Modeling distributed algorithms by Abstract State Machines compared to Petri Nets

Egon Börger[1]

Università di Pisa, Dipartimento di Informatica, I-56125 Pisa, Italy
boerger@di.unipi.it

Abstract. We show how to model distributed algorithms by Abstract State Machines (ASMs). Comparing these models with Petri nets (PNs) reveals a certain number of idiosyncrasies of PNs which complicate both model design and analysis. The ASMs we define illustrate how one can avoid such framework related technicalities.[1]

1 Introduction

This paper is about modeling of distributed algorithms and property verification using ASMs [9]. Since among theoretical computer scientists PNs seem to be considered suitable for distributed algorithms we compare the ASMs with PNs.

There is a variety of PNs with different semantics, expressiveness and complexity, depending on the incorporated features, e.g. priorities, time, colours, stochastic, continuous or hybrid discrete-continuous features, etc., see [25]. To make a concrete comparison feasable we stick to the PNs defined in [28] to "provide the expressive power necessary to model elementary distributed algorithms adequately, retaining intuitive clarity and formal simplicity" (ibid. p.VII).

To avoid any bias in selecting the comparison examples we follow the author of [28], who is considered an authority in the field, for the "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" [28, p.V]. We focus on the 'Advanced System Models' in [28, Parts B,D]; that they range among the simpler ones in the standard text book [22] should not diminish the relevance of a comparative analysis.

We discovered that the proposed PNs, compared to the ASMs shown below, are neither 'intuitively clear' nor 'formally simple' but hide the underlying intuition under various technicalities of the low-level token-based modeling approach and as a consequence make the mathematical analysis of model properties more complicated than necessary. We hope that the concrete comparison between PNs and ASMs in this paper will help the practitioner to see how ASMs allow one

[1] This work was partially supported by the European Commission funded project BIOMICS, Grant no. 318202.

to efficiently design and analyse distributed algorithms without being detracted by extraneous technicalities of the underlying modeling framework.

Caveat. I learnt PNs 50 years ago when Dieter Rödding at the Institute for Mathematical Logic and Foundational Studies in Münster started their systematic investigation and elaboration in his seminars on the theory of automata and networks, seminars which became a regular event at the universities of Münster, Dortmund and Paderborn until Rödding's premature death in 1984. I also heard Petri explain his ideas in person. My (at that time a logician's) interest in PNs was biased by their challenging mathematical theory, in particular by in the 60'ies/70'ies open expressiveness and decidability questions, but also by their use to study the semantics of concurrency (see [27] for a good textbook). Only in 1990 when I started systematic experiments to model computational systems with ASMs I looked at PNs through a practitioner's eyes, as I do here, namely to figure out how practical the language is for modeling distributed systems in combination with appropriate property verification methods.

2 Network Algorithms

In this section we investigate some distributed network algorithms for which one finds in [28] (carefully layed out!) PN formalizations we behaviorally compare with ASMs. For an unbiased statement of the requirements, unless we take them directly from [28] we resort to the problem descriptions in the standard textbook [22]. This allows the reader to evaluate to which degree the two frameworks support capturing requirements accurately, in a way that can be 'justified' and 'checked'(for epistemological reasons not mathematically verified!) to be 'correct' (*ground model problem* [3]), and to document design decisions in a transparent, easily accessible way [23]. To be able to also include visualization aspects into the behavioral comparison we define the ASM models using the traditional flowchart representation for control-state ASMs, an extension of FSMs with well-known meaning (precisely defined in [9, Fig.2.5]). The phase (also called mode) structure of FSMs offers componentwise[2] definitions and to separate visualized control-flow (phase structure) elements from (better textually described) communication/data/resource-related predicates and actions. *mode* can also be interpreted as a flag so that for ASM rules a guard with $mode = wait$ (as used below) does not necessarily imply busy waiting.

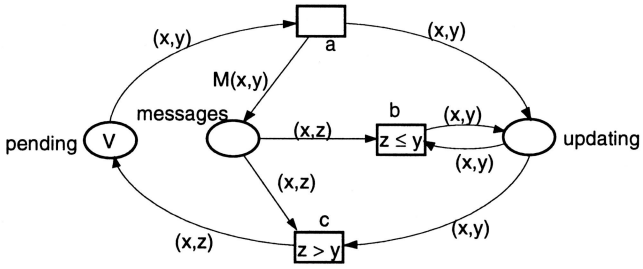
Many of the algorithms below have as background structure finite directed graphs $(Process, Edge)$ of agents each executing some program using (an abstract form of unless otherwise stated reliable) communication among *Neighb*ors where $Neighb_p = \{q \mid (p, q) \in Edge\}$ (outgoing neighbors). We use the following abstract operations on each agent's (initially empty) *mailbox*:

SEND(msg, p) with effect to (eventually) INSERT$(msg, mailbox_p)$
Received(msg) iff $msg \in mailbox$
CONSUME$(msg) =$ DELETE$(msg, mailbox)$

[2] See the use of modes in [23] as a means to structure the set of states.

We state explicitly if the graph is undirected or *mailbox* is considered to be a queue instead of a set (with corresponding refinement of its operations).

2.1 Leader Election in Connected Graphs



sort site
sort state : site × site
const U : set of sites
const V,W : set of states
≤ : total order on U
fct M : state → set of states

var x, y, z : site
x,y ∈ U → x W* y
$W_1 \cup W_2 = U$
V = {(u,u) | u ∈ U}
M(x,y) = W(x) × {y}

Fig. 1. Basic leader election Petri net ([28, Fig.32.1])

For a directed connected graph of linearly ordered *Process*es p design and verify a distributed algorithm which uses communication only among *Neighb*or processes. Each p starts considering itself as a leader *cand*idate ($cand_p = p$), PROPOSEs *cand* to its *Neighb*ors and checks it against its *Neighb*ors' proposals s.t.:

Leader Election Property ([22, 15.2]): In every asynchronous run of processes each 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 wrt their order $<$)
- mailbox $Proposals = \emptyset$ (there is no more communication)

Before proceeding to a detailed comparison of Fig. 1 and Fig. 2 we invite the reader to grasp an understanding of the two diagrams (noticing the effort and time needed to comprehend each model) and to compare the definitions in Fig. 1 with the textual definitions of the actions and predicates occuring in Fig. 2: they mark the transition from a description in natural language to a mathematically rigorous model (a *ground model* in the sense described in [3,5]) and for this reason must be understandable and checkable by domain experts to 'correctly' capture the intended intuitive meaning.

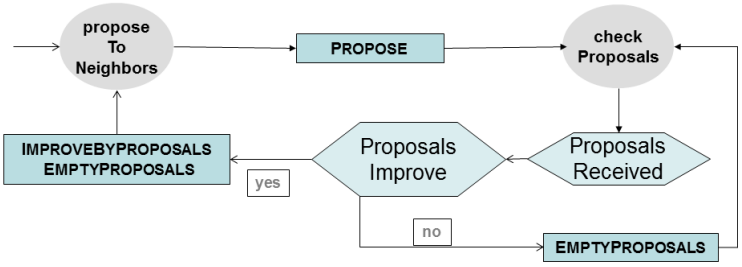


Fig. 2. 2-phase FLOODINGLEADELECT ASM

PROPOSE = **forall** $q \in Neighb$ SEND$(cand, q)$
$ProposalsImprove = (max(Proposals) > cand)$
IMPROVEBYPROPOSALS = $(cand := max(Proposals))$

We agree that one purpose of model verification is "to make intuitive statements and conclusions transparent and precise, this way deepening the reader's insight into the functioning of systems" [28, p.143]. With this in mind we invite the reader to compare the technically involved (formalistic, lengthy and hard to follow) 'proof graph'-based PN verification of Fig. 1 in [28, p.258-260] with proving the Leader Election Property for Fig. 2 by a step-properties-based, intuition-guided induction on ASM runs and on the sum of the differences $diff(max(Process), cand)$ until this sum becomes 0.[3] The induction progresses each time some p PROPOSEs its $cand_p$ to a neighbor q with smaller $cand_q$, so that next time that neighbor checks whether its $ProposalsImprove$, $cand_q$ increases (wrt $<$) yielding a decrease of $diff(max(Process), cand_q)$.

Idiosyncrasy 1. *Low-level token-based encoding* turns objects and executing agents indistinctly into (though abstract) tokens and actions into token manipulations, enforcing to carry agents around (often together with some of their attributes) to wherever they must perform an action. This complicates to grasp (via decoding) the intuitive behavioral meaning of actions of single agents.

For example to SEND the current leader idea *cand* to all *Neighb*ors (the PROPOSE action in Fig. 2) is implemented in Fig. 1 by a transition which

- deletes a token (x, y)—encoding a process x with leader candidate y—from place *pending* and adds it to place *updating*
  - This encodes the *mode* update for x from *send* (*proposeToNeighbors*) to *receive* (*checkProposals*): changing place x must drag along also y!
- 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 the communication medium forwarding the *cand* msg y into each neighbors'

[3] $diff(p, q) = orderNumber(p) - orderNumber(q)$ for processes numbered $1, 2, \ldots$, in increasing order.

mailbox it is x which moves around all its neighbors (coupled with y) for further processing at place *messages* (a global mailbox of all processes)!

Correspondingly the initialization condition that *cand* = **self** for each process is encoded by the token set V in place *pending*.

See Sect. 2.5 for a structural diagram change due to the token encoding of adding an agent attribute to satisfy a small requirements change.

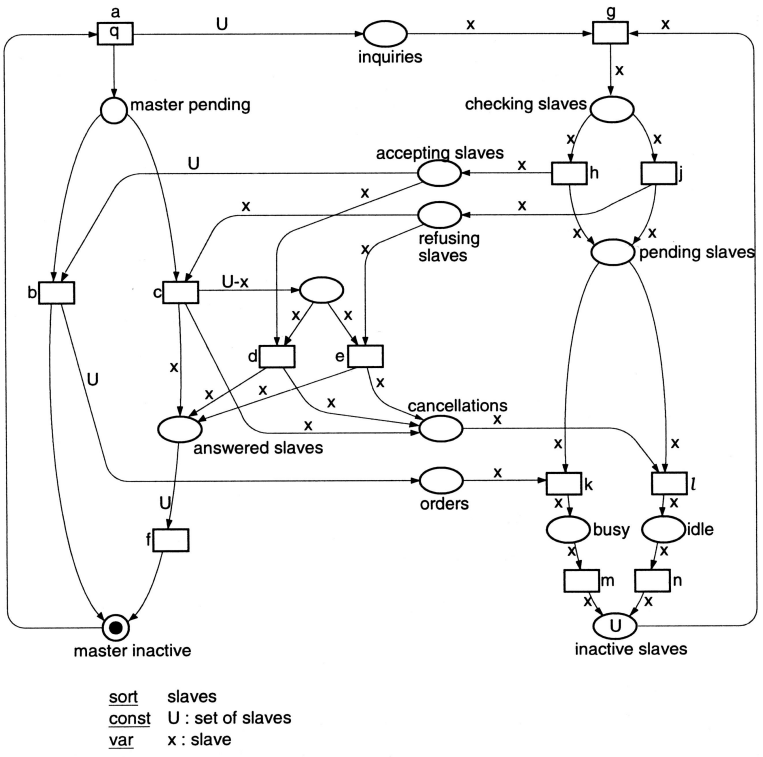


Figure 30.1. Distributed master/slave agreement

Idiosyncrasy 2. The *global overall process view*[4] obstructs the separation of concerns, obfuscates the architectural system view (here the structure of communicating agents) and burdens the net layout with background elements one better deals with in the background.

For example in Fig. 1 the *Neighb*orhood background structure is coded into the diagram together with the dynamic control structure instead of dealing with

[4] Not to be confused with the issue of a global vs local state view.

(frequently static) background elements separately using standard mathematical means. In *one* PN the initialization, actions and communications of each process are defined[5] instead of describing them locally as executed by agents with precise component interfaces, in component-based and stepwise refinable fashion. For ex. the components PROPOSE, CHECKPROPOSALS, IMPROVEBYPROPOSALS in Fig. 2 are defined with implicit (or where needed explicit **self** or p) parameter instead of carrying everywhere x explicitly around as done in Fig. 1 for the $\leq$ / $<$ test though only *cand* and message values y, z are needed.

Idiosyncrasy 3. The *visualization is helpful mostly for control flow* but must resort to less clear encodings to describe underlying data flow.[6]

For example, in Fig. 1 the alternation between *pending* and *updating* is explicitly visualized, but the equally important checking of *Proposals* is 'visible' only indirectly via its elementwise implementation, in contrast to the direct and explicit graphical representation of its conceptual meaning (applying the 'high-level' *max* function to the entire mailbox *Proposals*) in the flowchart of Fig. 2.

2.2 Master/Slave Agreement

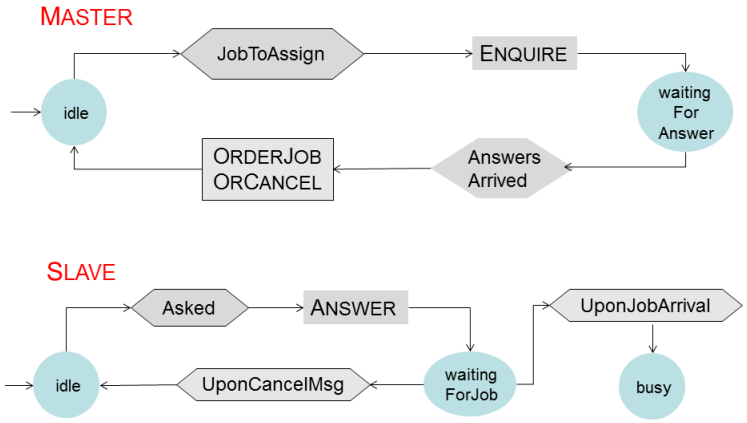


Fig. 3. 2/3-phase MASTER/SLAVE ASM programs

In this four-phase protocol a *master* ENQUIREs about a *JobToAssign* to slaves [28, p.119]; they will execute it if they all ANSWERed to accept the job which

[5] The problem to separate business from coordination logic triggered a similar observation in [18, p.133] that in a PN "every component must be modeled explicitly".

[6] One would expect that using colored tokens [16] may help, but the tokens in [28] are already of a most general nature, namely elements of abstract domains (represented by first-order logical terms) which comprise colored tokens.

otherwise is canceled: a) *master* sends an inquiry to all slaves and waits for their answer; b) slaves ANSWER to accept or refuse; c) when all *AnswersArrived* *master* will ORDERJOB execution (in case all accepted) ORCANCEL his request (otherwise); d) slaves go busy (*UponJobArrival*) or return idle (*UponCancelMsg*). Thus eventually *master* returns *idle*, with each slave *idle* or each slave *busy*.

Compare Fig.30.1 (copied from [28]) with Fig. 3 and its predicate/action definitions. *JobToAssign* describes the event that triggers a protocol round.

ENQUIRE = **forall** $s \in Slave$ SEND$(enquire, s)$
AnswersArrived = **forall** $s \in Slave$
  Received$((accept, s))$ **or** *Received*$((refuse, s))$
ORDERJOBORCANCEL =
  if *SomeSlaveRefused* **then forall** $s \in Slave$ SEND$(cancel, s)$
    else forall $s \in Slave$ SEND(job, s)
  CLEANUP // clean up work for next round
SomeSlaveRefused = **forsome** $s \in Slave$ *Received*$((refuse, s))$
CLEANUP = $\begin{cases} mailbox := \emptyset \\ JobToAssign := false \ // \text{ consume input event trigger} \end{cases}$

$Asked = Received(enquire)$
ANSWER = $\begin{cases} \textbf{choose } answer \in \{accept, refuse\} \text{ SEND}((answer, s), master) \\ \text{CONSUME}(enquire) \ // \text{ consume input msg} \end{cases}$
$UponCancelMsg = Received(cancel), \quad UponJobArrival = Received(job)$

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 $F + G + J + H \leq U$.

The surprisingly involved master/slave PN in Fig.30.1 compared to the simple MASTERSLAVE ASMs in Fig. 3 illustrates once more the PN idiosyncrasies explained in Sect. 2.1 and adds a consequence of Idiosyncrasy 1.

Idiosyncrasy 3b. The *complexity of the graphical layout to define PNs, in particular where unrelated to the algorithmic problem structure*, complicates both the understanding of the model[7] and its verification.

[7] We disregard here the peculiar (global overall process view triggered?) design decision in Fig.30.1 where the slaves organize the refusal case among themselves, trig-

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

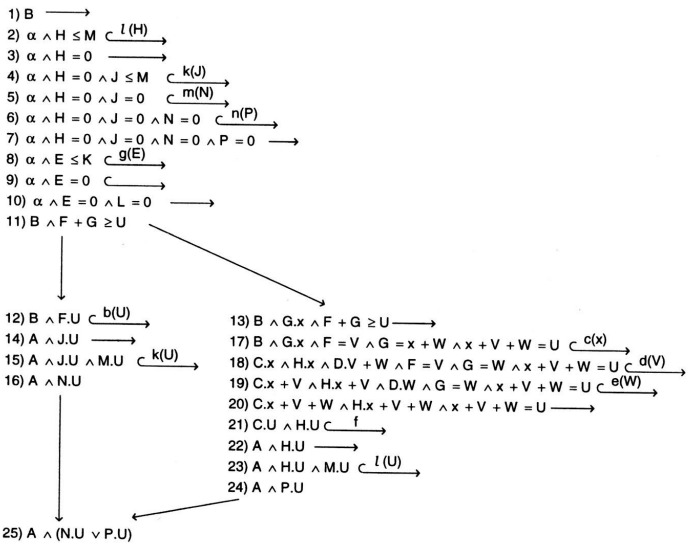


Figure 75.2. Proof graph for (1), with shorthand (3)

The verification of Fig.30.1 in [28, p.255-257] is to be compared with an induction on concurrent ASM runs of MasterSlave showing that if the master starts to Enquire, then (assuming that every enabled agent will eventually make a move) eventually the master becomes idle and either all slaves become idle too or all slaves become busy. In [28, p.255] the correctness property appears as:

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

and is proved using the proof graph of Fig. 75.2 which in turn is based upon 6 invariants (Sect.75.2 State properties, both copied from [28]) and is justified on p. 256-258 using some net-representation-driven (syntactical) 'proof patterns'. This illustrates one of the author's goal, namely to reach "a maximally tight combination of modeling and analysis techniques" [28, p.2]. It is the reader to judge whether such net-structure-driven verifications of properties which too are derived from the static net structure come up to the claim "to make intuitive statements and conclusions transparent and precise, this way deepening the reader's insight into the functioning of systems" [28, p.143].

---

gering (via c) the master to eventually return to idle (via f), without further master involvement: no slave 'reports ... refusal to the master' and never 'the master sends a cancellation to each slave', contrary to the formulated requirements [28, p.30].

2.3 Acknowledged Broadcast (Echo algorithm)

Here we illustrate the lack of support by PNs, mentioned in Sect. 2.2, to directly capture via their graphical layout the underlying algorithmic intuitions, which makes it difficult to build satisfactory *ground models* domain experts can understand, justify and check to correctly reflect the requirements ([5]). We use an algorithm which guarantees an initiator's message being broadcast (building a spanning tree) and acknowledged (echoed) through a connected, for bi-directional communication undirected graph of processes, using only communication between *Neighb*ours. The algorithmic idea ([22, 4.2.2]) is that

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

The encoding scheme for the PN solution in Fig.33.3 (copied from [28]) is explained there on two pages [p.127-129]. In contrast, the ASM programs in Fig. 4 directly reflect the algorithmic idea: the upper program lines (from left to right) describe building a spanning tree, the lower lines (from right to left) the notification of completion from leafs back to the *initiator*.

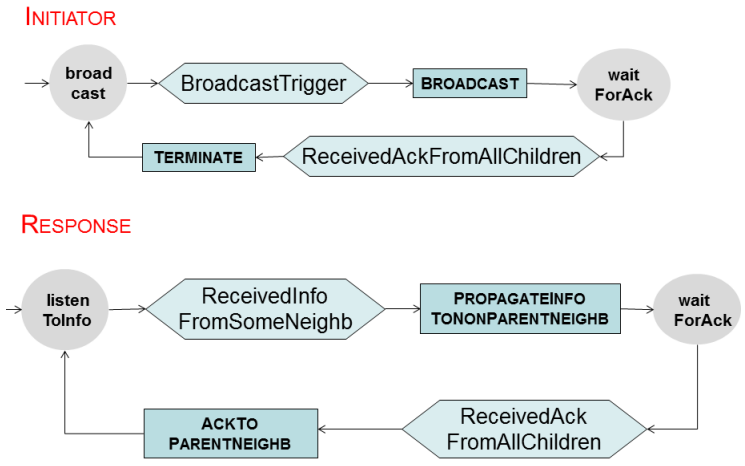


Fig. 4. 2-phase ECHO ASMs

In Fig. 4 event *BroadcastTrigger* triggers the *initiator* to start.

BROADCAST = **forall** $n \in Neighb$ SEND($infoFrom(initiator), n$)
TERMINATE = EMPTY($mailbox$) // clear for next round

Response actions capture spanning tree construction and navigation:

ReceivedInfoFromSomeNeighb =
  forsome $p \in$ *Neighb Received*(*infoFrom*(p))
PROPAGATEINFOTONONPARENTNEIGHB = // *tree building step*
  choose $p \in$ *Neighb* **with** *Received*(*infoFrom*(p))
    forall $n \in Neighb \setminus \{p\}$ SEND(*infoFrom*(**self**), n)
    parent := p // define receiver of later ackFrom msg
    INFORMABOUTCHILDRELATION(p)
ReceivedAckFromAllChildren = // true at leafs
  ChildKnowIsComplete **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*(**self**), *parent*)
  parent := **undef** EMPTY(*mailbox*) // clear for next round
INFORMABOUTCHILDRELATION(p) =
  SEND(*IamYourChild*(**self**), p)
  forall $q \in Neigh \setminus \{p\}$ SEND(*IamNotYourChild*(**self**), p)
ChildKnowIsComplete iff **forall** $n \in Neighb$
  Received(*IamYourChild*(n)) **or** *Received*(*IamNotYourChild*(n))

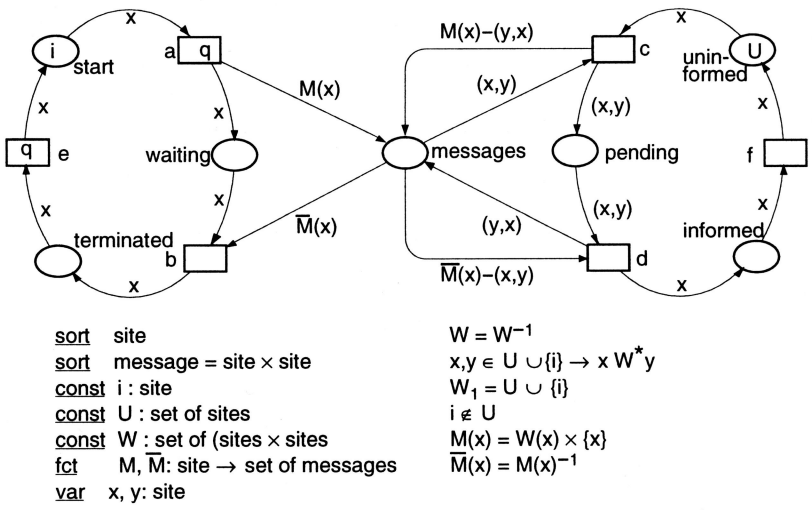


Figure 33.3. Cyclic echo algorithm

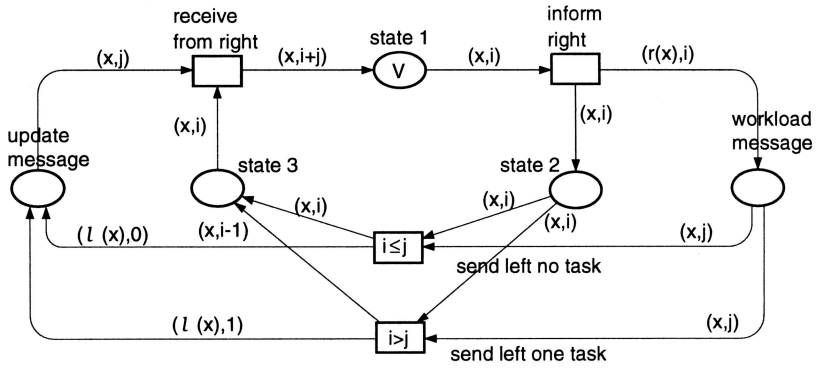
Comparing the verification of Fig.33.3 and Fig. 4 illustrates once more the heavy burden the graphical PN complexity can put on the verification effort.

The correctness proof for Fig.33.3 in [28, p.260-266] is 6 pages long and hides the intuition underlying the fundamental spanning tree method to a large extent in the PN 'proof graphs'. In contrast one can show by induction on concurrent ECHO runs (where each enabled agent will eventually make a move):

- **Lemma 1**. Each PROPAGATEINFOTONONPARENTNEIGHBORS increases in the tree of agents *waitingForAck* the distance to the *initiator* until leafs are reached. (Proof by downward induction.)
- **Lemma 2**. When 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.)

The two lemmas imply that each time the *initiator* performs a BROADCAST of an *infoFrom* message it will eventually TERMINATE (termination), but only after all other agents have *Received* that *infoFrom* msg and have acknowledged this to their *parent* neighbor by an *ackFrom* msg (correctness). (See [22, 15.3])

2.4 Load Balancing in Rings



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 ∃ 1i∈ 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

Here the goal is to balance the *workLoad* (number of *Task*s to be executed by a process) among a fixed set of (say at least 3) processes in a given ring using

communication only with *leftNeighb*or and *rightNeighb*or, assuming a fixed total workload. The algorithmic idea (see [28, p.140]) is that every process sends

- a *LeftNeighbLoad* message (i.e. its *workLoad*) to its *rightNeighb*or,
- a task *Transfer* message to its *leftNeighb*or to balance their workloads,

and when *ReceivedTransfer* message, i.e. some $t \in Task \cup \{nothingToTransfer\}$ from its *rightNeighb*, accepts the task to balance their workloads (unless $transfer = nothingToTransfer$).

This 3-phase protocol is directly expressed by Fig. 5 with its (straightforward) action/predicate definitions. In Fig.37.1 (copied from [28]) the control flow is encoded in the subgraph of places *state* i ($i = 1, 2, 3$); the other two places are two global (!) mailboxes (each one used by all processes).

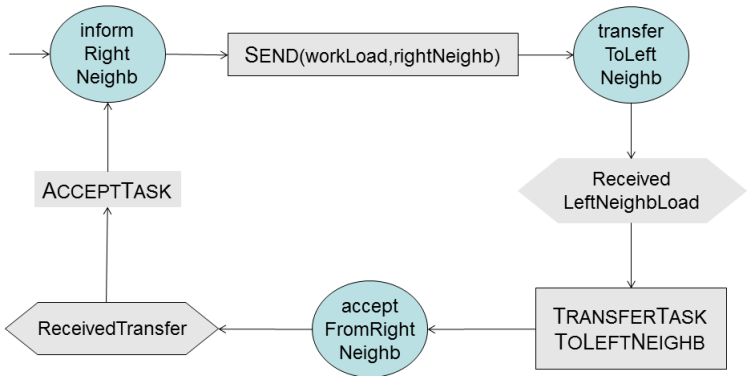


Fig. 5. 3-phase RINGLOADBALANCE ASM

TRANSFERTASKTOLEFTNEIGHB =
  let $\{leftNeighbLoad\} = mailbox \cap Nat$
    if $workLoad > leftNeighbLoad$ // there is a task to transfer
      then choose $task \in WorkLoad$
        SEND($task, leftNeighb$) DELETE($task$, $WorkLoad$)
      else SEND($nothingToTransfer, leftNeighb$)
    CONSUME($leftNeighbLoad$)
ACCEPTTASK =
  let $\{transfer\} = mailbox \cap (Task \cup \{nothingToTransfer\})$
    if $transfer \in Task$ **then** ADD($transfer$, $WorkLoad$)
    CONSUME($transfer$) // msg removal from mailbox

The correctness property—that eventually the workload difference between two neighboring nodes becomes and remains at most 1, with constant total workload—follows by induction on the *workLoad* count differences, to be compared with the lengthy verification of Fig.37.1 in [28, p.291-297].

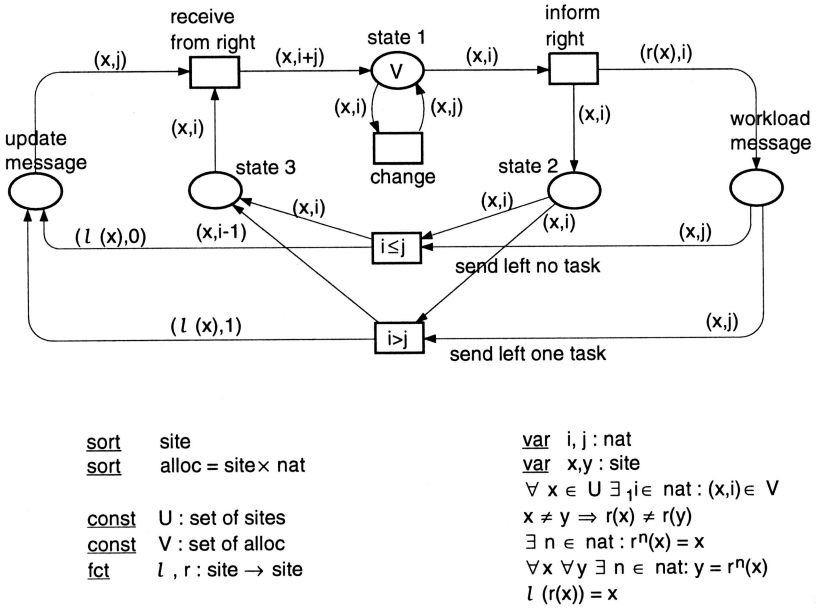


Figure 37.2. Distributed load balancing in a floating environment

In [28] Fig.37.2 is proposed for an adaptation to dynamic workload change triggered by the environment. Comparison with the ASM refinement in Fig. 6 illustrates the following problem with PN modeling of multi-agent systems.

Idiosyncrasy 4. The lack of component structure[8] and more generally of the structure of communicating agents and/or their environment leads to *model environment actions by nondeterministic internal transitions* (abuse of nondeterminism, see also Sect.2.9).

In fact Fig.37.2 models the interaction of one (global?!) environment with any local process as a nondeterministic internal transition *change*:

> From the perspective of the local balance algorithm, this interference shines up as nondeterministic change of the cardinality of the site's workload. ([28, p.141-142]

Similarly in the PNs in Fig.27.7,28.1,28.2 the message loss action of the communication medium is modelled as a nondeterministic internal action of the file transfer message protocol. Such a theoretically sound use of nondeterminism

[8] Compare this with the "illustration of the power of decomposition methods in enabling simple descriptions (and proofs) of complicated distributed algorithms" in [22], quote from p.532.

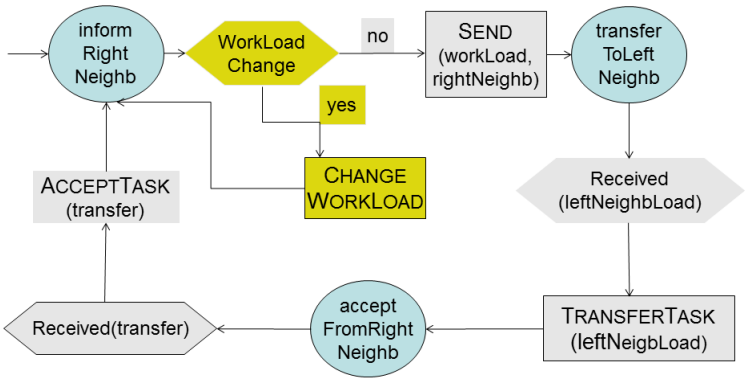


Fig. 6. DynRingLoadBalance ASM

fits the semantical PN and verification framework, but is inappropriate from the modeling point of view. In asynchronous systems "input actions are assumed not to be under the automaton's control—they just arrive from the outside" [22, p.200]. So we model the environment/process interaction in Fig. 6 by letting each p a) watch a local event—an input location $workLoadChange_p$ with values in $\{add(T), delete(T), noInput\}$—and b) ChangeWorkLoad when triggered to do so and otherwise execute RingLoadBalance as before. For the local process there is no nondeterminism here.

$WorkLoadChange$ 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

For the same reason we model in Fig. 4,3 the trigger events which start the process (but do not appear in the PNs in Fig.33.3,30.1) and in Sect. 2.6,2.7 the *timeout* events which trigger resending (but do not appear in the PN models in Fig.27.7,28.1,28.2) as guards for local process actions. Treating timeout as nondeterminism makes interfacing scheduling mechanisms difficult the *timeout* location in Fig. 9,8 prepares the ground for.

2.5 Consensus in graphs

The algorithmic idea to "organize consensus about some contract or agreement among the sites of a network", 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" [28, p.134] is that every agent (site, node) may

- spontaneously go to *agreed* (when without new requests),

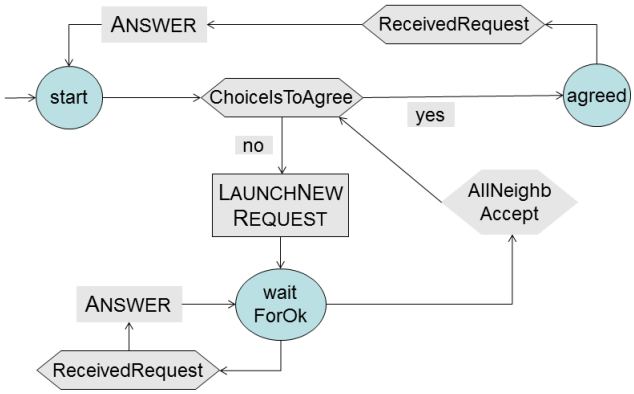


Fig. 7. 3-phase CONSENSUS ASM

- LAUNCHNEWREQUEST to its *Neighb*ors and *waitForOk* from them,
- receive and ANSWER requests,
- if *AllNeighbAccept* its last launched request either go to *agreed* or once more LAUNCHNEWREQUEST

such that IF the algorithm terminates (maybe never), then all agents *agreed* and there are no requests left.

The algorithmic idea can be directly traced in the control structure of Fig. 7, to be compared with Fig. 35.1 (copied from [28]). In particular the predominant role of nondeterminism in *start* mode is brought to the surface in Fig. 7 by the explicit *ChoiceIsToAgree* option (whose definition below uses a *choice* function as interface for "the criteria for a site to accept or refuse a proposed contract" [28, p.134]), whereas from the PN in Fig. 35.1 this feature has to be extracted from the edge structure at pending sites. Also the encoding of requests/answers by tokens $(y, x)/(x, y)$ in 'initiated/completed' places yields an artificial initialization (instead of an initially empty mailbox), described by (pg.135):

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

For the sake of comparison completeness we list here the (straightforward) definitions for CONSENSUS predicates and actions.

ChoiceIsToAgree iff $(choice(\{agree, propose\}) = agree)$
LAUNCHNEWREQUEST = // broadcast new request to neighbors
  forall $n \in Neighb$ SEND($requestFrom(\mathbf{self}), n$)
  REINITIALIZEREPLIES
REINITIALIZEREPLIES =
  forall $n \in Neighb$ CONSUME($replyFrom(n)$)
AllNeighbAccept = **forall** $n \in Neighb$ $Received(replyFrom(n))$
ReceivedRequest =
  forsome $n \in Neighb$ $Received(requestFrom(n))$

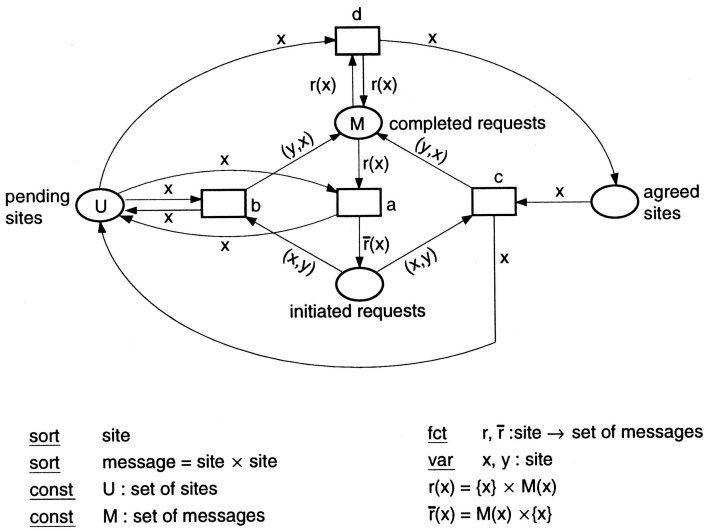


Figure 35.1. Basic algorithm for distributed consensus

ANSWER = **choose** $n \in Neighb$ **with** $Received(requestFrom(n))$
SEND($replyFrom(\mathbf{self}), n$) CONSUME($requestFrom(n)$)

An 'Advanced Consensus' PN with quiet/demanded sites (Fig.35.2 copied from [28]) illustrates how to adapt Fig.35.1 to the following requirements change request [28, 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.

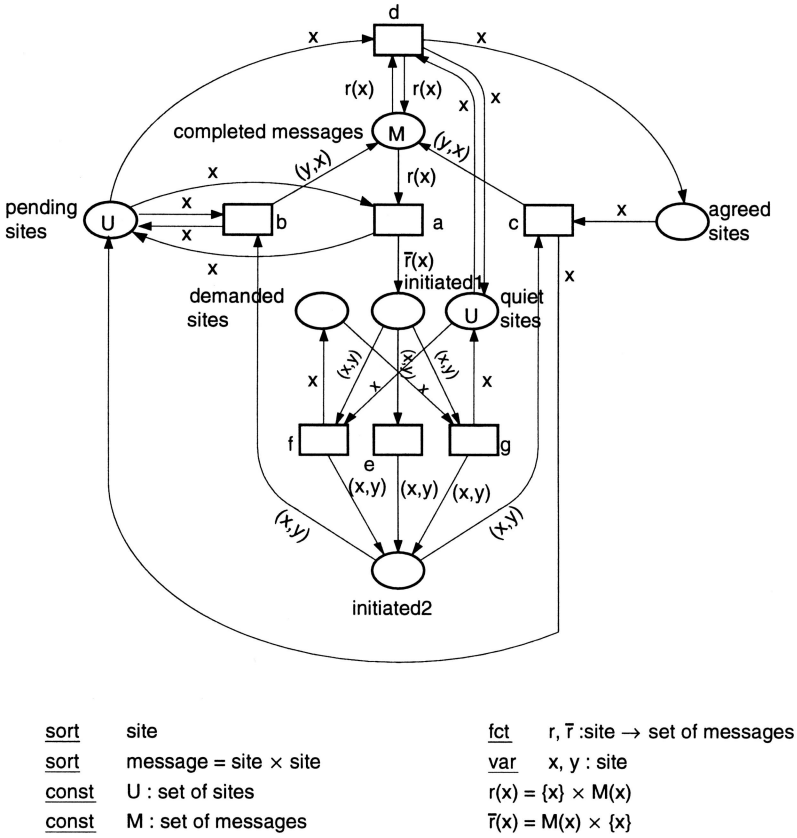
To reflect this in the ASM model it suffices to (textually!) add to the structurally unchanged Fig. 7 the needed new attribute, action and guard:

- $Quiet \in \{true, false\}$, $Demanded =$ **not** $Quiet$: attribute added to signature
- SWAP = ($Quiet$:=**not** $Quiet$): action added to ANSWER[9]
- $Quiet = true$: constraint added to *ChoiceIsToAgree* guard

2.6 Alternating Bit Protocol

The Alternating Bit protocol [22, 22.3] transfers any sequence $F(1), \ldots, F(n)$ of files from a *sender* to a *receiver* s.t. eventually the *receiver* has a copy $G = F$,

[9] We interprete 'may cause' as 'causes'; otherwise add **choose** $s \in \{swap, noSwap\}$ **in if** $s = swap$ **then** SWAP to ANSWER.



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

assuming that the communication medium may lose (but not change) finitely many consecutive messages.[10] The algorithmic idea is that in rounds (one per file with initially file number $currRound = 0$ at sender, $currRound = 1$ at receiver; the receiver remains round-ahead of the *sender*):

- the *sender* SENDs the current file and continues to RESENDFILE upon *timeout* until an acknowledgement of receipt arrives from the *receiver*, whereafter in the next $currRound + 1$ the *sender* will STARTNXTFILETRANSFER,
- when sending file $F(round)$ a synchronization bit *round* mod 2 is

[10] The protocol specification does not include a specification of the communication medium. It seems to be another example for PN Idiosyncrasy 2 that Fig.27.7,28.1,28.2 also define the possible message loss by the transmission lines as an internal nondeterministic action of the PN. Message loss is not an action of the protocol agents but of the communication medium.

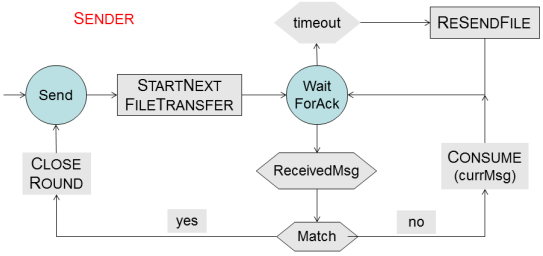


Fig. 8. 2-phase AltBitSender ASM

- attached to file messages $(F(round), round \bmod 2)$,
- extracted and resent by the *receiver* as acknowledgment message,
- checked upon *ReceivedMsg* by *sender*/*receiver* for *Match*ing its own synchronization bit and in case of matching is flipped for the next $round+1$.

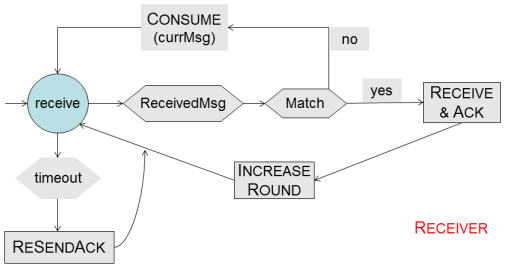


Fig. 9. AltBitReceiver ASM

The algorithmic idea is easily traced in Fig. 8 and Fig. 9; compare with Fig. 27.7 (copied from [28, p.107-111] where its details are explained on 5 pages).

In the ASM the sender/receiver predicates/actions work under the no-msg-overtaking assumption, reflected by a FIFO-queue mailbox *MsgQueue*.

StartNextFileTransfer =
  Send$((nextFile, nextSyncBit), receiver)$ IncreaseRound
    where $nextFile = F(currRound + 1)$
      $nextSyncBit = currRound + 1 \bmod 2$ // flipped sync bit
ReSendFile = Send$((F(currRound), currRound \bmod 2), receiver)$
ReceivedMsg = iff $MsgQueue \neq [\,]$ // mailbox not empty
Match iff $syncBit(currMsg) = currRound \bmod 2$
CloseRound = Consume$(currMsg)$ **where** $currMsg = head(MsgQueue)$
IncreaseRound = $(currRound := currRound + 1)$

Receive&Ack = StoreFile SendAck Consume$(currMsg)$

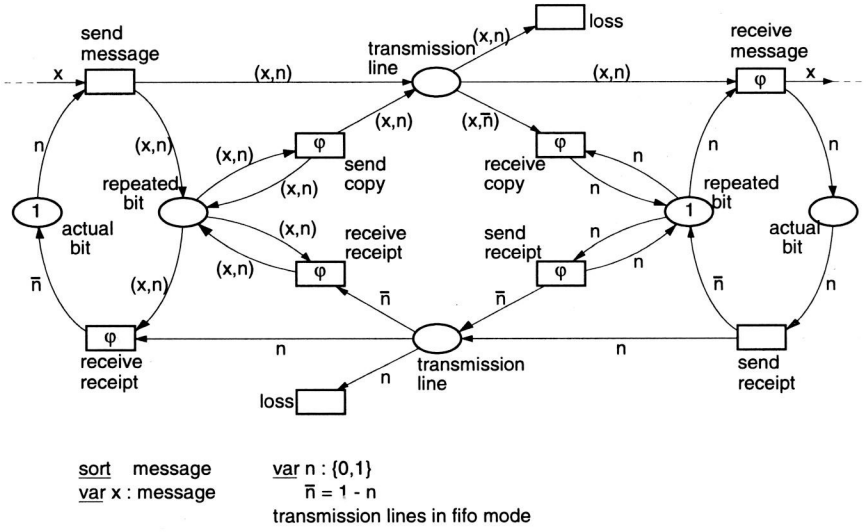


Figure 27.7. The alternating bit protocol

SendAck = Send(*syncBit*(*currMsg*))
ReSendAck = // NB. receiver is round-ahead of sender
Send(*flip*(*currRound* mod 2), *sender*) // previous sync bit
StoreFile = (*G*(*currRound*) := *file*(*currMsg*))

We found no PN verification for Fig.27.7 in [28] although it is easy to prove the correctness by induction on AltBit run phases, see Fig. 10 and [22, 22.3].

Idiosyncrasy 1b. *The token-based transition view introduces algorithmically extraneous technicalities* (language specific details about checking the presence of tokens, token insertion/deletion) which are unrelated to the subject matter, obfuscate the intutive understanding of the algorithm's behavior and complicate both its verification and further refinement to code (implementation).

For example the token-based transition view is responsible for:

- *moving around unchanged data* between places or worse deleting and simultaneously adding them from/to one and the same place (Fig.27.7 shows many examples), a technicality that may produce an overwhelming effect on net size and readability and is analogous to the *frame problem* (avoided by ASMs!) of logical descriptions of the no-change part of actions,
- *doubling of locations for same data* involved in different transitions, possibly with different values; e.g. in Fig. 27.7 places *actualbit* and *repeatedbit* double the *syncBit* location at sender and receiver part of the net,
- *simulation of shared locations by token manipulation* which multiplies places and transitions. This point is illustrated in detail in Sect. 2.8.

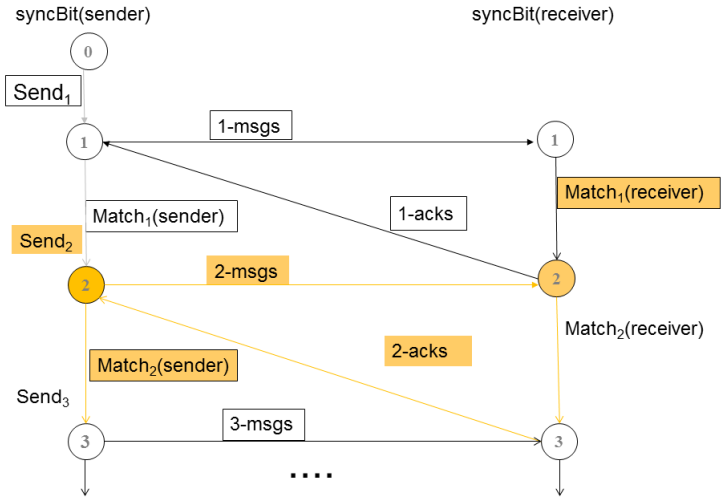


Fig. 10. AltBit run phases

2.7 Adapting AltBit to Sliding Window Protocol

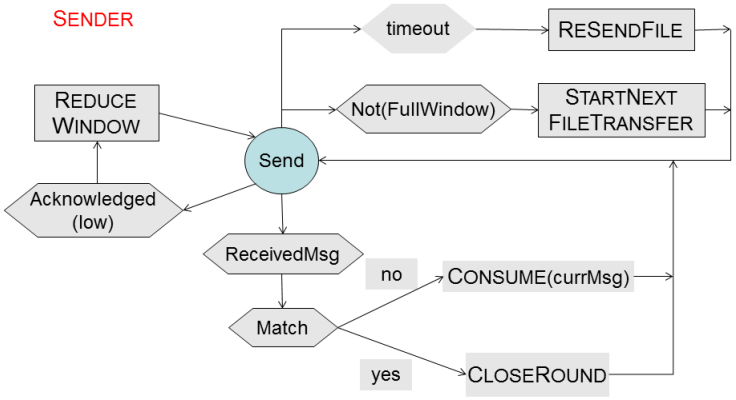


Fig. 11. SlidingWindowSender

The no-msg-overtaking assumption can be dismissed by replacing single file transfer rounds by (re-) sending in any order multiple files $F(i)$ (and corresponding acks) distinguished by their index i, to be used as (still called) 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 StartNextFileTransfer and IncreaseWindow (by the next syncBit $high := high + 1$) as long as **not** *FullWindow* (where $FullWindow = high - low + 1 = maxWinSize$),
- if $Acknowledged(low)$ (i.e. that $F(low)$ has been received) the *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 updating *low*.

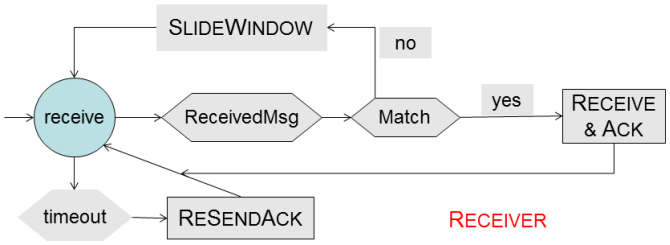


Fig. 12. SlidingWindowReceiver

To turn this description into an ASM model one can preserve the component structure of AltBit, except for adding the needed new ReduceWindow component and collapsing the abandoned sequential send/waitForAck phases; the rest is data refining the send/receive predicates and actions as indicated below. This yields the definition in Fig. 11 and Fig. 12 we invite the reader to compare with the largely *structural PN redesign* of Fig.27.7 to Fig.28.1 (copied from [28]).

- StartNextFileTransfer is refined by

  $nextFile = F(high + 1), \quad nextSyncBit = high + 1$
  IncreaseRound = $(high := high + 1)$ // IncreaseWindow
- $ReSendFile$ = Send($(F(low), low), receiver$)
- $Match = (low \leq syncBit(currMsg) \leq high)$ // syncBit in window
- CloseRound refines 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).

In the SlidingWindowReceiver Receive&Ack is not followed any more by IncreaseRound and in case of no *Match* one cannot Consume the $currMsg$ with $syncBit(currMmsg) > high$ but instead the receiver must SlideWindow to let $currMsg$ *Match*. Thus ReSendAck = Send($low, sender$) and

SlideWindow =
  let $s = syncBit(currMsg)$
    $high := s$ and $low := max\{1, s - maxWinSize + 1\}$

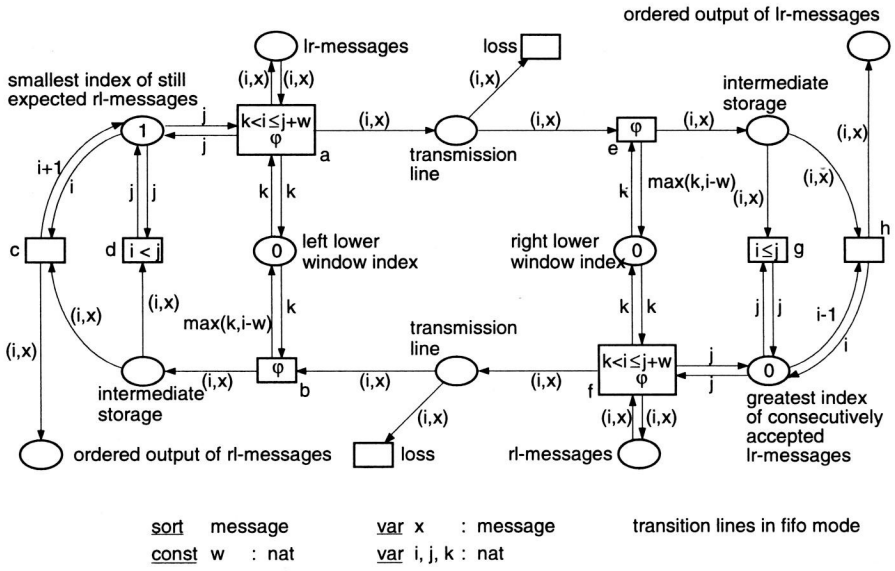


Figure 28.1. Balanced sliding window protocol with unbounded indices

Remark. In case of no message overtaking finitely many indices suffice using $+1 \bmod r$ for updating $low, high$ for a sufficiently large r depending on $maxWinSize$: a *pure data refinement* in the ASM framework. Compare this with the sophisticated extension of Fig. 28.1 to Fig.28.2 (copied from [28]).

2.8 Mutual Exclusion problem

This is about allocation of one or more nonshareable resources to $n \geq 2$ processes. Peterson's Mutex algorithm ([22, 10.5]) works for one resource and n processes, satisfying also the **Lockout-Freedom Requirement** ([22, 10.4]):

- If each process always returns the resource, every process that reaches the *trying* region (where it competes for the resource) *eventually* will enter the *critical* region (where it uses the resource).
- Every process that reaches its *exit* region (where it stops using the resource) eventually will enter a *remainder* region (without interest in the resource).

In MUTEXPETERSON$_n$ processes share some locations used to compete at successive $level = 1, \ldots, n-1$ (a local process variable, initially $level = 1$):

- for every *level* value a global location $stickAt_{level} \in \{1, \ldots, n\}$ (with arbitrary initial value) all processes can read and write. It must have been FETCHed by p (i.e. updated to p) and—in case some other process is interested—later be released (i.e. FETCHed by another interested process) before p becomes the *level Winner*,

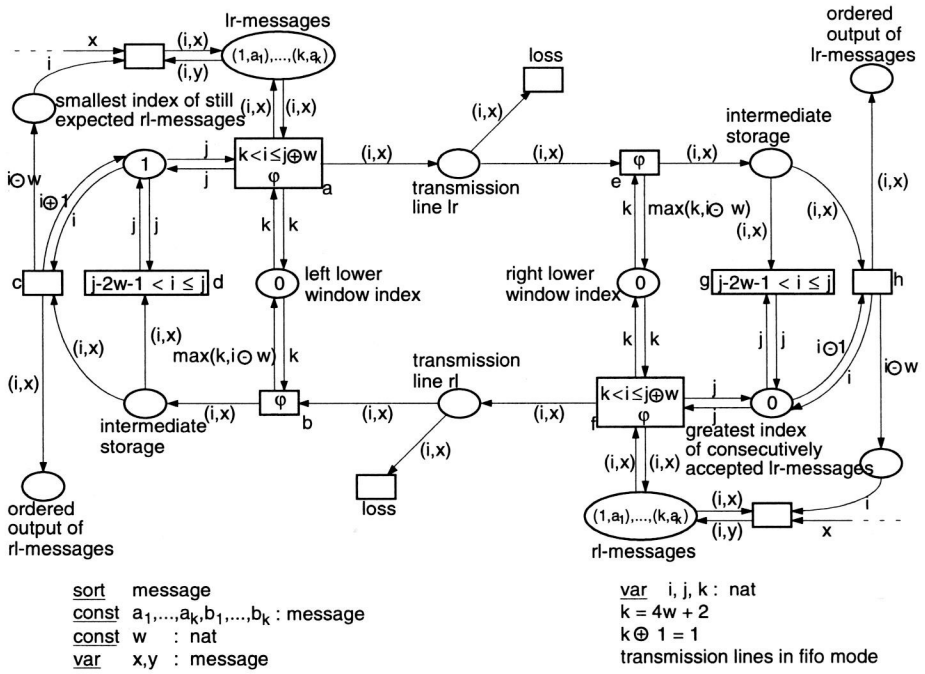


Figure 28.2. Balanced sliding window protocol with bounded indices

- a local location $flag \in \{0, \ldots, n-1\}$ at every p (initially $flag = 0$) which is writable by p and readable by each other process. $flag_p = l > 0$ indicates that p started the competition ('is interested') at level l to get the resource.

We first explain the case $n = 2$ treated in [28], with fixed competition $level = 1$. Compare Fig.13.7 (copied from [28]) with Fig. 13 and its local actions (formulated with *level* parameter to prepare their generalization to multiple levels):

RE/SETFLAG = ($flag := 0/level$) // for $n = 2$ holds $level = 1$
FETCHSTICK = // for $n = 2$, one more guard below for $n > 2$
  if (**not** $HasStick_{level}$) **then** $stickAt_{level} :=$ **self** // **skip** if $HasStick_{level}$
    where $HasStick_{level}$ iff $stickAt_{level} =$ **self**
Winner = *NobodyElseInterested* **or** *MeantimeSomebodyElseFetchedStick*
NobodyElseInterested($level$) = **forall** $p \neq$ **self** $flag_p < level$
  // for case $n = 2$ this means $flag_{theOtherProcess} = 0$
  where $theOtherProcess = \begin{cases} 1 \textbf{ if self}= 2 \\ 2 \textbf{ else} \end{cases}$ // only for case $n = 2$
MeantimeSomebodyElseFetchedStick($level$) iff $stickAt_{level} \neq$ **self**
  // meaning in case $n = 2$ that $stickAt_1 = theOtherProcess$

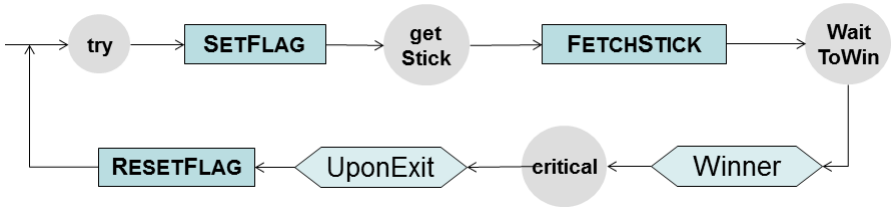


Fig. 13. 4-phase MUTEXPETERSON$_2$ ASM

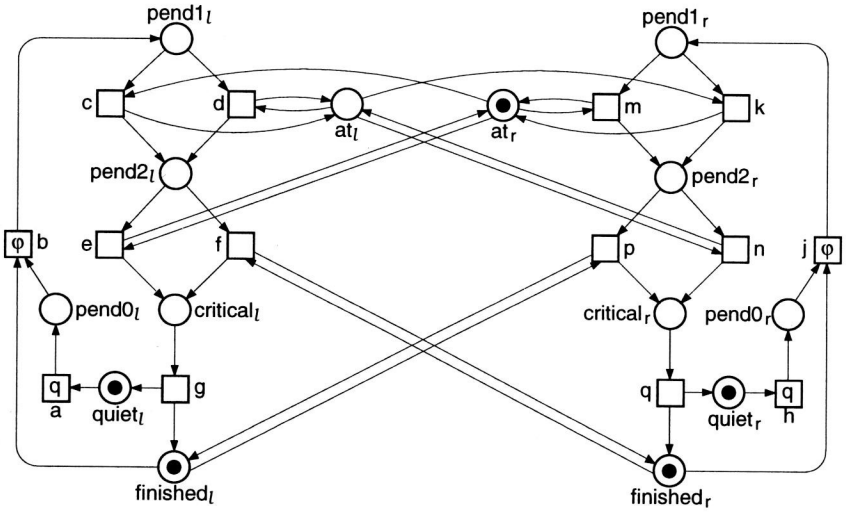


Figure 13.7. Peterson's *mutex* algorithm

The reader may also compare the detailed, easy-to-follow correctness proof for MUTEXPETERSON$_2$ in [22, p.281-282] with the technically involved PN verification in [28, p.180-182] which uses the 'evolution proof graph' in Fig.49.6 (copied from [28]) whose 16 'nodes ... are justified' one by one with help of 4 invariants. These nets and proof graphs risk to explode when generalizing Fig.13.7 from 2 to $n > 2$ processes (see [22, 10.5.2] and the ASM refinemen step below).

Fig.13.7 illustrates how the *simulation of shared locations by token manipulation* multiplies places and transitions (see Idiosyncrasy 1b, Sect. 2.6). Here it produces 8 places and 12 transitions to simulate updates of 3 locations:

- *stickAt* read/write is simulated by 6 places $pendi_l, pendi_r, at_l, at_r$ $(i = 1, 2)$ with
  - 2 token swapping transitions which simulate the 2 possible writes
  - 4 token checking read ('simultaneous delete/add token') transitions
- multi-reader single-writer *flag*s are encoded by places $finished_l, finished_r$ with

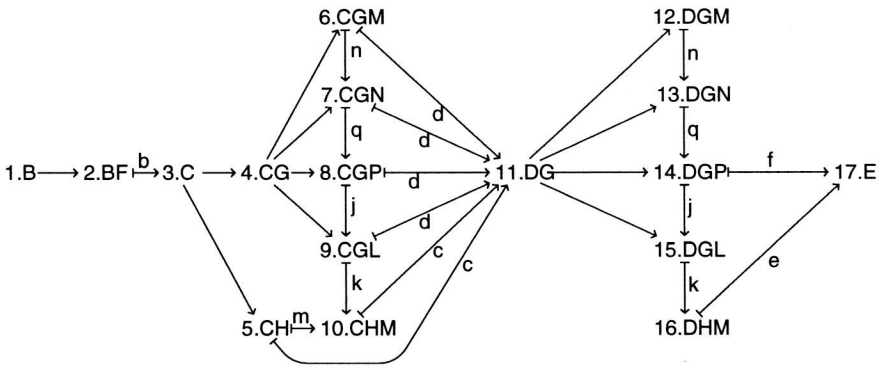


Figure 49.6. Evolution of Peterson's algorithm

- for a reader a transition to simulate reading a value of the writer's *flag*
- 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

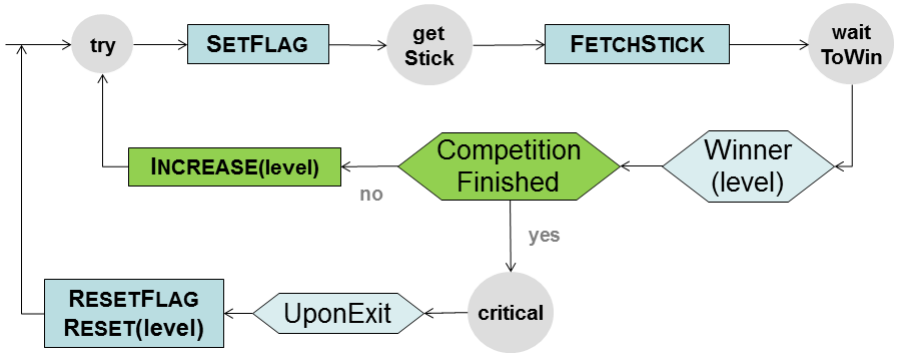


Fig. 14. MUTEXPETERSON$_n$ (competition thru levels) ASM

Multiplication of places/transitions *risks to result in spaghetti PNs* where shared locations involve more than two values and/or processes. In comparison the ASM refinement MUTEXPETERSON$_n$ for $n > 2$ in Fig. 14 iterates the MUTEXPETERSON$_2$ competition through *level*s 1 to $n-1$, guaranteeing that at each *level* there is a least one *loser*, i.e. a process that has to *waitToWin* until

NobodyElseInterested(*level*) any more. Thus at level k at most $n-k$ processes can win so that at most one can win at level $n-1$.

Refining Fig. 14 means to add an iterator component and a further guard to FETCHSTICK guaranteeing that in each step at most one process p can write $stickAt_{level}$. To permit an implementation by any component which computes the *select*ion we formulate the additional guard using a choice function:

$chosenWriterFor(stickAt_{level}) =$ **self**
  where $chosenWriterFor(stickAt_l) =$
    $select(\{p \mid flag_p = l$ **and** $mode_p = getStick$ **and** $stickAt_l \neq p\})$
CompetitionFinished iff $level = n-1$
INCREASE(*level*) = $(level := level + 1)$ RESET(*level*) = $(level := 1)$

2.9 Remark on nondeterminism and interleaving

MUTEXPETERSON$_2$ illustrates an adequacy problem related to the use of nondeterminism in modeling with PNs (see Idiosyncrasy 4). Consider the case that the two processes, say *left* and *right*, both are enabled to fetch the stick—formalized in MUTEXPETERSON$_2$ by both being in $mode = getStick$ and in Fig.13.7 by a token in both places $pend1_l, pend1_r$—and that the stick is with process *right*—i.e. in MUTEXPETERSON$_2$ $stickAt_1 = right$ resp. in Fig.13.7. place at_r has a token. Which process will be the first to enter the critical section depends in the PN of Fig.13.7 on a nondeterministic choice which one of the two enabled but conflicting transitions m and c is fired first; one will have to define this (centralized?) selection separately when refining the algorithm to an implementation. The MUTEXPETERSON$_2$ ASM needs no such additional selection procedure: both processes can independently (even simultaneously) make their next enabled FETCHSTICK step and proceed to $mode = waitToWin$ whereafter the decision about which process can enter the critical section depends only on the *Winner* condition, as in Fig.13.7 for the process which succeeded to put a token in place $pend2_l$ resp. $pend2_r$.[11] So the concurrent ASM needs no additional external help by nondeterministic or other choices: if such not furthermore specified choices are allowed to appear in the algorithm one could simplify matters by choosing right away one of the two processes to get the resource.

However for MUTEXPETERSON$_n$ with $n > 2$ the problem reappears. Consider the analogous case where at any given *level* there are $n - (level - 1)$ simultaneously enabled processes to fetch the stick. Since it is assumed that only one process can write the shared location $stickAt_{level}$ an additional selection procedure is needed[12] to decide upon who becomes the $chosenWriterFor(stickAt_{level})$

[11] [22, p.280] has a similar nondeterministic choice for the order in which the processes write the shared $stickAt_1$ (there called *turn*) location, a nondeterminism resulting from the interleaving assumption of the underlying asynchronous shared memory I/O automata execution model. But this extraneous nondeterminism could easily be avoided the same way as in the ASM model.

[12] This is already a form of mutual exclusion. For a solution of the mutual exclusion problem for multiple writers see [19].

(which in the considered case will make it the loser at this *level*). By introducing the function $chosenWriterFor(stickAt_{level})$ into the ASM model in Fig. 14 this selection is at least made explicit. In [22, p.286] the selection is hidden in the interleaving assumption which decides in each step upon which process among the enabled ones to write $stickAt_{level}$ (called there $turn(level)$) will be chosen to write that location. As refinement to compute such a choice function often the *head* function of a FIFO *queue* structure is used; this means that the queue is assumed to decide upon the insertion order if two or more processes request simultaneously to get INSERTed.[13] The question remains why instead of running a complicated algorithm such a queue structure is not used right away to organize the decision about which one among $n > 2$ simultaneously interested processes is selected to (be the first to) write a shared location which guards entering the critical section, choosing once a winner instead of choosing $n - 1$ times a loser (one per level).[14]

3 Lifting Petri nets to concurrent ASMs

The analysis of PNs in Sect. 2 reveals three major sources of the inadequacy of the PNs proposed in [28] as high-level models for distributed algorithms:

- *insufficient abstraction* and *introduction of algorithmically irrelevant technicalities* resulting from the low-level token-based view of objects (including executing agents), predicates and transitions—despite of the abstract data-interpretation of tokens in [28, Sect.16] as elements of some abstract domain so that "local states of nets are ... propositions and predicates, thus providing the elements for logic-based analysis techniques"(ibid., p.VII),
- *lack of component structure (architectural system view)*[15] *and of separation of concerns*, mainly due to the global overall process view, thus failing to support appropriate modeling of interaction among agents and/or the environment[16] and complicating the implementation process,
- *complexity of graphical layout* even for small algorithms, indicating a too great esteem of the graphical 'nature' of PNs as a help to understand (or even to define)[17] them and the *lack of an appropriate combination of visual and textual description elements*.

[13] For a prominent example see [22, p.346]. [20] uses the assumption only for concurrent atomic read/write operations of single digits.

[14] See [21, p.561] for a distributed mutual exclusion algorithm which deals with the case where the order matters in which requests for the resource are made.

[15] See the observation in [11, p.3]: "The core issue of Petri nets is that they model behavioral aspects of *distributed systems*, i.e., systems with components that are locally separated and communicate with each other. Surprisingly, neither components nor any notion of locality appears with the usual definition of a Petri net".

[16] Note that at the time of Petri's doctoral thesis [24] computers were monolithic mainframes, there were no agents interacting via pools of networks, servers, services, etc.

[17] In [28, p.V] it is even suggested that "The hurried reader may just study the pictures"! See also the analysis in [11, p.2] where the characteristics that "Petri nets are

ASMs are well-known not to suffer from any of these problems. In the contrary they provide a justifiably most general abstraction concept—see the various forms of ASM thesis ([8] contains over a dozen of references)—which is coupled to a correspondingly general refinement concept [4] that supports a strong separation of concerns and componentwise design discipline—see the numerous successful practical applications of the ASM refinement method (e.g. [29], for other programming language applications see the recent survey [6]).

The concept of multi-agent ASMs carries these abstraction and refinement capabilities to component-based multi-process modeling and concurrency [8].

Using control-state ASMs as we did in Sect. 2 allows one to explicitly separate and appropriately combine visual (control flow, run-time relevant) and textual (data structure related) description elements to express the underlying (whether static or dynamically changing) data, predicates on data, the effect of actions on data, communication and resources. Such a conceptually well-founded practical combination of these two fundamental elements of a complete system state is crucial for a practical modeling method.

How can PNs share these properties of concurrent ASMs? As has been analyzed already in [9, pg.297] (and elaborated in one more bit of detail in [8, Appendix C]) PNs represent a specific class of multi-agent ASMs, characterized by a particular notion of state and various concepts of (interleaved, lock-step, concurrent, also called distributed) run. In fact each PN P can be defined as a multi-agent ASM where each agent has exactly one transition t of P as its rule of form

if *Enabled*(t) **then** FIRE(t)

This view—which holds mutatis mutandis also for the version of PNs used in [28] and more generally for colored and so-called high-level PNs—allows one to enhance the modeling capabilities of PNs by abstraction, refinement, separation of concerns, component- and agent-based modeling features and combinability of visual and textual description elements the ASM method comes with. We wouldn't even be surprised to see (in particular PN-based) formulations of well-known concepts of concurrency theory [30] get simplified within an ASM-based mathematical framework [8],[18] in analogy to the simplifications one can obtain for classical computation theory [1] and the theory of algorithms by using ASMs instead of the multitude of specific (historically important) machine concepts and complexity measures [9, Sect.7.1.1].

---

a *graphical notion* and at the same time a *precise mathematical notion*" are taken as "the most important properties".

[18] Right before sending this paper for the Proceedings Klaus-Dieter Schewe and myself discovered that distributed PN runs are distributed ASM runs in the sense of Gurevich [13]. A further investigation of the consequences of this surprising fact remains to be done.

4 Related Criticism

Other researchers have observed the inadequacy of PNs to model complex computational systems in practice, in particular in the field of Business Process Modeling (BPM). One main issue there is the lack of sufficient support for a seamless inclusion of data, resources and communication among processes into control-flow-centric (e.g. PN inspired BPMN) models, for references see the survey in [17, Ch.2]. On the positive side the control-flow character of traditional PNs comes out clearly from the beautiful foundational result that computationally they are equivalent to asynchronous automata nets built up from a few rather elementary components [26]. But modeling BPs needs more expressive means. For further discussion of this issue see [10], the work at IBM on the data-centric Guard-State-Milestone approach [15],[19] Fleischmann's Subject-Oriented-BPM approach [12], the observation in [18, p.132] that with (even colored) PNs "the modeling is quite low-level and ... does not provide any higher-level abstractions" resulting in unreadable diagrams (a fascinatingly useless (PN-tool-generated!) example is shown in op.cit. Fig.9), the recent comparative BPM case study by YAWL experts reporting that "YAWL diagrams only depict the control flow and not the resource perspective" [14, Sect.4], etc. Also in the domain of stochastic simulation of biological systems it has been recognized that the PN approach "makes it difficult to exploit compositionality of nets to build models incrementally" [31, p.354].

5 Conclusion

The author of [28] warns the reader that the "book's scope is modest, as it sticks to a choice of small and medium size algorithms", furthermore the proposed design and analysis method is disclaimed to work for large systems where "systematic refinement of specifications and compositional proof techniques are inevitable"[28, p.13]. However it is also stated that "nevertheless this book's claims are ambitious: Just as PASCAL-like programming structures and Hoare-style proof techniques appear optimal for a wide class of sequential algorithms, this book's formalism is suggested to be optimal for a wide classe of distributed algorithms" and "can help the practitioner to design distributed algorithms"(ibid., p.V). Neither our experience in teaching nor the cooperation with industrial partners nor the analysis above confirm these two claims. Furthermore if "large algorithms require adequate techniques for *small* algorithms" (ibid., p.13), by contraposition an unsatisfactory framework for small algorithms will not be helpful for large systems. We hope that the above definition of ASMs shows that some improvement can be obtained for PNs, even for design and analysis of small distributed algorithms, by freeing oneself from the low-level token-based PN view and switching to a language where distributed control, data, resource and communication structures can be combined in a technically simpler way,

[19] In [7] ASM nets are defined through which IBM's Guard-State-Milestone approach [15] to BPM can be equipped with the systematic ASM refinement method.

exploiting for a better visualization also the old fashioned flowchart representation means. This holds even more because the ASM method is not exhausted by using control-state ASMs for small distributed algorithms but by its numerous (including industrial) applications in a variety of fields has contributed to reduce the huge gap between much of academic theory and the prevailing software and hardware practice (see [2],[9, Ch.9],[6] for a survey).

6 Appendix: Reaction to Criticism

Some critic argued that using only one book for the comparison could relativize the conclusions because the idiosyncrasies may belong less to PNs than to the way the book's author uses them. This is possible, however this paper targets not a person but a proposed scientific method. To all practical purposes, in front of thousands of PN publications, what better one could have done—without being an active PN researcher but with the desire to see how PNs behave compared to ASMs when modeling distributed systems—than carefully study a book about a core use of PNs, written by an author who is considered an authority in the field and who promises nothing less than a new proof technique at the level of the Hoare-style proof method, furthermore a very carefully written book (especially concerning the graphical layout of the PNs) whose examples are well-known distributed algorithms every computer science student learns so that the comparison can be explained and followed without need to assume additional knowledge.

Some critics wondered whether comparing the graphical PN notation and the graphical flowchart-like notation used for (the non-textual part of) control-state ASMs is fair. This paper makes no such comparison at all:[20] for given PNs and ASMs we compare the intellectual effort (and the time needed) to define them, explain them, understand them, justify them as adequate (ground) models of the requirements, prove properties of interest for them; in making all these comparisons we *use the behavioral semantics of the graphical notations* in which the models are expressed and which are rigorously defined for PNs as they are for ASMs (see [9, Sect.2.2.6]). For the behavioral comparison of models we point to figures which define the models, but we do NOT compare the figures as graphical objects. For the behavioral comparison of the PN/ASM *ground models* which are expected to satisfy given requirements and to "help the practitioner to design distributed algorithms" [28, p.V] it does not matter whether the graphical layout of the behavior defining diagrams is or can be furthermore formalized, e.g. to make the diagrams executable or machine checkable or subject to transformations, etc. That a rather formalized version of PN diagrams is needed for the proof-graph method is clear, which brings us to the next critical objection we heard.

Some critics argued that the reader finds no ASM-based proofs in this paper. Clearly not; the only thing one could reasonably do in this paper was to point

[20] Idiosyncrasies 3 and 3b which do speak about graphical PN features however discuss only possible deficiencies of PN diagrams; they are not related to flowcharts.

the reader to proofs in [22]—the pseudo-code there is similar to our ASMs, so the proofs apply mutatis mutandis to concurrent ASMs—to be compared to the corresponding ones in [28]; in addition in a few places, where the proofs are so simple to be built directly from scratch we gave some indication which property or proof scheme to use for this. But also in these cases the proofs can be found in [22] and be compared with those in [28]. We are not talking here about the use of machine support for proof activities. The comparison in this paper was about how to a) model distributed algorithms to "help the practitioner to design distributed algorithms"[28, p.V] and b) how to verify their properties of interest "to make intuitive statements and conclusions transparent and precise, this way deepening the reader's insight into the functioning of systems" [28, p.143]. The doubt we epxress is whether using PN proof graphs helps to reach this goal.

Acknowledgement. I thank over a dozen of colleagues who read and commented upon the many previous versions of this paper; I do not mention names so that the entire responsibility of what is stated in this paper remains mine.

References

1. E. Börger. *Computability, Complexity, Logic (English translation of "Berechenbarkeit, Komplexität, Logik", Vieweg-Verlag 1985)*, volume 128 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1989.
2. E. Börger. The origins and the development of the ASM method for high-level system design and analysis. *J. Universal Computer Science*, 8(1):2–74, 2002.
3. E. Börger. The ASM ground model method as a foundation of requirements engineering. In N.Dershowitz, editor, *Verification: Theory and Practice*, volume 2772 of *LNCS*, pages 145–160. Springer-Verlag, 2003.
4. E. Börger. The ASM refinement method. *Formal Aspects of Computing*, 15:237–257, 2003.
5. E. Börger. Construction and analysis of ground models and their refinements as a foundation for validating computer based systems. *Formal Aspects of Computing*, 19:225–241, 2007.
6. E. Börger. The Abstract State Machines method for modular design and analysis of programming languages. *J. Logic and Computation*, 2014. Special Issue *Concepts and Meaning* (Leitsch Festschrift). First published online December 18, 2014. doi:10.1093/logcom/exu077.
7. E. Börger and A.Fleischmann. Abstract State Machine nets. Closing the gap between business process models and their implementation. In *Proc. S-BPM ONE 2015*, ACM Digital Library, ISBN 978-1-4503-3312-2, April 2015. ACM.
8. E. Börger and K.-D. Schewe. Concurrent Abstract State Machines. *Acta Informatica*, 2015. http://link.springer.com/article/10.1007/s00236-015-0249-7, DOI 10.1007/s00236-015-0249-7.
9. E. Börger and R. F. Stärk. *Abstract State Machines. A Method for High-Level System Design and Analysis*. Springer, 2003.
10. D. Cohn and R. Hull. Business artifacts: a data-centric approach to modeling business operations and processes. *IEEE Data Engineering Bulletin*, 32:3–9, 2009.
11. J. Desel and G. Juhas. What is a Petri net? Informal answers for the informed reader. In H. Ehrig et al., editors, *Unifying Petri Nets*, volume 2128 of *LNCS*, pages 1–25. Springer, 2001.

12. A. Fleischmann, W. Schmidt, C. Stary, S. Obermeier, and E. Börger. *Subject-Oriented Business Process Management*. Springer Open Access Book, Heidelberg, 2012. www.springer.com/978-3-642-32391-1.
13. Y. Gurevich. Evolving algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
14. A. V. Hense and R. Malz. Comparison of the subject-oriented and the Petri net based approach for business process automation. In *S-BPM ONE 2015*. ACM, 2015.
15. R. Hull et al. Introducing the Guard-State-Milestone approach for specifying business entity lifecycles. In M. Bravetti and T. Bultan, editors, *Proc.7th Int.Conf. on Web Services and Formal Methods*, volume 6551 of *LNCS*, pages 1–24. Springer, 2011.
16. K. Jensen and L. M. Kristensen. *Coloured Petri Nets*. Springer, 2009.
17. F. Kossak et al. *A Rigorous Semantics for BPMN 2.0 Process Diagrams*. Springer, 2014.
18. E. Kühn et al. Peer-based programming model for coordination patterns. In R. De Nicola and C. Julien, editors, *COORDINATION 2013*, volume 7890 of *LNCS*, pages 121–135, 2013.
19. L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Commun. ACM*, 17(8):453–455, 1974.
20. L. Lamport. Concurrent reading and writing. *Commun. ACM*, 20(11):806–811, 1977.
21. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.
22. N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.
23. D. L. Parnas. The use of precise documentation in software development. Tutorial at FM 2006, see http://fm06.mcmaster.ca/t8.htm, August 2006.
24. C. A. Petri. *Kommunikation mit Automaten*. PhD thesis, Institut für Instrumentelle Mathematik, Universität Bonn, Schriften des IMM Nr.2, 1962.
25. http://www.informatik.uni-hamburg.de/TGI/PetriNets/.
26. L. Priese. Automata and concurrency. *TCS*, (25):221–265, 1983.
27. L. Priese and H. Wimmel. *Theoretische Informatik. Petri-Netze*. Springer, 2003.
28. W. Reisig. *Elements of Distributed Algorithms*. Springer-Verlag, 1998.
29. R. F. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer-Verlag, 2001.
30. G. Winskel and M. Nielsen. Models for concurrency. In *Handbook of Logic and the Foundations of Computer Science vol.4 (Semantic Modelling)*, pages 1–148. Oxford University Press, 1995.
31. R. Zunino et al. An imperative DSL to stochastically simulate biological systems. In C. Bodei et al., editors, *Degano Festschrift*, volume 9465 of *LNCS*, pages 354–374. Springer, 2015.