc{Enquire} =$ **forall** $s \in Slave$ $\textsc{Send}(enquire,$ **to** $s)$

$AnswersArrived =$ **forall** $s \in Slave$

  $Received((accept, s))$ **or** $Received((refuse, s))$

$\textsc{OrderJobOrCancel} =$

  if $SomeSlaveRefused$ **then forall** $s \in Slave$ $\textsc{Send}(cancel,$ **to** $s)$

    else forall $s \in Slave$ $\textsc{Send}(job,$ **to** $s)$

  $\textsc{CleanUp}$ -- clean up work for next round

$SomeSlaveRefused =$**forsome** $s \in Slave$ $Received((refuse, s))$

$\textsc{CleanUp} =$

  $mailbox := \emptyset$

  $JobToAssign := false$ -- consume input trigger

From English to ASM: Slave predicates and actions

$Asked = Received(enquire)$

$\textsc{Answer} =$

  choose $answer \in \{accept, refuse\}$

    $\textsc{Send}((answer, s), \textbf{to}\ master)$

  $\textsc{Consume}(enquire)$ -- consume input msg

$UponCancelMsg = Received(cancel)$

$UponJobArrival = Received(job)$

Master/Slave correctness proof

Proposition. In every concurrent run of a set of master and slaves, all equipped with the corresponding MASTER, SLAVE algorithm,

- if the master starts to ENQUIRE and every enabled agent will eventually make a move,
- then eventually the master becomes idle and
  - either all slaves become idle too
  - or all slaves become busy (executing the job sent by the master).

Proof: by run induction investigating the agents' phases.

Petri net encoding of Master/Slave agreement

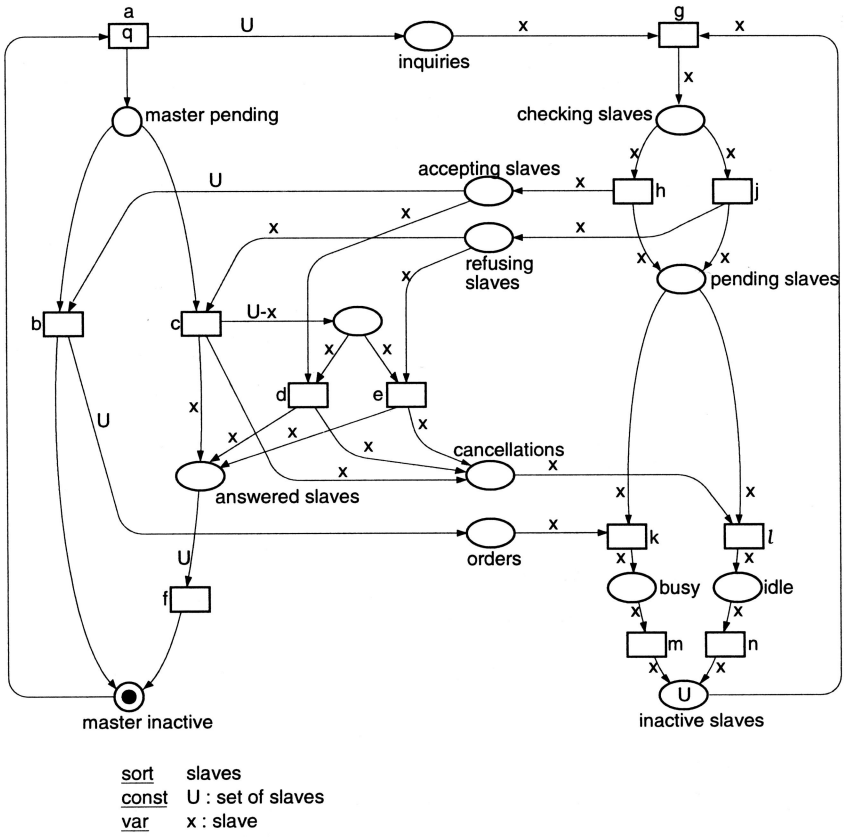


Figure 30.1. Distributed master/slave agreement

Petri net idiosyncrasy: *complex graphical layout* (even for small systems), *extraneous to algorithmic structure*, complicates understanding

Global PN view complicates order cancellation

Global overall process model of Fig.30.1 lets the *slaves organize the order cancellation among themselves*, without further master/slave communication

- in case one slave x refuses, it triggers (by transition c)
  - the master to change mode (inverting the master/slave relation!)
    - walk from place $master\ pending$ to place $answered\ slaves$ to prepare its return to mode $idle$
  - the other processes (in $U - x$) to (via transition d or e)
    - prepare $cancellations$ (by walking to that place)
    - inform the master to have answered (never mind what) by walking to place $answered\ slave$

NB. *The requirements were*: 'Each slave ... reports ... acceptance or refusal to the master' and 'In case one slaves refuses, the master sends a cancellation to each slave' (op.cit. p.119)

Master/slave Petri net verification: (1) rename places

$$\Sigma_{75.1} \models B \hookrightarrow A \wedge (N.U \vee P.U). \qquad (1)$$

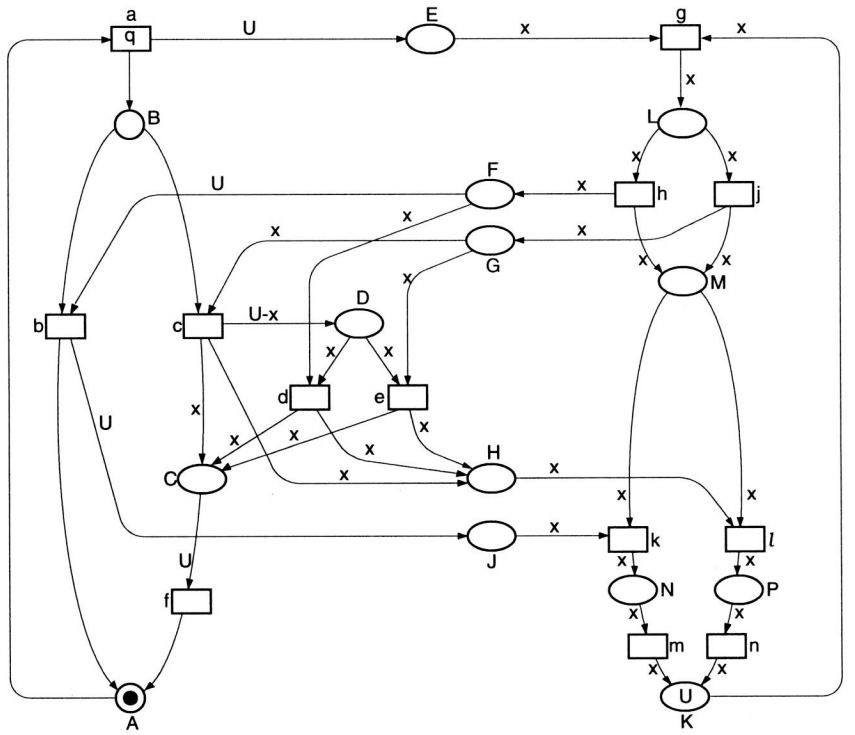


Figure 75.1. Renamed master/slave agreement $\Sigma_{30.1}$

Step 1: redraw the diagram and rename the places (p.255)
'The essential aspect ... in the redrawn version ... is formally represented by (1)'

Master/slave PN verification: (2) prove auxiliary invariants

75.2 State properties

Proof of (1) is based on the following place invariants of $\Sigma_{75.1}$:

inv1: $E + L + F + G - D - U * |B| = 0$
inv2: $F + G + H + J + N + P + K + L = U$
inv3: $U * A + U * B + C + D = U$
inv4: $F + G + J + H - M = 0$
inv5: $H + J + N + P + K - E - U * A - C = 0$
inv6: $L + M + N + P + K = U$

inv4 and inv6 imply $F + G + J + H \leq U$.

Master/Slave: (3) ‘A proof graph for the essential property’

$$\alpha = B \wedge (E + L + F + G \geq U). \qquad (3)$$

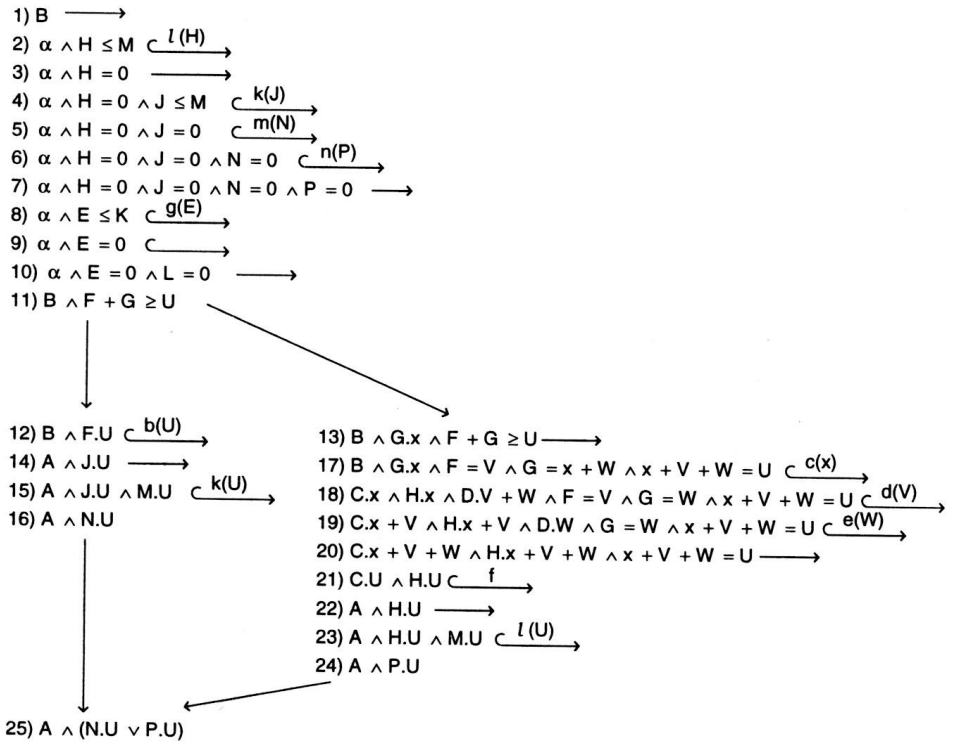


Figure 75.2. Proof graph for (1), with shorthand (3)

The nodes of this graph are justified on p.257-258 using some previously defined and justified ‘proof patterns’ (ibid., Sect.69)

Acknowledged Broadcast Algorithm

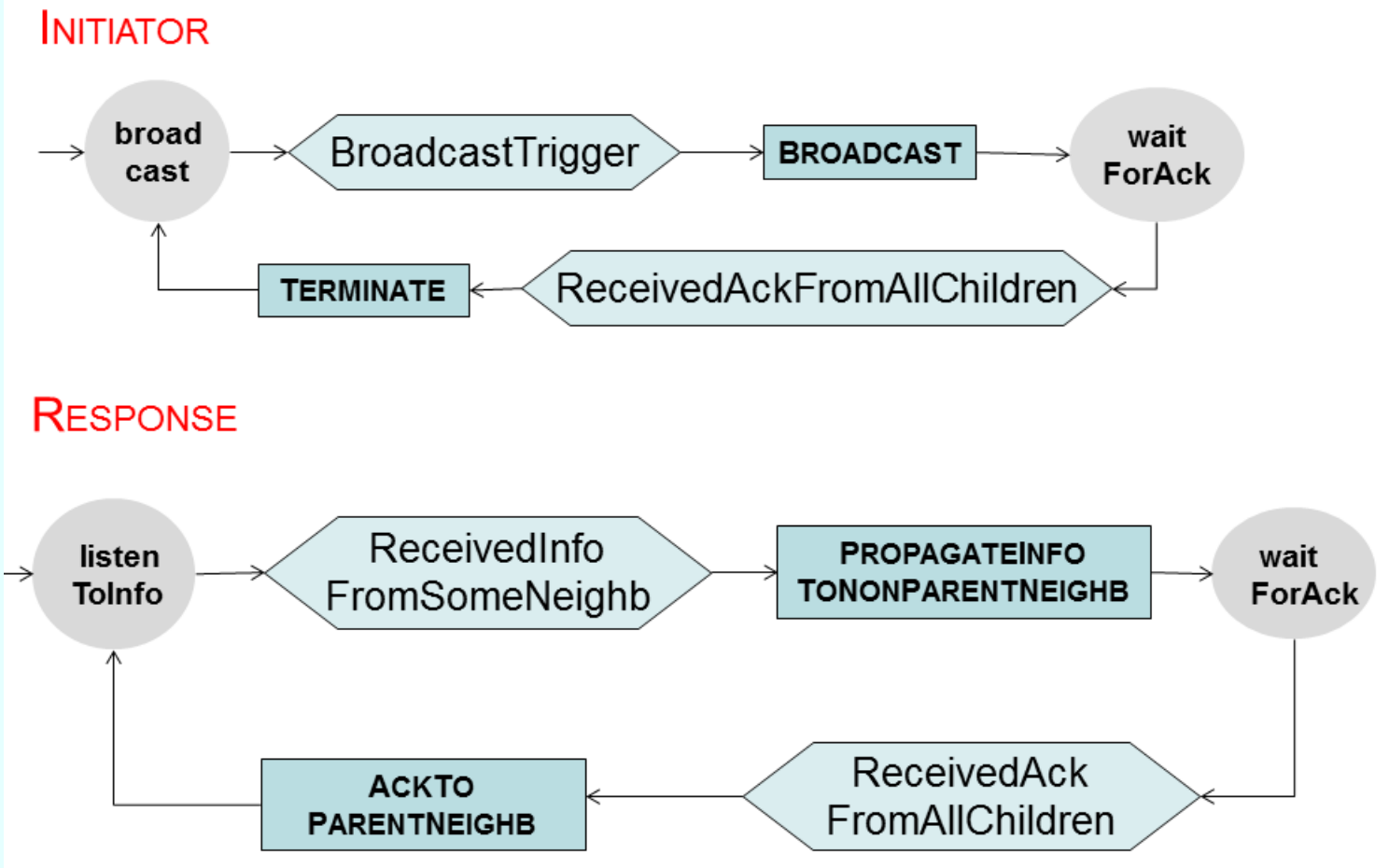
Requirement (Lynch 4.2.2)

- Distributed algorithm that guarantees an *initiator*'s msg being broadcast (building a spanning tree) and acknowledged (echoed) through a connected undirected graph communicating bw *Neighb*

Algorithmic Idea

- distinguished *initiator* upon *BroadcastTrigger* will BROADCAST an info msg to its *Neighb*ors and then *waitForAck* msgs from them
- if a not yet informed non-initiator node *ReceivedInfoFromSomeNeighb*or it will PROPAGATEINFOTONONPARENTNEIGHBors and then *waitForAck*
- once a non-initiator *ReceivedAckFromAllChildren* it in turn sends an ACKTOPARENTNEIGHB node by which it had been informed
- initiator once it *ReceivedAckFromAllChildren* TERMINATEs

From English to ASM: 2-phase ECHO algorithms



NB. Upper lines describe building a Breadth-First Spanning Tree, lower lines notification of completion from leaves back to initiator.

From English to ASM: ECHO state (signature)

- $Node \cup \{initiator\}$: *connected undirected graph of agents* with locs:
  - *Neighb*, the set of neighbors of a node
  - mailbox (called $EchoMsg$) for messages $infoFrom(n)$, $ackFrom(n)$, $IamYourChild(n)$, $IamNotYourChild(n)$
    - where $n \in Node \cup \{initiator\}$, initially empty
  - *parent* $\in Node \cup \{initiator\}$, initially **undefined**
    - records a neighbor node who has sent an info msg which is to be acknowledged—once all $Children$ to whom the info msg is forwarded have acknowledged that msg (not needed for $initiator$)
- $initiator$ equipped with program INITIATOR, input location (monitored event) $BroadcastTrigger$, initially $mode = broadcast$
- each $a \in Node$ equipped with program RESPONSE, initially $mode = listenToInfo$

From English to ASM: ECHO initiator/response actions

$\textsc{Broadcast} =$ -- *triggers building breadth-first spanning tree*

  forall $n \in Neighb$ $\textsc{Send}(infoFrom(initiator), \textbf{to}\ n)$

$\textsc{Terminate} = \textsc{Empty}(EchoMsg)$ -- empty mailbox for next round

$ReceivedInfoFromSomeNeighb =$

  forsome $p \in Neighb$ $Received(infoFrom(p))$

$\textsc{PropagateInfoToNonParentNeighb} =$ -- *tree building step*

  choose $p \in Neighb$ **with** $Received(infoFrom(p))$

    forall $n \in Neighb \setminus \{p\}$ $\textsc{Send}(infoFrom(\textbf{self}), \textbf{to}\ n)$

    $parent := p$ -- define receiver of later ackFrom msg

    $\textsc{InformAboutChildReln}(p)$

From English to ASM: ECHO response actions

$ReceivedAckFromAllChildren =$ -- true at leafs

  $ChildKnowlIsComplete$ **and forall** $m \in Children$

    $Received(ackFrom(m))$

$Children_n = \{m \in Neighb_n \mid parent(m) = n\}$

ACKTOPARENTNEIGHB = -- *pass notification along spanning tree*

  SEND$(ackFrom(\textbf{self}), parent)$

  $parent :=$**undef** EMPTY$(EchoMsg)$ -- clear for next round

INFORMABOUTCHILDRELN$(p) =$

  SEND$(IamYourChild(\textbf{self}), p)$

  forall $q \in Neighb \setminus \{p\}$ SEND$(IamNotYourChild(\textbf{self}), p)$

$ChildKnowlIsComplete$ iff **forall** $n \in Neighb$

  $Received(IamYourChild(n))$ **or** $Received(IamNotYourChild(n))$

ECHO correctness property

Proposition

In every concurrent ECHO run (where each enabled agent will eventually make a move):

- each time the *initiator* performs a BROADCAST of an *infoFrom* msg it will eventually TERMINATE (termination)
- the *initiator* will TERMINATE only after all other agents have *Received* that *infoFrom* msg and have acknowledged this to their *parent* neighbor (correctness)

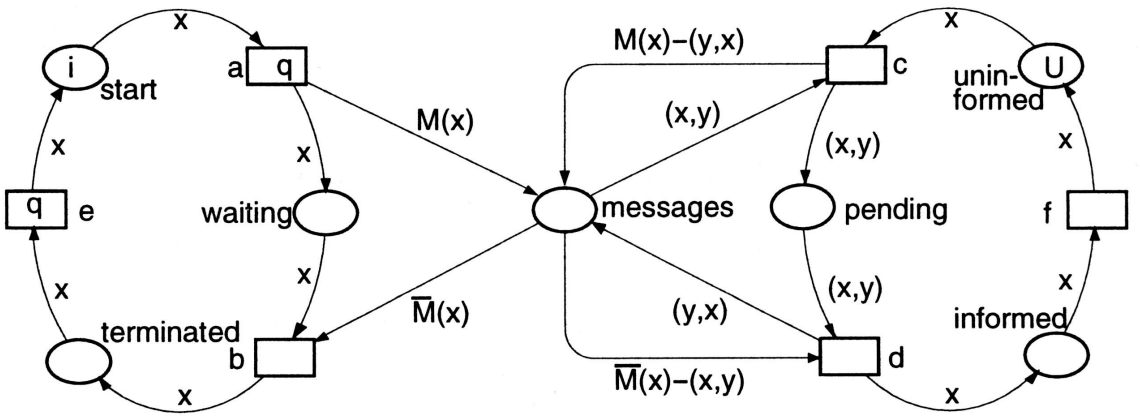
Proof. Follows by run induction from two lemmas.

ECHO correctness proof

- **Lemma 1**. In every concurrent (completed) ECHO run, each time an agent executes PROPAGATEINFOTONONPARENTNEIGHBors, in the tree of agents $waitingForAck$ the distance to the $initiator$ grows until leafs are reached.
  - Proof by downward induction on ECHO runs
- **Lemma 2**. In every concurrent (completed) ECHO run, each time an agent executes ACKTOPARENTNEIGHBor, in the 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 ECHO runs

Compare such step-properties-based intuition-guided inductions with technically involved lengthy proof graph verification in Reisig p.260-266

Petri net encoding of Echo algorithm



sort site
sort message = site × site
const i : site
const U : set of sites
const W : set of (sites × sites
fct M, $\overline{M}$: site → set of messages
var x, y: site

$W = W^{-1}$
$x,y \in U \cup \{i\} \rightarrow x\ W^{*} y$
$W_1 = U \cup \{i\}$
$i \notin U$
$M(x) = W(x) \times \{x\}$
$\overline{M}(x) = M(x)^{-1}$

Figure 33.3. Cyclic echo algorithm

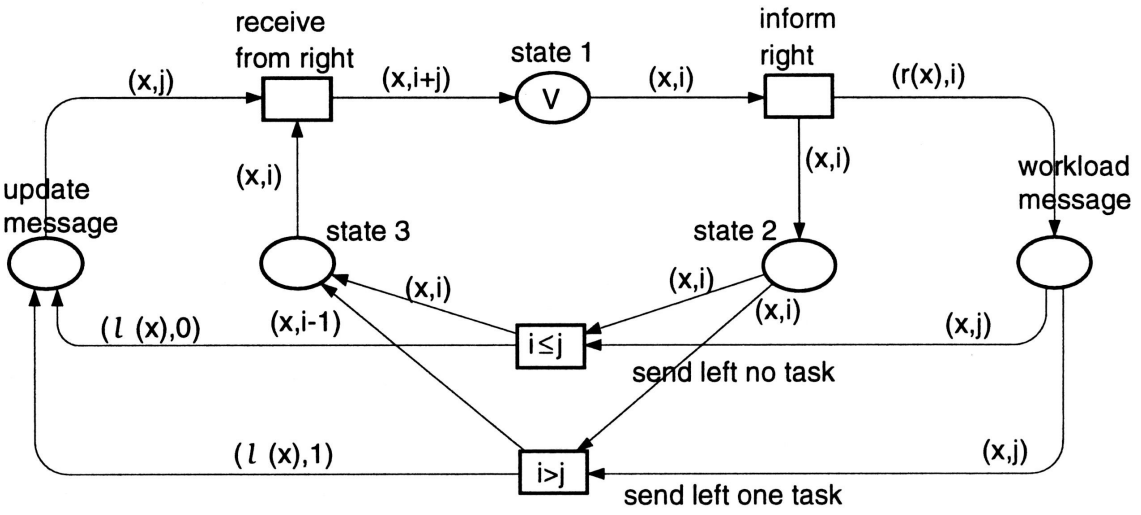
NB. 2-page explanation of this encoding in op.cit. p. 127-129.
Hides intuition of underlying spanning tree method.

Load Balancing in rings (modeling agents' interaction)

- Goal: distributed algorithm for reaching workload balance among a fixed set of (say at least 3) processes in a given ring using
  - communication only between right/left neighbors
  - abstract message passing and task transfer mechanism
  - fixed total workload (below made subject to dynamic change)
- Algorithmic Idea: every process (ring node) alternately
  - SENDs
    - a *LeftNeighbLoad* message (i.e. its $workLoad$) to its $rightNeighb$or,
    - a task *Transfer* mssg *to* its *left neighbor* to balance their workloads
  - when $ReceivedTransfer$ msg t *from its right neighbor accepts the task* t (if $t \neq nothingToTransfer$) to balance their workloads.

so that eventually the workload difference of two nodes becomes ≤ 1

Petri net encoding of distributed load balancing



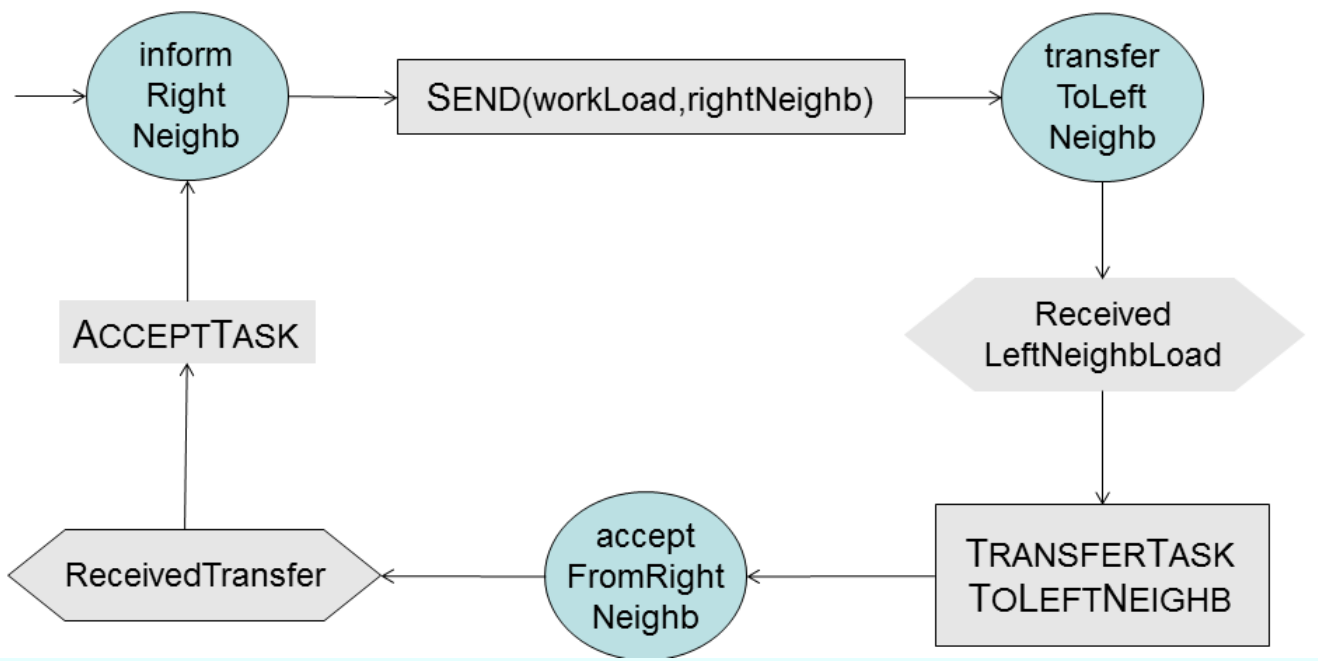
sort site
sort alloc = site× nat

const U : set of sites
const V : set of alloc
fct l , r : site → site

var i, j : nat
var x,y : site
∀ x ∈ U ∃ $_1$i∈ nat : (x,i)∈ V
x ≠ y ⇒ r(x) ≠ r(y)
∃ n ∈ nat : $r^n(x) = x$
∀x ∀y ∃ n ∈ nat: $y = r^n(x)$
l (r(x)) = x

Figure 37.1. Distributed load balancing

From English to ASM: 3-phase RingLoadBalance



- flowchart encoded in $state\ i$-Petri-subnet ($i = 1, 2, 3$)
- places $workloadmessage/updatemessage$ are global (Sic) mailboxes – one mailbox for all processes
- transfered tasks themselves are not represented in the PN (Sic)

From English to ASM: states of Ring Load Balancing

Process, a static set of agents, each equipped with

- static *ring structure* with *leftNeighb, rightNeighb* $\in Process$
- the current $WorkLoad \subseteq Task$
  - $workLoad = |\ WorkLoad\ |$ count (derived location)
- mailbox called $WorkLoadMsg$, initially empty
  - containing $LeftNeighbLoad$ msgs from the left neighbor
    - i.e. a $workLoad$ sent by its $leftNeigb$

    or task $Transfer$ msgs (from the right neighbor)
    - i.e. $transfer \in Task \cup \{nothingToTransfer\}$
- $mode$ switching from initial value $informRightNeighb$ to $transferToLeftNeighb$ back to $acceptFromRightNeighb$
- program RingLoadBalance

From English to ASM: Interaction with left neighbor

$ReceivedLeftNeighbLoad$ iff

thereissome $n \in Nat \cap WorkLoadMsg$

$\textsc{TransferTaskToLeftNeighb} =$

let $\{leftNeigbLoad\} = WorkLoadMsg$

if $workLoad > leftNeigbLoad$ -- there is a task to transfer

then choose $task \in WorkLoad$

$\textsc{Send}(task, \textbf{to}\ leftNeighb)$

$\textsc{Delete}(task, WorkLoad)$

else $\textsc{Send}(nothingToTransfer, \textbf{to}\ leftNeighb)$

$\textsc{Consume}(leftNeighbLoad)$

From English to ASM: Interaction with right neighbor

$ReceivedTransfer$ iff

thereissome $t \in (Task \cup \{nothingToTransfer\}) \cap WorkLoadMsg$

$\textsc{AcceptTask} =$

let $\{transfer\} = WorkLoadMsg$

if $transfer \in Task$ **then** $\textsc{Add}(transfer, WorkLoad)$

$\textsc{Consume}(transfer)$ -- msg removal from mailbox

RingLoadBalance correctness property

Proposition. In every concurrent run of processes equipped with RingLoadBalance, if in the run every enabled process will eventually make a move, then eventually the workload difference between two nodes becomes and remains at most 1 and the total workload remains constant.

Proof by induction on the $workLoad$ count differences. Let w be the sum of $workLoad$ count of all processes, $a =\mid Process \mid$.

- Case 1: $a|w$. Then eventually $workLoad(p) = w/a$ for every process p.
- Case 2: otherwise. Then eventually the $workLoad$ of nodes will differ by at most 1.

Compare with complicated 6-pages-long proof in Reisig pg.291-297.

Ring load balance with dynamic $WorkLoad$ change

Requirement: environment may trigger to ChangeWorkLoad by increasing or decrasing $WorkLoad$ by a set of tasks (Reisig 37.4)

Modeling idea: each process watches a trigger (monitored location)

$workLoadChange$ with values in $\{add(T), delete(T), noInput\}$

When triggered (by the env!) the process will ChangeWorkLoad, otherwise it executes RingLoadBalance (as before)

From English to ASM:

$WorkLoadChange$ iff $workLoadChange \in \{add(T), delete(T)\}$

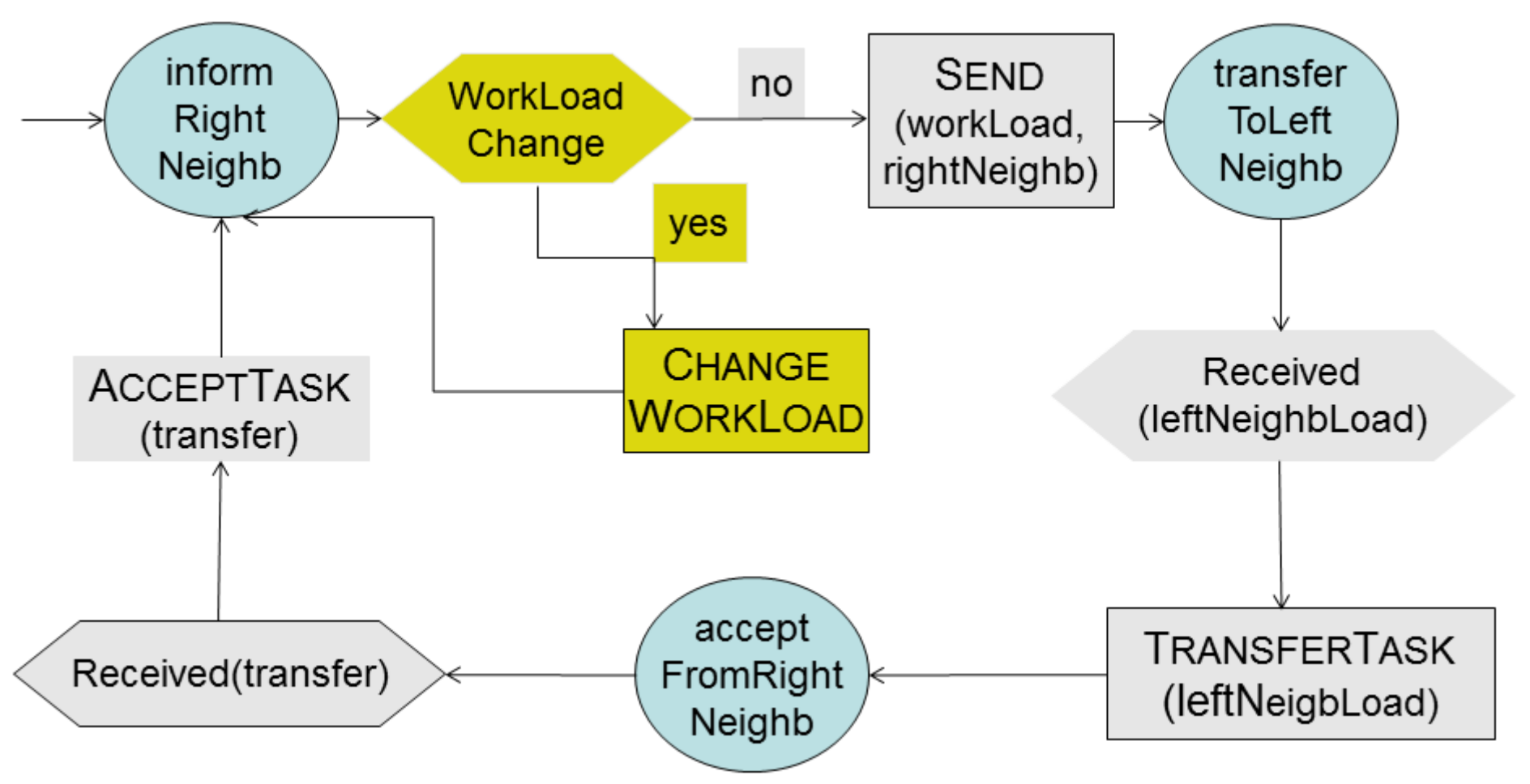
ChangeWorkLoad =

if $workLoadChange = add(T)$ **then** Add$(T, WorkLoad)$

if $workLoadChange = delete(T)$ **then** Delete$(T, WorkLoad)$

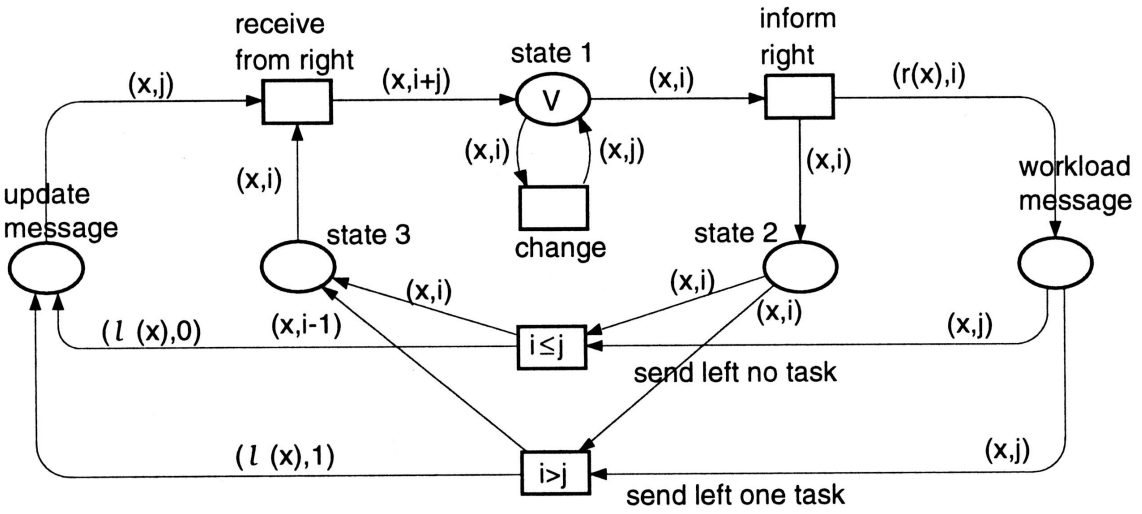
Consume$(workLoadChange)$ -- input consumption

From English to ASM: DynRingLoadBalance algorithm



NB. Typical *conservative refinement*: $newProgram$ in $newCase$, otherwise unchanged (supporting componentwise design)

Petri net refinement: env trigger as nondeterminism



sort site
sort alloc = site× nat

const U : set of sites
const V : set of alloc
fct l , r : site → site

var i, j : nat
var x,y : site
$\forall\ x \in U\ \exists_1 i \in$ nat : $(x,i) \in V$
$x \neq y \Rightarrow r(x) \neq r(y)$
$\exists\ n \in$ nat : $r^n(x) = x$
$\forall x\ \forall y\ \exists\ n \in$ nat: $y = r^n(x)$
$l\ (r(x)) = x$

Figure 37.2. Distributed load balancing in a floating environment

Petri net idiosyncrasy (global system view):

external environment triggers are modeled as internal transition

Abuse of nondeterminism

- the lack of component structure in PNs
  - more generally of the structure of the communication between agents and/or their environment

  leads to *model env actions by nondeterministic internal transitions*
  - here $change$ to model the interaction of the (one global?) environment with any local process, justified as follows:

  'From the perspective of the local balance algorithm, this interference shines up as nondeterministic change of the cardinality of the site's workload.' (op.cit. p.141-142)

Other exl below: msg loss by communication medium modelled as nondeterministic internal action of PN for file transfer protocol!

"input actions are assumed not to be under the automaton's control—they just arrive from the outside" (Lynch p.200)

Consensus in graphs (Dijkstra 1978)

Requirement (as interpreted by Reisig pg.134)

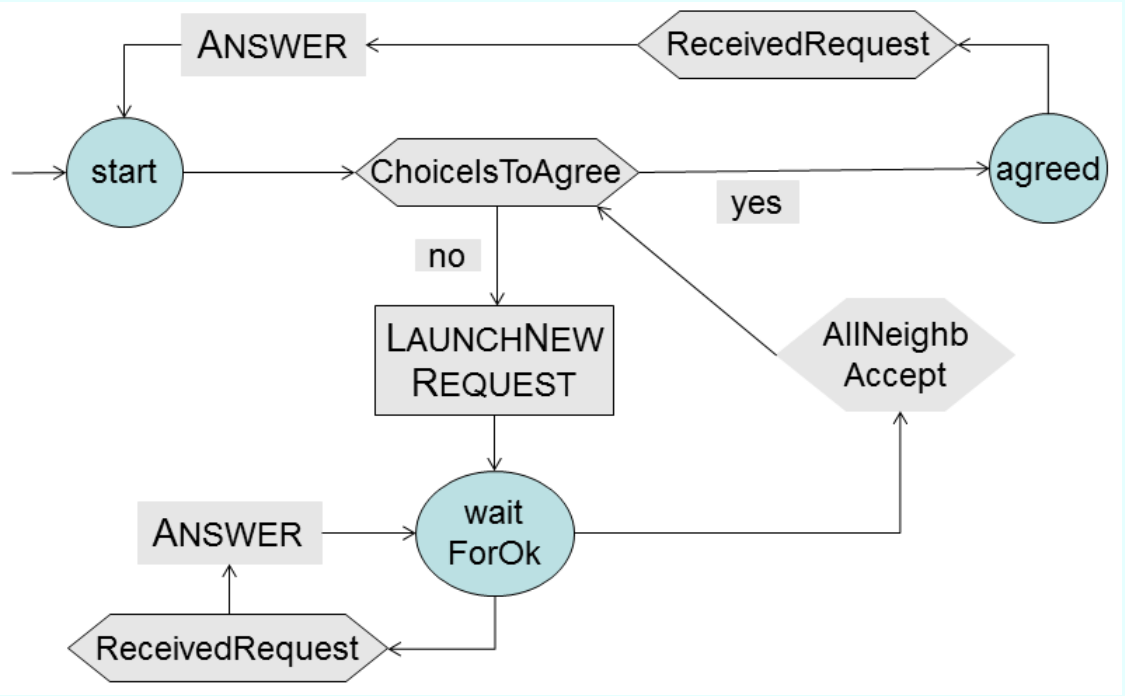
- Design a distributed algorithm to "organize consensus about some contract or agreement among the sites of a network" (graph), using only communication between neighbors, without considering "neither the contents of messages nor the criteria for a site to accept or refuse a proposed contract"

Algorithmic Idea. Every agent (site) has *4 possible actions*:

- spontaneously go to $agreed$ (when without new requests)
- LaunchNewRequest to its neighbors and $waitForOk$ from them
- receive and Answer requests
- if $AllNeighbAccept$ its last launched request either go to $agreed$ or once more LaunchNewRequest

Goal. Stability Property: if the algorithm terminates (maybe never), then all agents $agreed$ and there are no requests left.

From English to ASM: Consensus algorithm



ChoiceIsToAgree visualizes dominant nondeterminism in *start* phase (see below its role for the refinement by a quiet/demanded attribute)

- Petri net idiosyncrasy: nondeterminism is not directly visible but has to be extracted from the edge structure at pending sites

From English to ASM: CONSENSUS state

- $Node$: *graph of agents* (sites) with locations:
  - *Neighb*, the set of neighbors of a node
  - mailbox (called $ConsensusMsg$) for *messages* $requestFrom(n)$, $replyFrom(n)$ where $n \in Node$, mailbox initially empty
  - phases $mode \in \{start, waitForOk, agreed\}$, initially $mode = start$
  - program CONSENSUS
    - using a dynamic *choice function* agent chooses upon $start$ (e.g. non-deterministically, maybe otherwise) bw multiple enabled transitions

NB. Instead of abstracting from msg content, for further refinement one would better parameterize replies by the request to which they answer:

$$replyFrom(n, requestFrom(m))$$

For strict model comparability we do not pursue this further.

From English to ASM: Consensus agree/propose actions

$ChoiceIsToAgree$ iff $choice(\{agree, propose\}) = agree$

LaunchNewRequest = -- broadcast new request to neighbors

  forall $n \in Neighb$ Send($requestFrom(\textbf{self}), \textbf{to}\ n$)

  ReInitializeReplies

ReInitializeReplies =

  forall $n \in Neighb$ Consume($replyFrom(n)$)

$AllNeighbAccept = \textbf{forall}\ n \in Neighb\ Received(replyFrom(n))$

From English to ASM: Consensus reply action

$ReceivedRequest =$

  forsome $n \in Neighb$ $Received(requestFrom(n))$

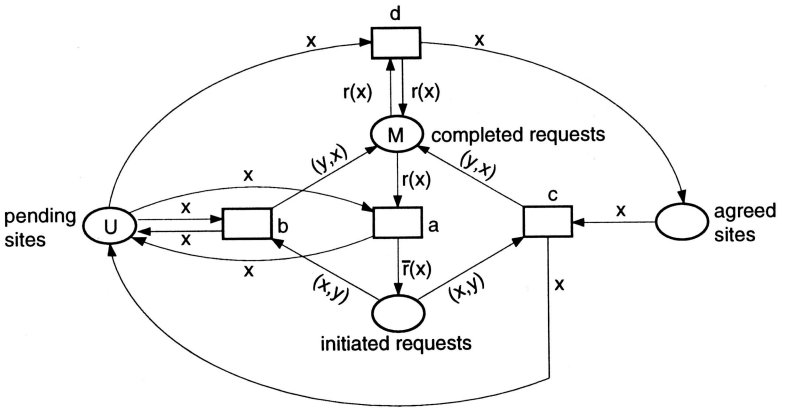
Answer = -- better specify using **forall** instead of **choose**

  choose $n \in Neighb$ **with** $Received(requestFrom(n))$

    Send($replyFrom($**self**$),$ **to** $n)$

    Consume($requestFrom(n)$)

Petri net model for Consensus



sort	site	fct	r, r̄ :site → set of messages
sort	message = site × site	var	x, y : site
const	U : set of sites	r(x) = {x} × M(x)	
const	M : set of messages	r̄(x) = M(x) ×{x}	

Figure 35.1. Basic algorithm for distributed consensus

Petri net idiosyncrasy: encoding of requests/answers by tokens $(y, x)/(x, y)$ in 'initiated/completed' yields artificial initialization

'Initially ... each msg is *completed* (i.e. in the hands of its sender)'

(encoding $mailbox = \emptyset$ at *start*) & makes req/answ checking tricky

Refining CONSENSUS by quiet/demanded sites

Requirements Change ('Advanced Consensus', Reisig p.136)

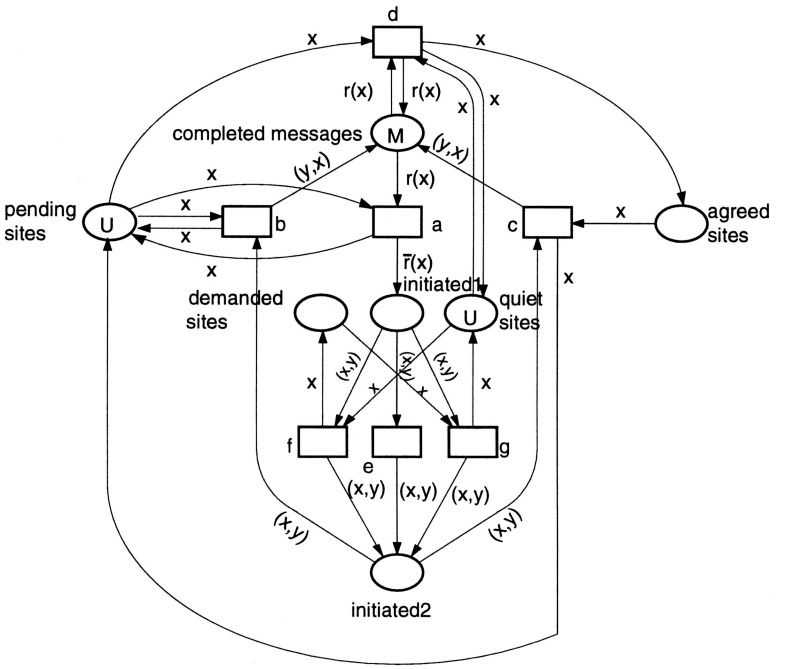
... two further states, *demanded sites* and *quiet sites*. All sites are initially *quiet*. Each newly sent message ... may cause its receiver ... to swap from *demanded* to *quiet* and vice versa... A demanded site u is not *quiet*. *If demanded and pending, the immediate step to agreed is ruled out*.

ASM refinement of structurally unchanged CONSENSUS by

- $Quiet \in \{true, false\}$, $Demanded =$ **not** $Quiet$
  attribute added to signature
- $\textsc{Swap}(Quiet) = (Quiet :=$**not** $Quiet)$: *action added* to ANSWER
- $Quiet = true$: *constraint added* to $ChoiceIsToAgree$ guard

That's all one needs to capture the requirements change!

‘Advanced Consensus’ Petri net (with quiet/demanded sites)



sort	site	fct	r, r̄ :site → set of messages
sort	message = site × site	var	x, y : site
const	U : set of sites	r(x) = {x} × M(x)	
const	M : set of messages	r̄(x) = M(x) × {x}	

Figure 35.2. Distributed consensus with demanded negotiators

Petri net idiosyncrasy: *token encoding of agent attributes* requires new places & structural diagram changes to restrict/add guards/actions

Proving stability property for Consensus

Proposition. In every terminating concurrent run of agents, each equipped with Consensus, the following holds:

- every agent agreed (read: is in $mode = agreed$)
- no request messages are left unanswered (read: for every agent its mailbox $ConsensusMsg = \emptyset$)

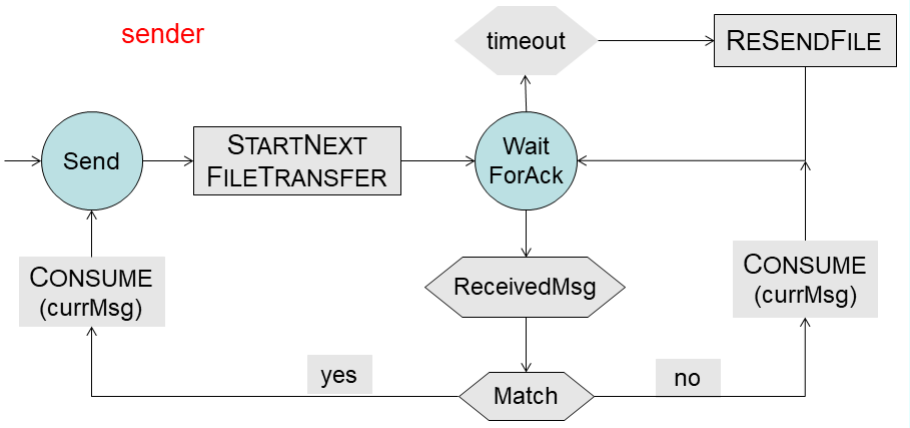
Proof follows from unfolding the definition of LaunchNewRequest and Answer.

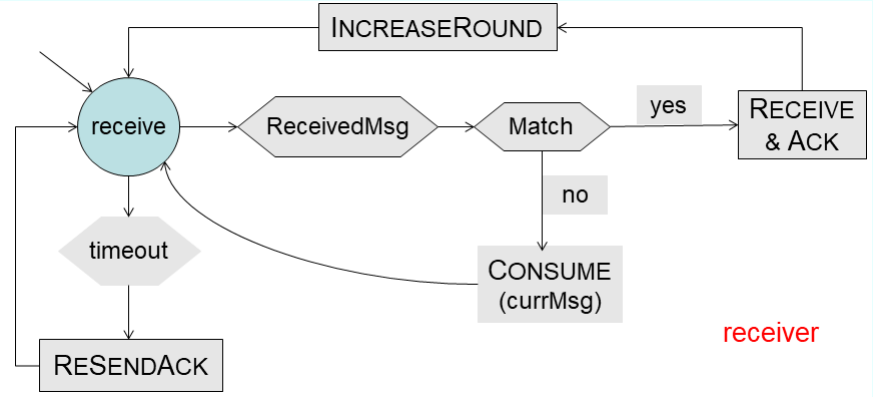
Compare to complicated proof (with additional complexity that results from Petri net technicalities) in op.cit., pg.279-283

File Transfer Acknowledgment: Alternating Bit Protocol

- **Goal**. Protocol to transfer a finite sequence $F(1), \ldots, F(n)$ of files from a sender to a receiver s.t. eventually the receiver has received the entire sequence (say $F = G$), assuming that (only finitely many consecutive) messages may get lost (but not changed) via the communication medium (which is kept abstract).
- **Algorithmic idea**. In rounds (one per file) $sender$ SENDs the current file and continues to RESENDFILE upon $timeout$ until an ack of receipt arrives from the $receiver$, whereafter in $currRound + 1$ $sender$ will STARTNXTFILETRANSFER until the sequence is transfered.
- When sending file $F(round)$ a sync bit $round \bmod 2$ is
  - attached to file msgs $(F(round), round \bmod 2)$
  - extracted and resent by the receiver as acknowledgment msg
  - checked upon $ReceivedMsg$ by $sender/receiver$ for $Match$ing own sync bit and in case of matching is flipped for next $round + 1$

From English to ASM: AltBit protocol





From English to ASM: Alternating Bit state

- mailbox $MsgQueue$: queue reflects no-msg-overtaking assumption
  - *FileMsgs* $(F(i), i \bmod 2)$ at $receiver$, *AckMsgs* $\in \{0, 1\}$ at $sender$
  - projection fcts to extract $file((f, b)) = f$ and $syncBit$
    $$syncBit((file, b)) = b, syncBit(bool) = bool$$
- *currRound* $\in \{0, 1, \ldots, n\}$: identifier of currently transmitted file
  - initially $currRound = 0$ at sender, $currRound = 1$ at receiver; receiver remains round-ahead of $sender$:
    $$currRound(sender) \leq currRound(receiver)$$
  - derived synchronization bit $currRound \bmod 2$
- $timeout$: interface to timer mechanism triggering resending
- file sequence locations F (at $sender$) and G (at $receiver$ initially $[\,]$)

Assumption that (only finitely many consecutive) messages may get lost means that for every SEND/RESEND... sequence eventually some SEND results in delivery into the recipient's mailbox.

From English to ASM: Alternating Bit sender actions

$\textsc{StartNextFileTransfer} =$

$\quad \textsc{Send}((nextFile, nextSyncBit), \textbf{to}\ receiver)$

$\quad \textsc{IncreaseRound}$

$\quad\quad \textbf{where}\ \ nextFile = F(currRound + 1)$

$\quad\quad\quad nextSyncBit = currRound + 1 \bmod 2$ -- flipped sync bit

$\textsc{ReSendFile} =$

$\quad \textsc{Send}((F(currRound), currRound \bmod 2), \textbf{to}\ receiver)$

$ReceivedMsg =$ iff $MsgQueue \neq [\,]$ -- mailbox not empty

$Match$ iff $syncBit(currMsg) = currRound \bmod 2$

$\textsc{CloseRound} = \textsc{Consume}(currMsg)$

$\quad \textbf{where}\ currMsg = head(MsgQueue)$

$\textsc{IncreaseRound} = (currRound := currRound + 1)$

From English to ASM: Alternating Bit receiver actions

$\textsc{Receive\&Ack} =$

$\quad \textsc{StoreFile}$

$\quad \textsc{SendAck}$

$\quad \textsc{Consume}(currMsg)$

$\textsc{ReSendAck} =$ -- NB. receiver is round-ahead of sender

$\quad \textsc{Send}(\mathit{flip}(currRound \bmod 2), \textbf{to}\ sender)$ -- previous sync bit

where

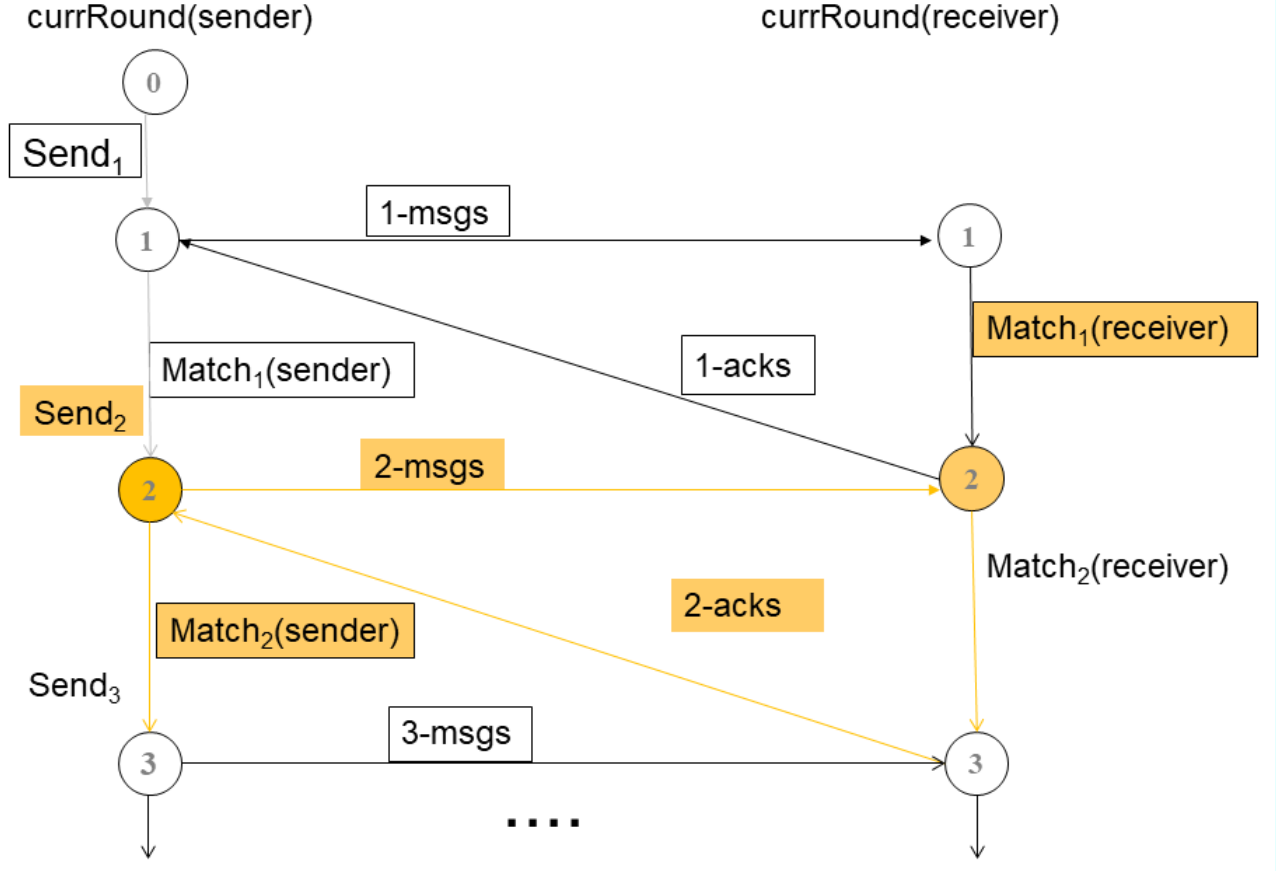
$\quad \textsc{SendAck} = \textsc{Send}(syncBit(currMsg), \textbf{to}\ sender)$

$\quad \textsc{StoreFile} = (G(currRound) := \mathit{file}(currMsg))$

NB. Assume $timeout$ to be initialized such that $\textsc{ReSendAck}$ can happen only after the first $\textsc{Receive\&Ack}$ (e.g. by initializing $timer = \infty$).

Correctness of AltBit runs (termination with $F = G$

is easily proved by induction on AltBit run phases:



In op.cit. we found no proof for the AltBit Petri net.

Alternating Bit Petri Net encoding

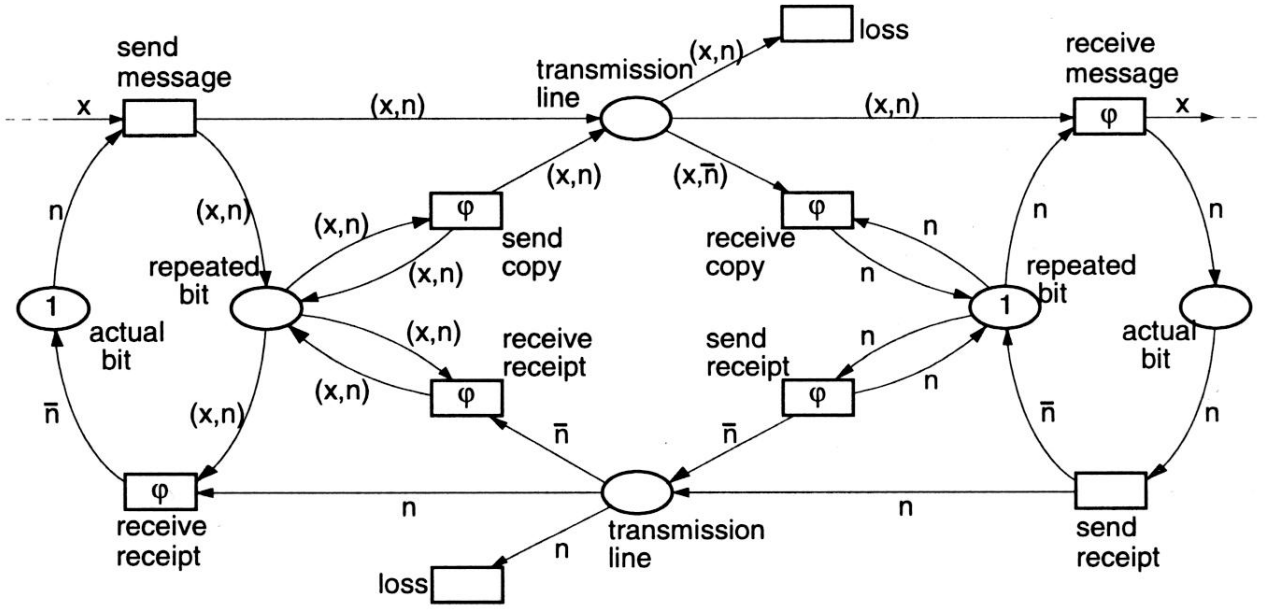


Figure 27.7. The alternating bit protocol

Petri net idiosyncrasies observable in AltBit net

Token-based transition view (token presence-checking/deletion/insertion)

- imposes *moving around unchanged data*—bw places or worse to delete and simultaneously add them from/to one place
  - technicality with overwhelming effect on net size and readability
    - analogous to *frame problem* of logical descriptions of no-change part of actions
- imposes *doubling of locations for same data* which are involved (possibly with different current values) in different transitions
  - e.g. $actualbit/repeatedbit$ places double $syncBit$ location at both sender and receiver part of the net
- invites to model natural $timeout$ *trigger* for resending *as* (here rather inappropriate) *non-determinism*

NB. Petri net technicalities for AltBit explained in op.cit. on 5 pages!

Improving AltBit protocol by 'sliding window' technique

Idea: get rid of no-msg-overtaking assumption replacing single file transfer rounds by (re)sending in any order multiple files $F(i)$ (and corresponding acks) distinguished by their index i

- above called $currRound$, now used as syncBit instead of $i \bmod 2$

in a *window* $[low, high]$ *between* low *and* $high$ s.t.

- initially $low = 1, high = 0$ at $sender$ and $receiver$
- $sender$ can perform StartNextFileTransfer and IncreaseWindow by new syncBit $high := high + 1$ as long as **not** $FullWindow$
- if $Acknowledged(low)$ (read: upon ack receipt of file with index low) $sender$ will ReduceWindow at its left end ($low := low + 1$)
- since $high_{receiver} \leq high_{sender}$, each time a file is received for the first time, its index i is larger than the $receiver$'s $high$ window end, triggering to SlideWindow at the right end by setting $high := i$ (and adapting its left end low correspondingly)

Petri net redesign: Sliding Window with unbounded indices

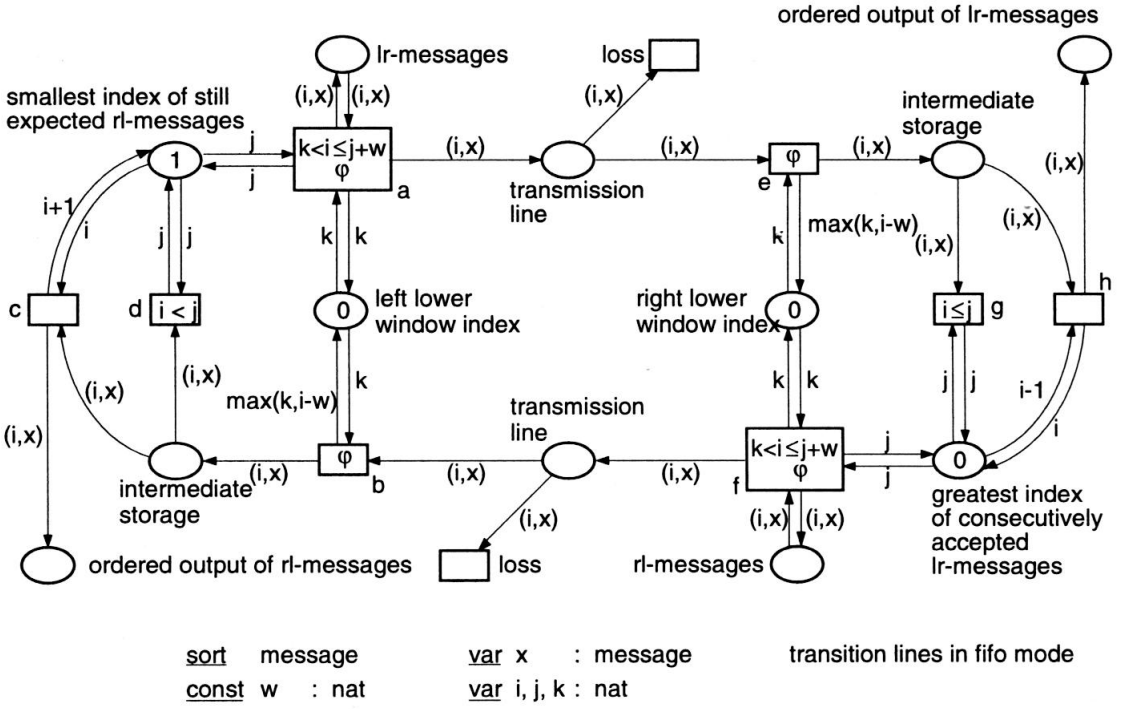
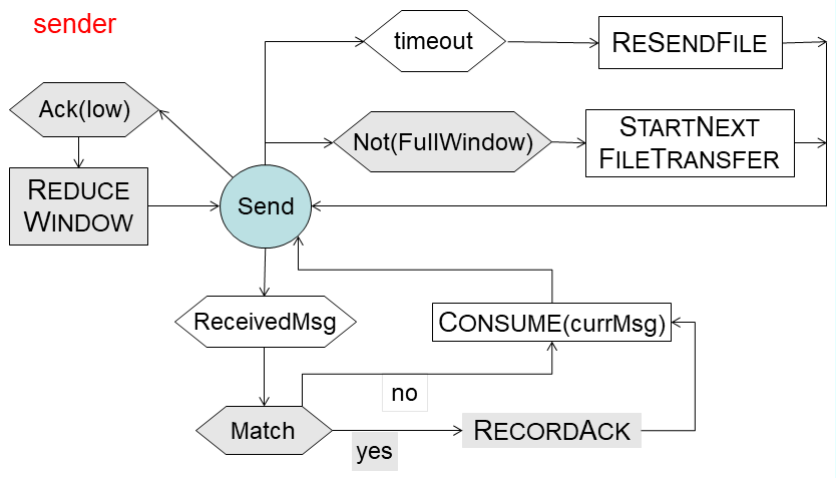


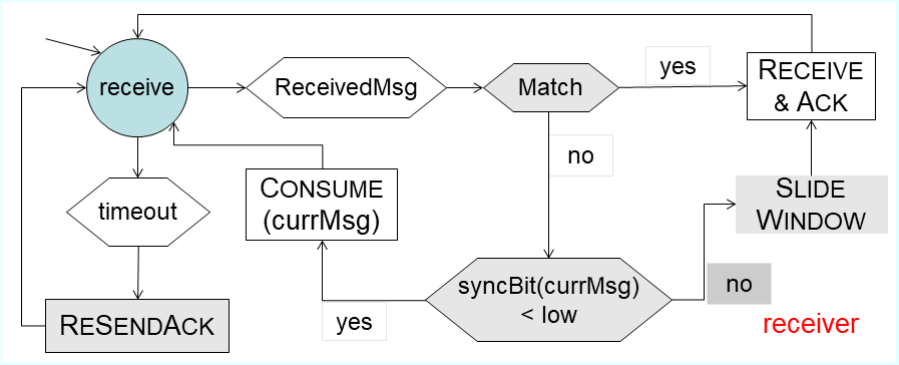
Figure 28.1. Balanced sliding window protocol with unbounded indices

Petri net idiosyncrasy: refinements tend to impose structural net changes (explained in op.cit on 3.5 pages!)

From English to ASM: SLIDINGWINDOW protocol



Component structure of ALTBIT *preserved* except for adding REDUCEWINDOW. Sequential send/waitForAck control flow collapsed



NB. All enabled actions can be performed in parallel

Sliding Window sender action refinement

- StartNextFileTransfer
  - guarded by **not** $FullWindow$
    - where $FullWindow = high - low + 1 = maxWinSize$
  - refined by

    $nextFile = F(high + 1), \quad nextSyncBit = high + 1$

    $\text{IncreaseRound} = (high := high + 1)$ -- IncreaseWindow

- $ReSendFile = \text{Send}((F(low), low), \textbf{to}\ receiver)$

- $Match = (low \leq syncBit(currMsg) \leq high)$ -- syncBit in window

- CloseRound refines $\text{Consume}(currMsg)$ by additionally recording that receipt of $currMsg$ has been acknowledged, i.e.

  $$Acknowledged(syncBit(currMsg)) := true$$

  initially $Acknowledged(i) = false$ for each i

Sliding Window receiver action refinement

- RECEIVE&ACK is not followed any more by INCREASEROUND
- in case of no $Match$ the $currMsg$ with $syncBit(currMmsg) > high$ cannot be CONSUMEd: the receiver must first SLIDEWINDOW to let $currMsg$ $Match$

  SLIDEWINDOW $=$ **let** $(s = syncBit(currMsg))$ **in**

  $high := s$

  $low := max\{1, s - maxWinSize + 1\}$

  NB. When receiving a new file with $FullWindow$, the ack for low must have been received by the $sender$ so that $low_{receiver}$ can be shifted to the right.
- $ReSendAck =$ SEND(*low*, **to** $sender$)

NB. Finitely many indices suffice using $+1 \bmod r$ for updating $low, high$ for a sufficiently large r depending on $maxWinSize$ if there is no msgt overtaking: a *pure data refinement*.

Petri net redesign: Sliding Window with bounded indices

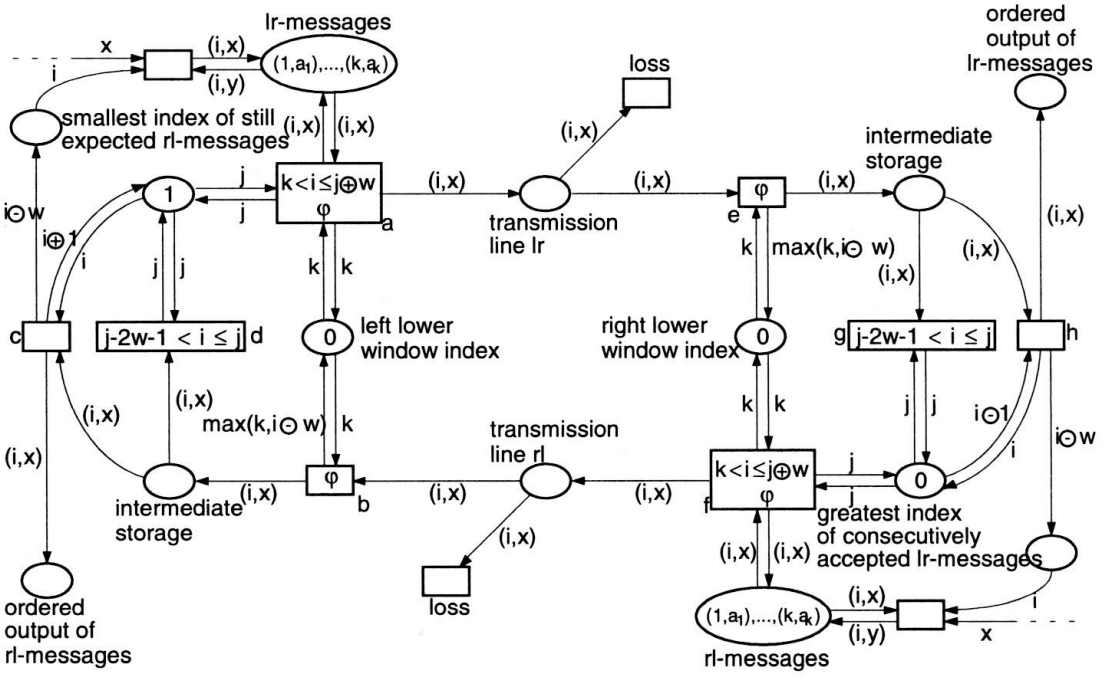
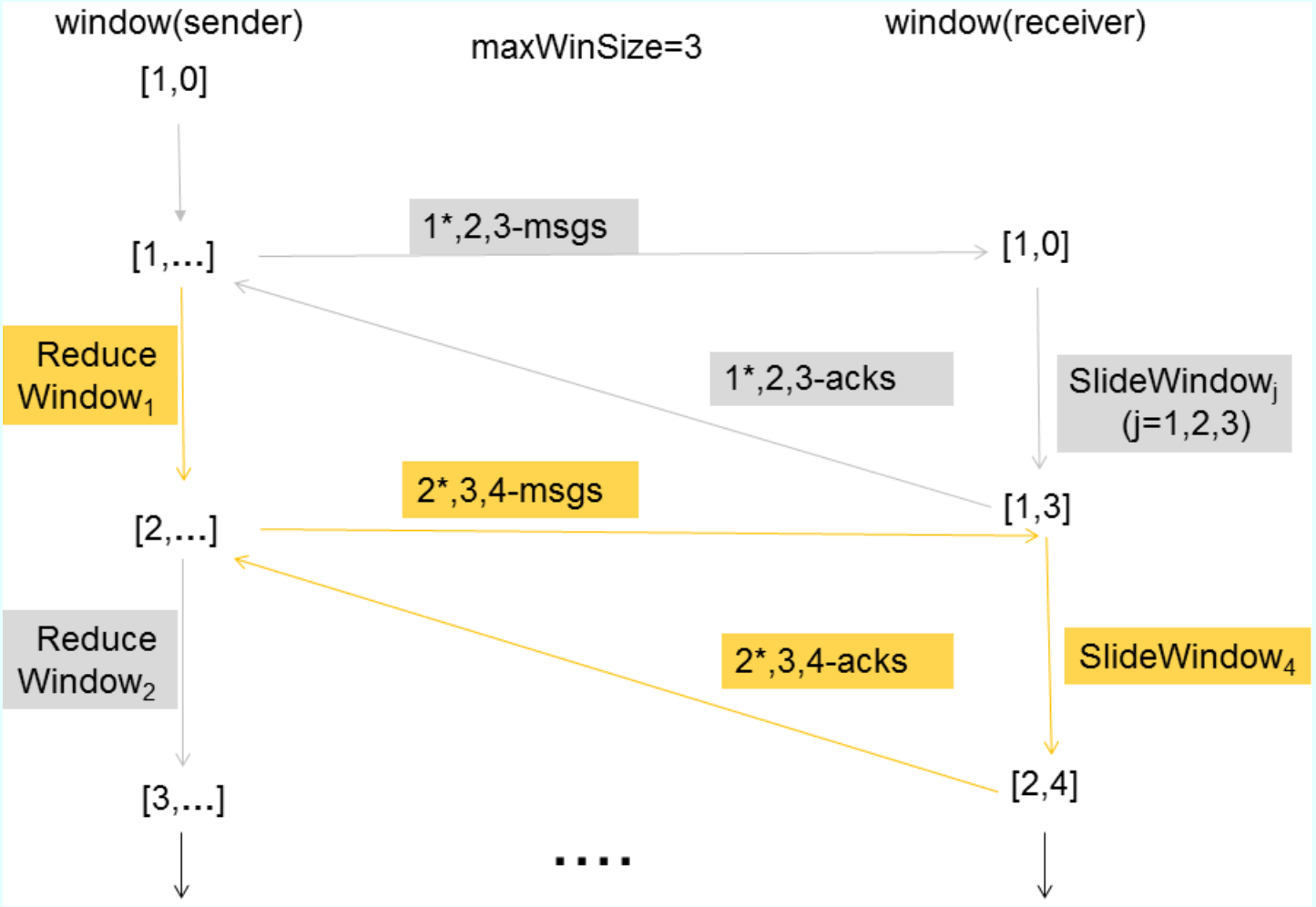


Figure 28.2. Balanced sliding window protocol with bounded indices

Correctness of SlidingWindows runs

is easily proved by induction on SlidingWindows run phases:



(we found no proof for this refinement in op.cit.)

Mutual Exclusion problem (Lynch 10.2, 10.4)

Problem: allocate *one nonshareable resource among n processes*

Algorithmic idea: each process from its remainder region (R)

- moves into a trying region (T) to gain access to the critical region (C)
- when the resource is not needed any more executes an exit protocol in its exit region (E)

$$R \longrightarrow T \longrightarrow C \longrightarrow E \longrightarrow R$$

Lockout-Freedom Requirement:

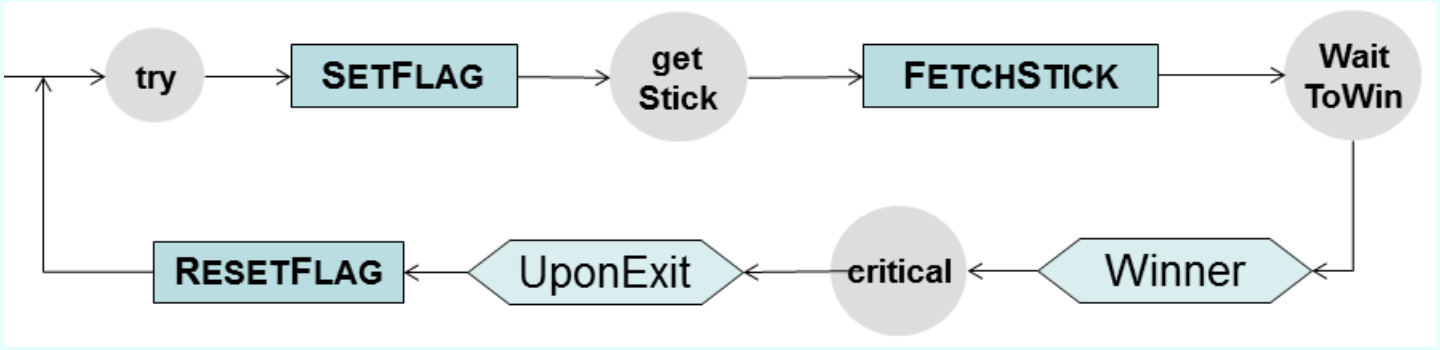
- If each process always returns the resource, every process that reaches the *trying* region *eventually* will enter the *critical* region
- Every process that reaches its *exit* region eventually will reenter its *remainder* region

Peterson's Mutex algorithm (Peterson 1981, Lynch 10.5)

- $Process = \{1, \ldots, n\}$, each with
  - for each local iterator variable $level \in \{0, \ldots, n-1\}$ (initially $level = 1$) a global shared loc
    - $stickAt(level) \in \{1, \ldots, n\}$ readable/writable by all processes with arbitrary initial value
    - p before getting the resource must $\textsc{Fetch}$ $stickAt(level)$
    - later release it (by being $\textsc{Fetch}$ed by another interested process) in case there is another interested process
  - local shared loc $\mathit{flag} \in \{o, \ldots, n-1\}$ at each process, initially $\mathit{flag} = 0$, writable by process p and readable by all other processes
    - $\mathit{flag}_p = l > 0$ indicating that p has started the competition ('is interested') at level l to get the resource

We first explain the case $n = 2$ with fixed competition $level = 1$.

From English to 4-phase MutexPeterson$_2$



Re/SetFlag $= (flag := 0/level)$

FetchStick $=$ // for $n = 2$, one more guard below for $n > 2$

if (**not** $HasStick_{level}$) **then** $stickAt_{level} :=$ **self**

-- NB. This means **skip** if $HasStick_{level}$

where $HasStick_{level}$ iff $stickAt_{level} =$ **self**

$Winner = NobodyElseInterested$ **or**

$MeantimeSomebodyElseFetchedStick$

Actions and Predicates for MUTEXPETERSON_2

$NobodyElseInterested(level) =$ **forall** $p \neq$ **self** $flag_p < level$

-- for case n$= 2$ this means $flag_{theOtherProcess} = 0$

where $theOtherProcess = \begin{cases} 1 \textbf{ if } \textbf{self} = 2 \\ 2 \textbf{ else} \end{cases}$ // only for case $n = 2$

$MeantimeSomebodyElseFetchedStick(level)$ iff $stickAt_{level} \neq$ **self**

-- meaning in case $n = 2$ that $stickAt_{level} = theOtherProcess$

Petri net for Peterson's Mutex algorithm

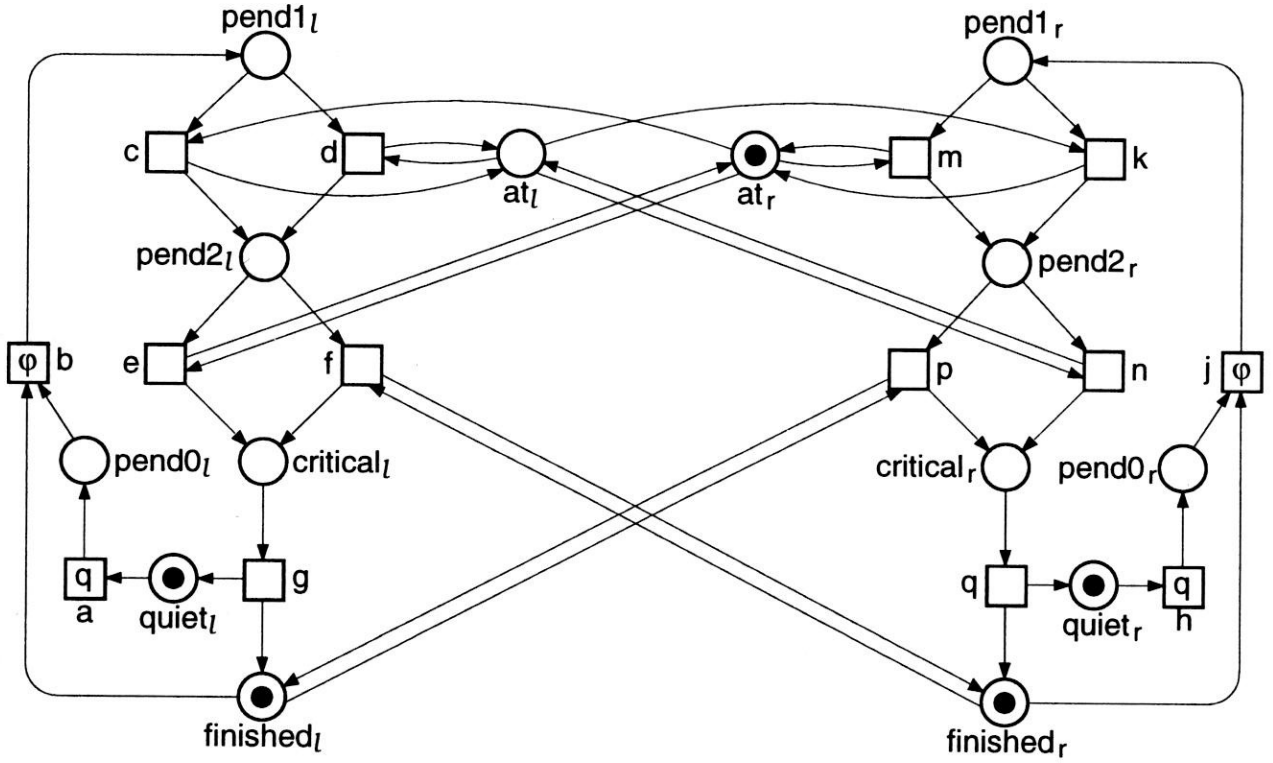


Figure 13.7. Peterson's *mutex* algorithm

Petri net idiosyncrasy: *Simulation of shared locations by token manipulation* multiplies places and transitions (8+12 for 3 locs)

Simulation of shared locations multiplies places/transitions

Petri net idiosyncrasy: PN for $\textsc{MutexPeterson}_2$ introduces

8 places and 12 transitions for 3 locs

- $stickAt$ reads/writes simulated by $pendi_l, pendi_r, at_l, at_r$ $(i = 1, 2)$
  - 2 token swapping transitions which simulate the 2 possible writes
  - 4 token checking read ('simultaneous delete/add token') transitions
- multiple-reader single-writer $flag$s encoded by places $finished_l, finished_r$ with
  - a reader transition to simulate reading the writer's $flag$ value
  - for the writer for each possible update value one transition
    - here two transitions encoded as delete resp. add token transitions at place $finished$ of each writer process

Celebrated Petri net visualization risks to result in spaghetti diagrams

- in presence of shared locs involving > 2 processes or > 2 loc vals

Proving mutual exclusion, progress and lockout-freedom

Compare detailed, easy-to-understand proof in Lynch 281-282 with Petri net verification using 'evolution proof graph' whose 16 'nodes ... are justified' one by one with help of 4 invariants (Reisig p.180-182).

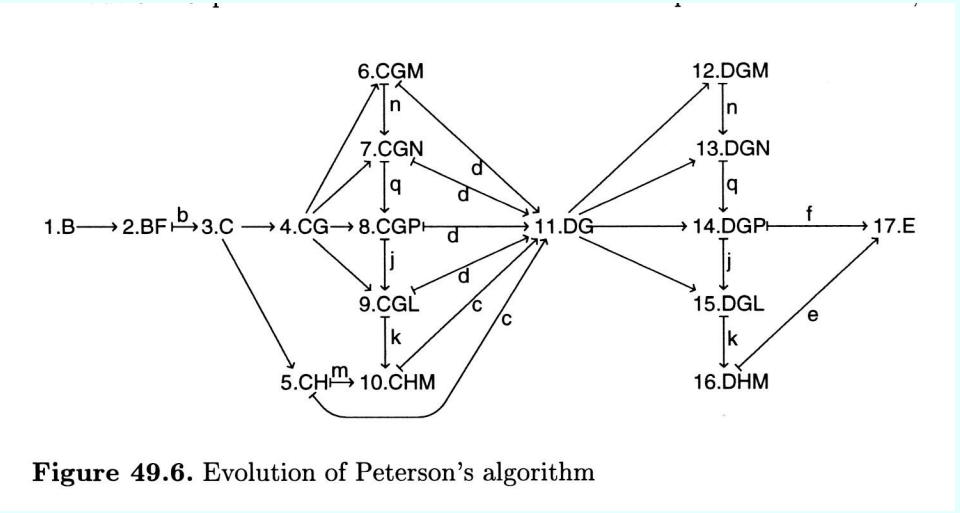


Figure 49.6. Evolution of Peterson's algorithm

Natural design/proof extensions from 2 to $n > 2$ processes (see Lynch 10.5.2 and 10.5.3) risk to explode with Petri net proof graphs.

Generalization to $\textsc{MutexPeterson}_n$ for $n > 2$

Algorithmic idea: iterate $\textsc{MutexPeterson}_2$ competition through $level$s 1 to $n-1$ s.t.

- for each level value there is a least one $loser$
  - a process that has to $waitToWin$ until $NobodyElseInterested$
- so that at level k at most $n-k$ processes can win
  - and therefore at most one can win at level $n-1$

Main refinement: add iterator component and *a further guard for* $\textsc{FetchStick}$ guaranteeing that

- in each step at most one process p can write $stickAt$

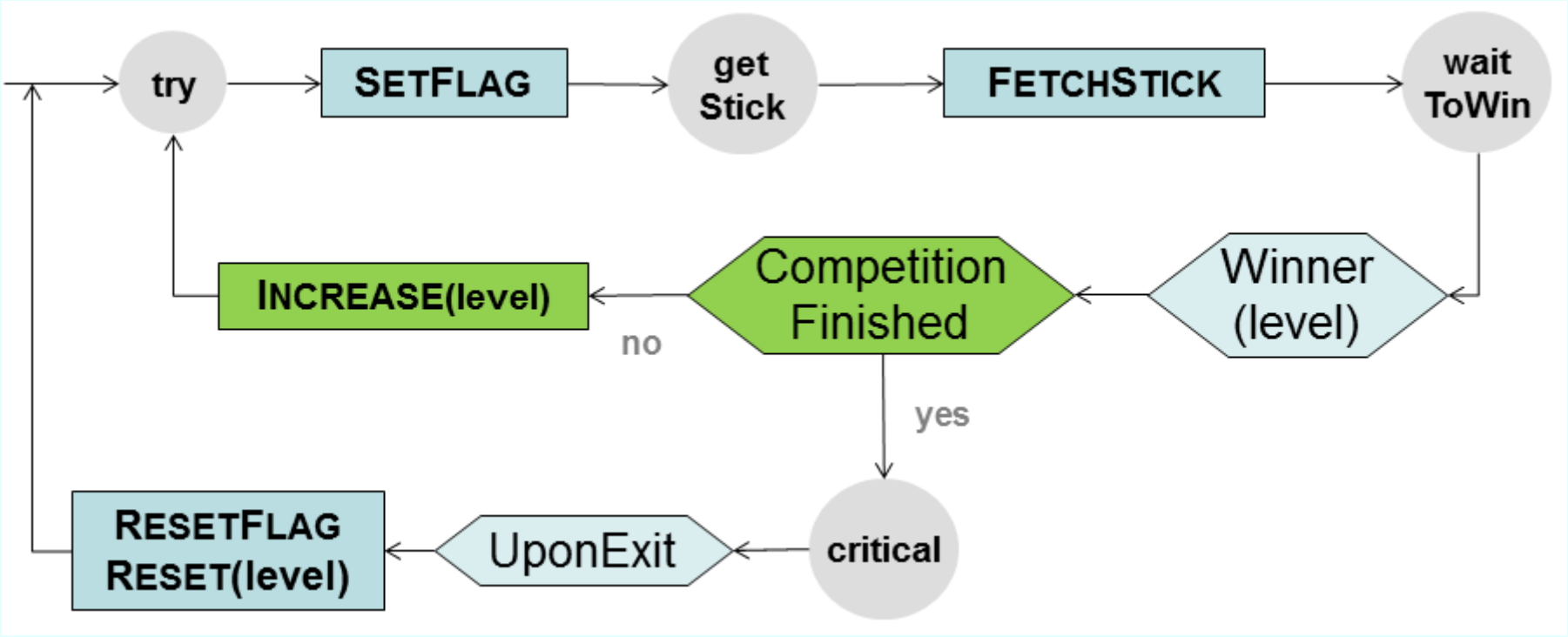
$chosenWriterFor(stickAt_{level}) =$ **self**

where $chosenWriterFor(stickAt(l)) =$

$select(\{p \mid flag_p = l$ **and** $mode_p = getStick$

and $stickAt(l) \neq p\})$

MutexPeterson$_n$ for $n > 2$



$CompetitionFinished$ iff $level = n - 1$

$\textsc{Increase}(level) = (level := level + 1)$

$\textsc{Reset}(level) = (level := 1)$

ASMs enhance Petri net expressivity, e.g. by choose/forall

Goal: generate (over alphabet A, say starting with empty $vw = \Lambda$) the pairs $vw \in A^*$ of different words $v \neq w$ of same length $|v| = |w|$

WordPairGenerator$(A) =$

choose $n, i \in Nat$ **with** $i < n$ -- at least one position

choose $a, b \in A$ **with** $a \neq b$ -- with different values

$v(i) := a$

$w(i) := b$

forall $j < n, j \neq i$ -- in every other position

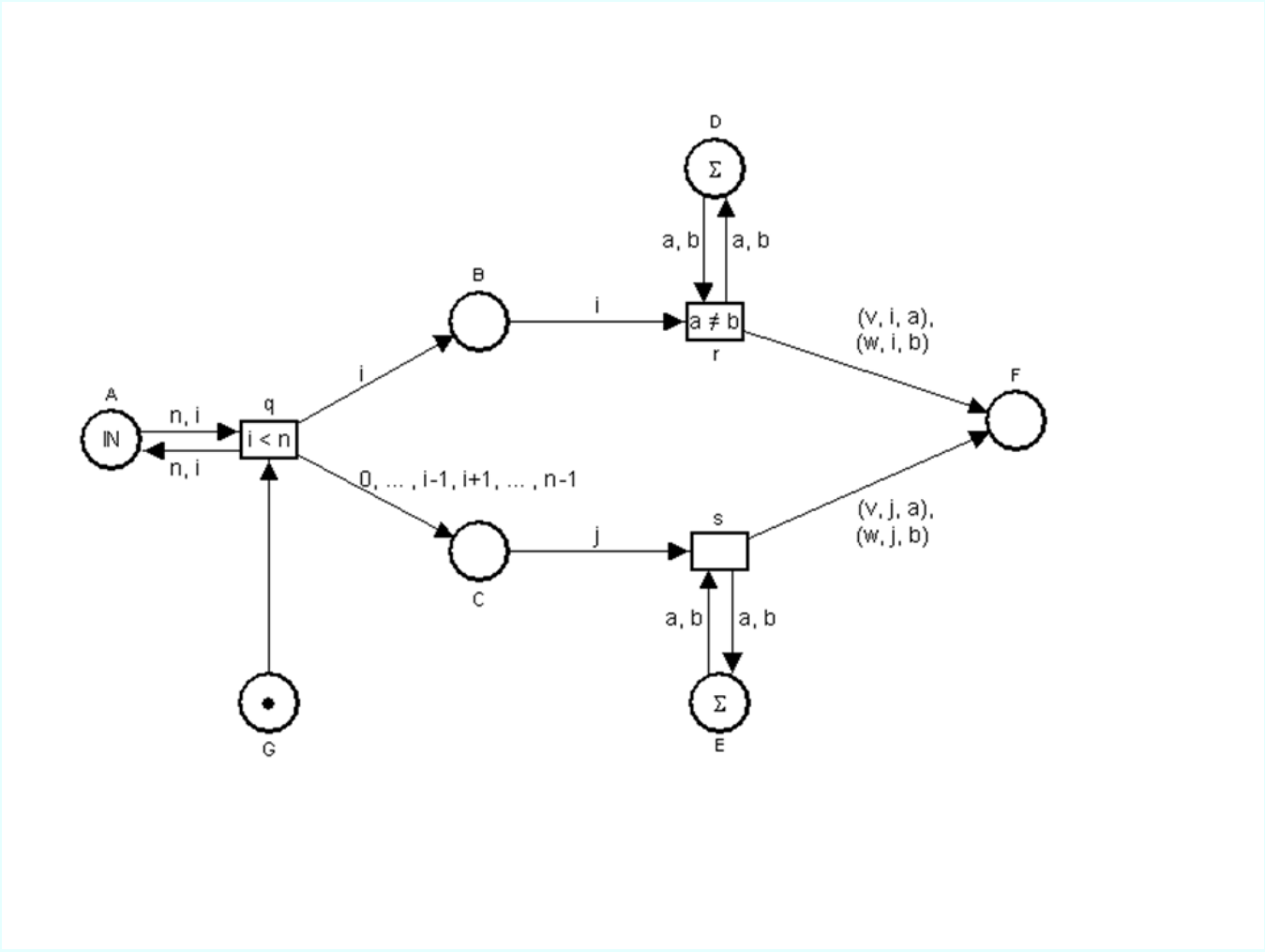
choose $a, b \in A$ -- whatever value

$v(j) := a$

$w(j) := b$

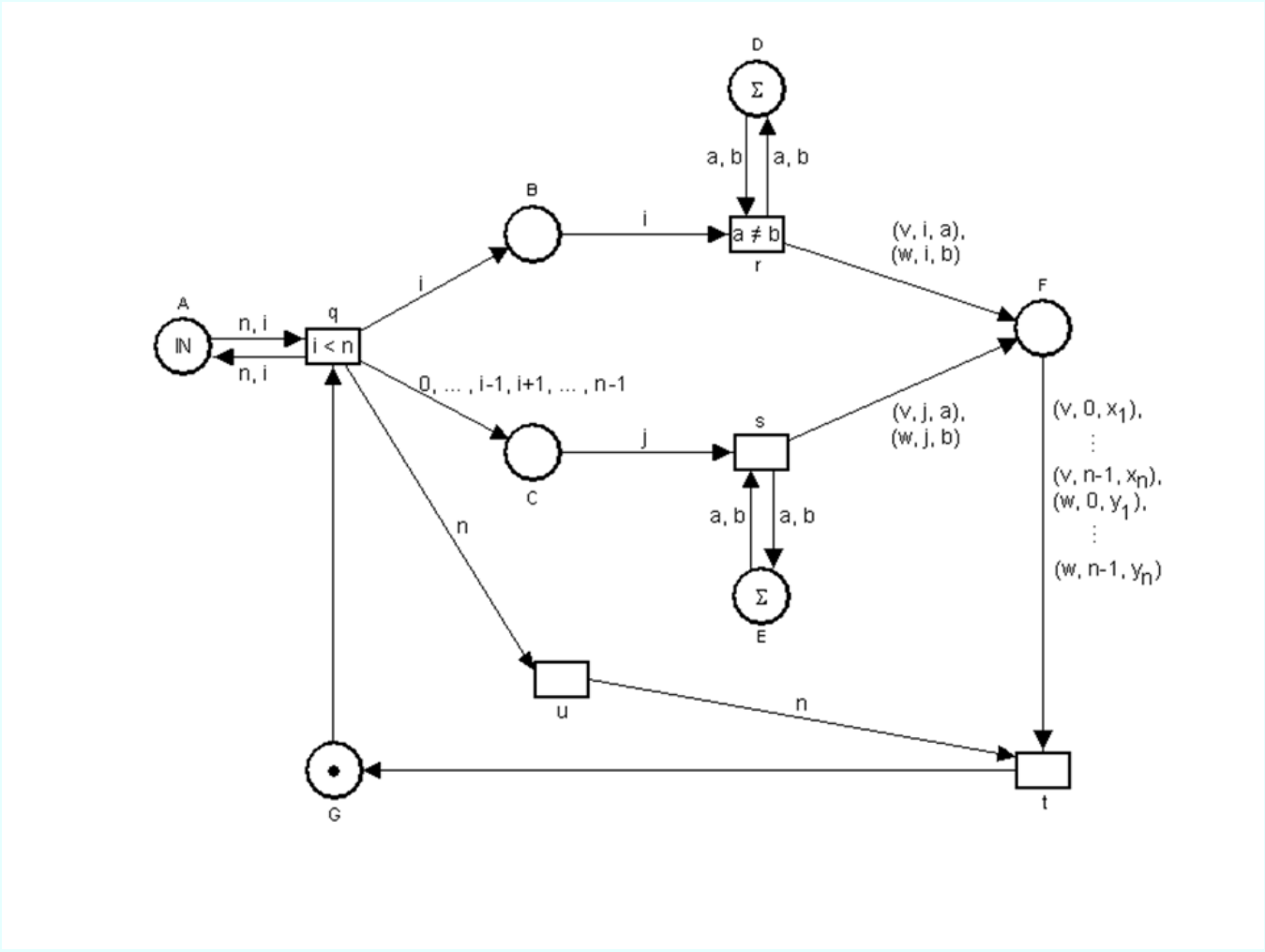
Correctness Lemma. The set of states reachable (from $vw = \Lambda$) by whatever possible choice is $\{vw \in A^* \mid v \neq w$ **and** $|v| = |w|\}$.

Petri net to generate one word pair (courtesy W. Reisig)



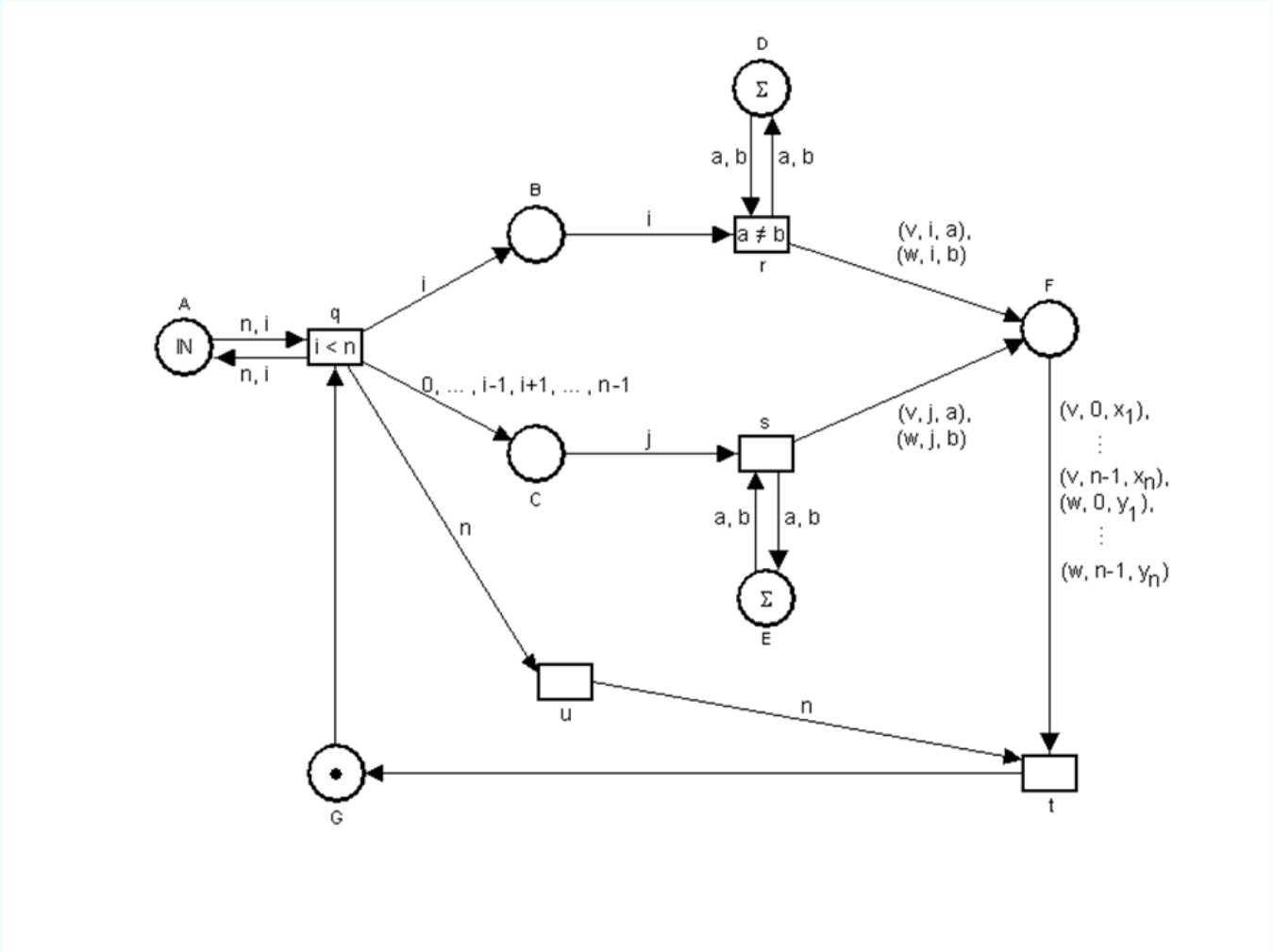
How much it takes to explain/prove that this net simulates *one* step of the above ASM (where const v, w; var $a, b \in \Sigma$; var $n, i, j : Nat$)?

Add iteration to generate all word pairs (courtesy W. Reisig)



Measure the effort to define/explain this Petri net and formulate/prove the Correctness Lemma for it (where var $a, b, x_1, ..., x_n, y_1, ..., y_n : \Sigma$)!

Explaining the Petri net encoding of word pair generation



Idea: transition q produces at B one mark for $a \neq b$, at C $n-1$ marks for other letters. To iterate store n and after completion delete vw.

Conclusion: three major sources of PN inadequacy

- insufficient abstraction and introduction of irrelevant technicalities
  - implementation details resulting from low-level token-based view

  ASM states can be tailored to any level of abstraction.
- lack of component structure to separate different concerns
  - mainly due to global overall process view

  Main program of an ASM can call any components.
- complexity of graphical layout indicating
  - a too great esteem of the graphical 'nature' of PNs as a help to understand/define them
  - lack of appropriate combination of visual/textual description elements (control versus data)

  Control state ASMs integrate flowchart representation of control flow with component structure and (textual definition of) data flow
  - including *dynamic set of agents with changeable program/context*

References

- E. Börger: Modeling distributed algorithms by ASMs compared to Petri Nets. Proc. ABZ'2016 (Springer LNCS 9675, pg. 3-34)

- Nancy A. Lynch, *Distributed Algorithms*. Morgan Kaufmann, San Franciso 1996. See also material for course 6.852 at MIT, Fall 2013.
- W.Reisig, *Elements of Distributed Algorithms*. Springer 1998
- E. Börger and K.-D.Schewe: *Concurrent Abstract State Machines*. – Acta Informatica 53 (2016), 469-492

- E. Börger and A. Raschke: Modeling Companion for Software Practitioners. Springer 2018
  http://modelingbook.informatik.uni-ulm.de

Copyright Notice