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
Assuming abstract communication medium

Given a set of *Processes* each equipped with its own *mailbox*
- usually assumed to be initially empty (and often renamed)

and the following process actions/predicates:

- **Send** \((msg, p)\)  // *deliver* \(msg\) *to mailbox of* \(p\) *triggers (agent of)* communication medium to **Insert** \((msg, mailbox_p)\)

- **Received** \((msg) = (msg \in mailbox_{self})\)  // *msg has been delivered*

- **Consume** \((msg) = \text{Delete}(msg, mailbox_{self})\)

Abstract view: **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\(_q\)*
For evaluation fairness we base the comparison on:

- W. Reisig, Elements of Distributed Algorithms (1998) 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. to make the comparison more meaningful

- Nancy A. Lynch, Distributed Algorithms (1996) for requirements (where not taken directly from Reisig’s book)
The declared goal of model verification

Goal of modeling resp. verification of model properties:

“retaining intuitive clarity and formal simplicity” (p.VII) resp.

“to 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
Requirement: Design and verify an algorithm such that

- over a finite *connected directed graph* \((Process, Edge)\)
- using *communication only bw Neighbors*

eventually every process knows the leader \(\max(\text{Process})\) (wrt linear order \(<\) of Processes) and the communication terminates.

Algorithmic idea: repeatedly each process \(p\) *alternates*

- to first **Propose** its current leader knowledge \(cand\) (greatest process seen so far, initially \(cand_p = p\)) **Send**ing it to its **Neighbors**
- 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

Source: W. Reisig, Elements of Distributed Algorithms. Fig.32.1
PROPPOSE = $\forall q \in \text{Neighb} \ \text{SEND}(\text{cand}, q)$

ProposalsImprove = $(\max(\text{Proposals}) > \text{cand})$

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

Each family of $(p, \text{FloodingLeadElect})$ forms a concurrent ASM
Comparison reveals Petri net idiosyncrasy (1)

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

- **forall** \( q \in \text{Neighb} \) **SEND**(cand, \( q \)) implemented by a transition which
  - 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** \( q \in \text{Neighb} \)’: instead of communication medium forwarding candidate msg \( y \) into neighbors’ local **mailboxes** 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
    • instead of describing them locally, separately for parameterized single processes, with implicit or explicit param (self or p)
    - Propose, CheckProposals, ImproveByProposals to not move x around for ≤ where only cand/msg y/z needed to keep models small (easy to understand, feasibly analysable)
  - **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 Neighborhood 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 encodings of data flow
  - alternating bw pending and updating explicitly visualized
  - only ‘indirect’ (implicit) visibility of (elementwise implementation of) checking Proposals compared to its direct, conceptually high-level (using max function) and explicit flowchart representation

NB. Diagrams considered as defining the algorithm:

“The hurried reader may just study the pictures” (Reisig p.V)

and generally the characteristics that “Petri nets are a graphical notion and at the same time a precise mathematical notion” are taken as “the most important properties” (Desel et al. 2001)
**Proposition**: In every concurrent run of *Processes*, each equipped with program *FloodingLeadElect*, if every enabled process will eventually make a move, eventually for every \( p \in \text{Process} \) holds:

- \( \text{cand} = \max(\text{Process}) \) (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 \( \text{diff}(\max(\text{Process}), \text{cand}) \) until this sum becomes 0

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

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. (Reisig 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: 2/3-phase \textit{Master/Slave} protocol

\textbf{MASTER}

- \textbf{idle} \rightarrow \textbf{JobToAssign} \rightarrow \textbf{ENQUIRE} \rightarrow \textbf{Answers Arrived} \rightarrow \textbf{ORDERJOB ORCANCEL} \rightarrow \textbf{idle}

\textbf{SLAVE}

- \textbf{idle} \rightarrow \textbf{Asked} \rightarrow \textbf{ANSWER} \rightarrow \textbf{UponJobArrival} \rightarrow \textbf{busy}

- \textbf{idle} \rightarrow \textbf{UponCancelMsg} \rightarrow \textbf{waiting ForJob}
From English to ASM: Master/Slave states (signature)

- **master**: a distinguished agent with locations:
  - \( job \in \text{Job} \) for the job to assign to the slaves
  - \( \text{JobToAssign} \), a trigger predicate signalling that the master should start trying to assign the \( job \)
  - \( \text{mode} \in \{\text{idle}, \text{waitingForAnswer}\} \), initially \( \text{mode} = \text{idle} \)
  - program **MASTER**

- **Slaves**: a set of agents with locations:
  - \( \text{mode} \in \{\text{idle}, \text{waitingForJob}, \text{busy}\} \), initially \( \text{mode} = \text{idle} \)
  - program **SLAVE**
**From English to ASM: Master predicates and actions**

\[
\text{ENQUIRE} = \text{forall } s \in \text{Slave} \text{ SEND(enquire, s)}
\]

\[
\text{AnswersArrived} = \text{forall } s \in \text{Slave}
\]

\[
\text{Received}((\text{accept, s})) \text{ or } \text{Received}((\text{refuse, s}))
\]

\[
\text{ORDERJOBORCANCEL} =
\]

\[
\text{if} \ SomeSlaveRefused \text{ then forall } s \in \text{Slave} \text{ SEND(cancel, s)}
\]

\[
\text{else forall } s \in \text{Slave} \text{ SEND(job, s)}
\]

\[
\text{CLEANUP} /\!\!\!/ \text{ clean up work for next round}
\]

\[
\text{SomeSlaveRefused} = \text{forsome } s \in \text{Slave} \text{ Received}((\text{refuse, s}))
\]

\[
\text{CLEANUP} =
\]

\[
\text{mailbox} := \emptyset
\]

\[
\text{JobToAssign} := \text{false} /\!\!\!/ \text{ consume input trigger}
\]
\textit{Asked} = \textit{Received}(\textit{enquire})

\textbf{Answer} =

\begin{verbatim}
choose answer \in \{\textit{accept}, \textit{refuse}\}
\text{SEND}((\text{answer}, s), \text{master})
\text{CONSUME}(% enquire) // consume input msg
\end{verbatim}

\textit{UponCancelMsg} = \textit{Received}(\textit{cancel})
\textit{UponJobArrival} = \textit{Received}(\textit{job})
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 idiosyncrasy: complex graphical layout (even for small systems), extraneous to algorithmic structure, complicates understanding
Exl: global process view modeling of order cancellation

Global overall process model view of Reisig’s 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 $\text{master pending}$ to place $\text{answered slaves}$
      - To prepare its return to mode $\text{idle}$
    - The other processes (in $U - x$) to (via transition $d$ or $e$)
      - Prepare $\text{cancellations}$ (by walking to that place)
      - Inform the master to have answered (never mind what) by walking to place $\text{answered slave}$

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’ (Reisig p.119)
Master/slave Petri net verification: (1) rename places

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

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

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

inv4 and inv6 imply $F + G + J + H \leq U$. 
The nodes of this graph are justified on p.257-258 using some previously defined and justified ‘proof patterns’ (ibid., Sect.69)
Acknowledged Broadcast (role of ground models)

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 Neighbors

Algorithmic Idea
- distinguished initiator upon BroadcastTrigger will Broadcast an info msg to its Neighbors and then waitForAck msgs from them
- if a not yet informed non-initiator node ReceivedInfoFromSomeNeighbor it will PropagateInfoToNonParentNeighbors and then waitForAck
- once a non-initiator ReceivedAckFromAllChildren it in turn sends an AckToParentNeighbor node by which it had been informed
- initiator once it ReceivedAckFromAllChildren Terminates
From English to ASM: 2-phase \textit{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**

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

\texttt{Broadcast} =  // \textit{triggers building breadth-first spanning tree}
\texttt{forall } n \in \texttt{Neighb} \texttt{ SEND(infoFrom(initiator), n)} \texttt{\}

\texttt{Terminate} = \texttt{Empty(EchoMsg)}  // empty mailbox for next round

\texttt{ReceivedInfoFromSomeNeighb} =  
\texttt{forsome } p \in \texttt{Neighb} \texttt{ Received(infoFrom(p))}

\texttt{PropagateInfoToNonParentNeighb} =  // \textit{tree building step}
\texttt{choose } p \in \texttt{Neighb} \texttt{ with} \texttt{ Received(infoFrom(p))}
\texttt{forall } n \in \texttt{Neighb \setminus \{p\}} \texttt{ SEND(infoFrom(self), n)}
\texttt{parent} := p  // define receiver of later ackFrom msg
\texttt{InformAboutChildReln}(p)
From English to ASM: \textsc{Echo} response actions

\begin{align*}
\text{ReceivedAckFromAllChildren} &= \quad \text{// true at leaves} \\
\text{ChildKnowlIsComplete and forall } m \in \text{Children} \quad \\
&\quad \text{Received}(\text{ackFrom}(m)) \\
\text{Children}_n &= \{ m \in \text{Neighb}_n \mid \text{parent}(m) = n \} \\
\text{AckToParentNeighb} &= \quad \text{// pass notification along spanning tree} \\
&\quad \text{SEND}(\text{ackFrom}(\text{self}), \text{parent}) \\
&\text{parent} := \text{undef} \quad \text{EMPTY}(\text{EchoMsg}) \quad \text{// clear for next round} \\
\text{InformAboutChildReln}(p) &= \quad \\
&\quad \text{SEND}(\text{IamYourChild}(\text{self}), p) \\
&\quad \text{forall } q \in \text{Neighb} \setminus \{ p \} \quad \text{SEND}(\text{IamNotYourChild}(\text{self}), p) \\
\text{ChildKnowlIsComplete iff forall } n \in \text{Neighb} \quad \\
&\quad \text{Received}(\text{IamYourChild}(n)) \text{ or Received}(\text{IamNotYourChild}(n))
\end{align*}
Proposition

In every concurrent \textbf{Echo} run (where each enabled agent will eventually make a move):

- each time the \textit{initiator} performs a \texttt{Broadcast} of an \textit{infoFrom} msg it will eventually \texttt{Terminate} (termination)
- the \textit{initiator} will \texttt{Terminate} only after all other agents have \texttt{Received} that \textit{infoFrom} msg and have acknowledged this to their \texttt{parent} neighbor (correctness)

Proof. Follows by run induction from two lemmas.
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

Figure 33.3. Cyclic echo algorithm

NB. 2-page explanation of this encoding in Reisig 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 `rightNeighbor`,
    - a task `Transfer` msg to its `left neighbor` to balance their workloads
  - when `ReceivedTransfer` msg `t` from its `right neighbor` accepts the task `t` (if `t ≠ nothingToTransfer`) to balance their workloads.

so that eventually the workload difference of two nodes becomes ≤ 1
Figure 37.1. Distributed load balancing

\begin{align*}
\text{sort} & \quad \text{site} & \quad \text{var} & \quad i, j : \text{nat} \\
\text{sort} & \quad \text{alloc} = \text{site} \times \text{nat} & \quad \text{var} & \quad x, y : \text{site} \\
\text{const} & \quad U : \text{set of sites} & \forall x \in U \ \exists i \in \text{nat} : (x, i) \in V \\
\text{const} & \quad V : \text{set of alloc} & x \neq y \Rightarrow r(x) \neq r(y) \\
\text{fct} & \quad l, r : \text{site} \to \text{site} & \exists n \in \text{nat} : r^n(x) = x \\
& \forall x \forall y \exists n \in \text{nat} : y = r^n(x) \\
& l(r(x)) = x
\end{align*}
flowchart encoded in \textit{state i}-Petri-subnet \((i = 1, 2, 3)\)

- places \textit{workloadmessage/updatedmessage} are global (Sic) mailboxes
  - one mailbox for all processes
- transferred tasks themselves are not represented in the PN (Sic)
Process, a static set of agents, each equipped with

- static ring structure with leftNeighb, rightNeighb ∈ Process
- the current WorkLoad ⊆ 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 leftNeighb
      or task Transfer msgs (from the right neighbor)
    - i.e. transfer ∈ Task ∪ {nothingToTransfer}
- mode switching from initial value informRightNeighb to transferToLeftNeighb back to acceptFromRightNeighb
- program RingLoadBalance
ReceivedLeftNeighbLoad  iff

\[ \text{there is some } n \in \text{Nat} \cap \text{WorkLoadMsg} \]

\text{TRANSFER\_TASK\_TO\_LEFT\_NEIGHB} =

\text{let } \{ \text{leftNeighbLoad} \} = \text{WorkLoadMsg}

\text{if } \text{workLoad} > \text{leftNeighbLoad} // \text{there is a task to transfer}

\text{then choose } \text{task} \in \text{WorkLoad}

\text{SEND}(\text{task}, \text{leftNeighb})

\text{DELETE}(\text{task}, \text{WorkLoad})

\text{else } \text{SEND}(\text{nothingToTransfer}, \text{leftNeighb})

\text{CONSUME}(\text{leftNeighbLoad})
ReceivedTransfer iff

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

\[
\text{ACCEPT\_TASK} = \\
\text{let } \{\text{transfer}\} = \text{WorkLoadMsg} \\
\text{if } \text{transfer} \in \text{Task} \text{ then } \text{ADD}(\text{transfer}, \text{WorkLoad}) \\
\text{CONSUME}(\text{transfer}) \quad // \quad \text{msg removal from mailbox}
\]
**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, \( a = |\text{Process}|. \)

- **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 \textit{WorkLoad change}

\textbf{Requirement:} environment may trigger to \texttt{CHANGEWORKLOAD} by increasing or decreasing \textit{WorkLoad} by a set of tasks (Reisig 37.4)

\textbf{Modeling idea:} each process watches a trigger (monitored location) \texttt{workLoadChange} with values in \{\texttt{add}(T), \texttt{delete}(T), \texttt{noInput}\}

When triggered (by the env!) the process will \texttt{CHANGEWORKLOAD}, otherwise it executes \texttt{RINGLOADBALANCE} (as before)

\textbf{From English to ASM:}

\begin{align*}
\textit{WorkLoadChange} \iff \texttt{workLoadChange} \in \{\texttt{add}(T), \texttt{delete}(T)\} \\
\texttt{CHANGEWORKLOAD} = \\
\quad \text{if} \ \texttt{workLoadChange} = \texttt{add}(T) \ \text{then} \ \texttt{ADD}(T, \textit{WorkLoad}) \\
\quad \text{if} \ \texttt{workLoadChange} = \texttt{delete}(T) \ \text{then} \ \texttt{DELETE}(T, \textit{WorkLoad}) \\
\quad \texttt{CONSUME} (\texttt{workLoadChange}) \ // \ \text{input consumption}
\end{align*}
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

Petri net idiosyncrasy:

external env triggers modeled as internal transition

Copyright © Egon Börger, Dipartimento di Informatica, Università di Pisa, Pisa, Italy.
Abuse of nondeterminism

- the lack of component structure
  - 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.’ (Reisig, 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.
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 $\text{requestFrom}(n)$, $\text{replyFrom}(n)$ where $n \in \text{Node}$, mailbox initially empty
  - phases $\text{mode} \in \{\text{start}, \text{waitForOk}, \text{agreed}\}$, initially $\text{mode} = \text{start}$
  - 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:

\[
\text{replyFrom}(n, \text{requestFrom}(m))
\]

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 ∈ Neighb SEND(requestFrom(self), n)

ReInitializeReplies

ReInitializeReplies =
  forall n ∈ Neighb CONSUME(replyFrom(n))

AllNeighbAccept = forall n ∈ Neighb Received(replyFrom(n))
\[\text{ReceivedRequest} = \]
\[\text{forsome } n \in \text{Neighb} \ \text{Received(requestFrom}(n))\]
\[\text{Answer} = \text{// better specify using } \text{forall} \text{ instead of} \text{ choose}\]
\[\text{choose } n \in \text{Neighb with} \ \text{Received(requestFrom}(n))\]
\[\text{SEND(replyFrom}(\text{self}), n)\]
\[\text{CONSUME(requestFrom}(n))\]
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

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

That’s all one needs to capture the requirements change!
‘Advanced Consensus’ Petri net (with quiet/demanded sites)

Petri net idiosyncrasy: token encoding of agent attributes requires new places & structural diagram changes to restrict/add guards/actions.
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 with proof (with additional complexity due to Petri net technicalities) in Reisig 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 \( \text{currRound} + 1 \) *sender* will **StartNxtFileTransfer** until the sequence is transferred.

- When sending file \( F(\text{round}) \) a sync bit \( \text{round} \mod 2 \) is
  - attached to file msgs \( (F(\text{round}), \text{round} \mod 2) \)
  - extracted and resent by the receiver as acknowledgment msg
  - checked upon **ReceivedMsg** by *sender/receiver* for Matching own sync bit and in case of matching is flipped for next \( \text{round} + 1 \)
From English to ASM: \textsc{AltBit} protocol
From English to ASM: Alternating Bit state

- **mailbox** `MsgQueue`: queue reflects no-msg-overtaking assumption
  - `FileMsgs` \( (F(i), i \mod 2) \) at receiver, `AckMsgs` \( \in \{0, 1\} \) at sender
  - projection fcts to extract \( \text{file}((f, b)) = f \) and `syncBit`
    \[ \text{syncBit}((\text{file}, b)) = b, \text{syncBit}(\text{bool}) = \text{bool} \]

- **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:
    \[ \text{currRound}(\text{sender}) \leq \text{currRound}(\text{receiver}) \]
  - derived synchronization bit `currRound \mod 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

\textbf{StartNextFileTransfer} =
\begin{align*}
\text{Send}((\text{nextFile}, \text{nextSyncBit}), \text{receiver})
\end{align*}

\textbf{IncreaseRound}
\begin{align*}
\text{where} \quad \text{nextFile} &= F(\text{currRound} + 1) \\
\text{nextSyncBit} &= \text{currRound} + 1 \text{ mod } 2 // \text{flipped sync bit}
\end{align*}

\textbf{ReSendFile} =
\begin{align*}
\text{Send}((F(\text{currRound}), \text{currRound} \text{ mod } 2), \text{receiver})
\end{align*}

\textbf{ReceivedMsg} = \text{iff} \quad \text{MsgQueue} \neq [ ] // \text{mailbox not empty}

\textbf{Match} \quad \text{iff} \quad \text{syncBit} (\text{currMsg}) = \text{currRound} \text{ mod } 2

\textbf{CloseRound} = \text{Consume} (\text{currMsg})
\begin{align*}
\text{where} \quad \text{currMsg} &= \text{head} (\text{MsgQueue})
\end{align*}

\textbf{IncreaseRound} = (\text{currRound} := \text{currRound} + 1)
**Receive&Ack =**

**StoreFile**

**SendAck**

**Consume(currMsg)**

**ResendAck =**  // NB. receiver is round-ahead of sender

**Send(flip(currRound \ mod \ 2), sender)**  // previous sync bit

where

**SendAck = Send(syncBit(currMsg))**

**StoreFile = (G(currRound) := file(currMsg))**

NB. Assume *timeout* to be initialized such that **ResendAck** can happen only after the first **Receive&Ack** (e.g. by initializing *timer = ∞*).
Correctness of $\text{AltBit}$ runs (termination with $F = G$) is easily proved by induction on $\text{AltBit}$ run phases:

We found no proof for the AltBit Petri net.
Figure 27.7. The alternating bit protocol
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 \( i \mod 2 \)

in a window \([low, high]\) between \( low \) and \( high \) s.t.

- initially \( low = 1, high = 0 \) at sender and receiver

- sender can perform \texttt{STARTNEXTFILETRANSFER} and \texttt{INCREASEWINDOW} by new syncBit \( high := high + 1 \) as long as not \texttt{FULLWINDOW}

- if \( Acknowledged(low) \) (read: upon ack receipt of file with index \( low \)) sender will \texttt{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 \texttt{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 $\text{FullWindow} = \text{high} - \text{low} + 1 = \text{maxWinSize}$) and refined by
  
  $\text{nextFile} = F(\text{high} + 1)$, $\text{nextSyncBit} = \text{high} + 1$

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

- **ResendFile** = Send(($F(\text{low}),\text{low}$), receiver)

- **Match** = ($\text{low} \leq \text{syncBit}(\text{currMsg}) \leq \text{high}$) // syncBit in window

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

  $\text{Acknowledged}(\text{syncBit}(\text{currMsg})) := \text{true}$

initially $\text{Acknowledged}(i) = \text{false}$ 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 \( \text{syncBit}(\text{currMsg}) > \text{high} \) cannot be **Consumed**: the receiver must first **SlideWindow** to let **currMsg** **Match**

\[
\text{SlideWindow} = \text{let } (s = \text{syncBit}(\text{currMsg})) \text{ in }
\]
\[
\text{high} := s
\]
\[
\text{low} := \max\{1, s - \text{maxWinSize} + 1\}
\]

**NB.** When receiving a new file with **FullWindow**, the ack for **low** must have been received by the **sender** so that **low** can be shifted to the right.

- **ReSendAck** = **SEND**(low, sender)

**NB.** Finitely many indices suffice using \( +1 \mod r \) for updating **low**, **high** for a sufficiently large \( r \) depending on **maxWinSize** if there is nomsgt 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 in Reisig op.cit.)
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 ($E$)

\[ R \rightarrow T \rightarrow C \rightarrow E \rightarrow R \]

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 \(\text{FETCH} \ stickAt(level)\)
    - later release it (by being \(\text{FETCHed}\) by another interested process) in case there is another interested process
  - local shared loc \(flag \in \{0, \ldots, n - 1\}\) at each process, initially \(flag = 0\), writable by process \(p\) and readable by all other processes
    - \(flag_p = l > 0\) 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**

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

**FetchStick** =  // for n = 2, one more guard below for n > 2
if (not HasStick\_level) then stick\_At\_level := self
  // NB. This means skip if HasStick\_level
where HasStick\_level iff stick\_At\_level = self

Winner = NobodyElseInterested or
MeantimeSomebodyElseFetchedStick
NobodyElseInterested(level) = forall p ≠ self  flag_p < level
// for case n = 2 this means flag_{theOtherProcess} = 0

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

MeantimeSomebodyElseFetchedStick(level) iff stickAt_{level} ≠ self
// meaning in case n = 2 that stickAt_{level} = theOtherProcess
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 **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 flags 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).

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 $\text{MutexPeterson}_n$ for $n > 2$

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

- for each level value there is a least one loser
  - a process that has to $\text{waitToWin}$ until $\text{NobodyElseInterested}$
- 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 $\text{FetchStick}$ guaranteeing that

- in each step at most one process $p$ can write $\text{stickAt}$

\[
\text{chosenWriterFor}(\text{stickAt}_{level}) = \text{self}
\]

where $\text{chosenWriterFor}(\text{stickAt}(l)) = \text{select}\{p \mid \text{flag}_p = l \text{ and } \text{mode}_p = \text{getStick} \text{ and } \text{stickAt}(l) \neq p}\}$
MutexPeterson$_n$ for $n > 2$

CompetitionFinished iff level = $n - 1$

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

Reset$(\text{level}) = (\text{level} := 1)$
Conclusion: three major sources of PN inadequacy

- insufficient abstraction and introduction of irrelevant technicalities (implementation details) resulting from the low-level token-based view
- lack of component structure and separation of concerns
  - mainly due to global overall process view
- 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

BUT one can view each Petri net as a concurrent ASM where each transition $t$ is executed by one associated agent with rule

\[
\text{if } \text{Enabled}(t) \text{ then } \text{FIRE}(t)
\]
ASM nets use (most) **general notion of state** supporting:

- transitions read/write directly **values of any type** in locs ('pre/post-places')
  - uniform combination of control, data, resources avoiding particular (e.g. token-based) encodings of abstract objects
  - in the quantifier-free case finitely many locations
    - inherit/generalize pre-/post place visual display of transitions
- **dynamic set of agents with changeable program/context** etc.
- **communication** mechanism bw multiple agents
  - via input (monitored) and output locations (or a mailbox)
ASM nets offer (most) **general computational power** (transition step)

- **transformation of objects** of whatever complexity as needed for a given application (not only control flow centered token manipulation)
  - avoiding restrictions due to decidability of Petri net reachability problem (e.g. no 0-test, no general recursion, no quantifiers)
- constructs for **synchronous parallelism** (**forall**) and **selection** (**choose**)
- **global behavior view** where needed
  - avoiding difficulties to describe **non-local** behavior by Petri nets, e.g. OR-join or cancellation in BPs
- **explicit scheduling** (including prioritization schemes)
- **truly concurrent runs**, not only partial order, interleaving or non-deterministic runs
  - avoiding implicit 1-core interleaving assumption in Petri nets (‘no 2 transitions fire simultaneously’)

---

Copyright © Egon Börger, Dipartimento di Informatica, Università di Pisa, Pisa, Italy.
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 $n, i \in \text{Nat}$ 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 $\in A^*$ | $v \neq w$ and $|v| = |w|$\}. 
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 \( n, i, j : \text{Nat} \))?
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 \( \text{var } a, b, x_1, \ldots, x_n, y_1, \ldots, y_n : \Sigma \))!
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 $vw$. 
Concurrent ASM nets enhance Petri nets (3)

supporting

- practical refinement method for
  - componentwise design (horizontal refinement)
  - separating different design decisions (vertical refinement)
    - e.g. atomic from durative actions

- e.g. parallel (not-ordered) execution from implementation by stepwise (sequentialized) execution

- modularity by
  - parameterizable locations and components
  - FSM-phase based system structuring and visualization
References

- E. Börger: Modeling distributed algorithms by Abstract State Machines compared to Petri Nets. Proc.ABZ’2016 (Springer LNCS)

- Nancy A. Lynch, *Distributed Algorithms*. 
  - Morgan Kaufmann, San Francisco 1996
  See also material for course 6.852 at MIT, Fall 2013

- W. Reisig, *Elements of Distributed Algorithms*
  - Springer-Verlag 1998


- E. Börger and K.-D. Schewe: *Concurrent ASMs*.
  - Acta Informatica 2015