Protocol ASMs with time constraints

Kermit (Alternating Bit and Sliding Window) Protocol

Specification and Verification

Egon Börger

Dipartimento di Informatica, Universita di Pisa
http://www.di.unipi.it/~boerger

For details see Chapter 6.3 (Time-Constrained Asynchronous ASMs) of:

E. Börger, R. Stärk

Abstract State Machines

A Method for High-Level System Design and Analysis

Springer-Verlag 2003

For update info see AsmBook web page:

http://www.di.unipi.it/AsmBook

File Transfer Protocols

- Goal: for transferring sequences of files from a sender to a receiver, design communication protocols which guarantee the files to be transferred correctly (without fail, in the right order).
- Basic method: every file is sent, & retransmitted upon timeout, until an acknowledgement of receipt arrives from the receiver. When this happens, the current file transfer is closed & the next one is started.
- Identification technique:
  - alternating bit technique [Bartlett, Scantlebury, Wilkinson 1969]: sender and receiver use a synchronization bit. Its current value is
    - attached to file transfer messages by the sender
    - extracted and resent by the receiver as acknowledgement
    - upon message arrival checked by sender & receiver to match their own synchronization bit, which in case of matching is flipped in a synchronous manner, to be ready for the transmission round of the next file
  - sliding window technique: sender and receiver use window boundaries low, high, assigned to the numbers of messages in transit, which are checked upon arrival to match the boundaries, with sliding of the boundaries in case of matching

Sending - Checking - Resending Phases

The basic method results in the following template for sender rules, instantiated below for Alternating Bit & Sliding Window

Send ≡ If SendingTime then StartNxtFileTransfer

Check ≡ If CheckingTime then
If Match then CloseCurrFileTransfer
ClearCurrMsg

Retransmit ≡ If RetransmitTime then Retransmit

The template for the receiver rule is as follows:

Receive ≡ If ArrivalTime
then AcknowledgeReceipt
If Match then AcceptCurrFile
ClearCurrMsg

The Agents of File Transfer and their States (1)

- **sender, receiver** each equipped with its own
  - file(self): N $\longrightarrow$ DATA for the sequence of files to be transferred/stored: monitored for sender, controlled for receiver
    - null: DATA, a placeholder for no-data in acknowledgement messages
  - currNum(self): N, a controlled fct yielding the number of the file which is currently to be transferred resp. the next to be stored (this fct is refined below to a sliding window)
  - send, abstract msg sending action with source/target defined by the message (here wlog predefined)
  - queue(self): MESSAGE* to store arriving input (shared with input agent, see below)
  - timeout: monitored (with fairness constraint, see below)

The Agents of File Transfer and their States (2)

- **message carrier** transmitting msgs bw sender and receiver (here wlog assumed to be known) using an input agent at both sides
  - MESSAGE = DATA × MsgID with static projection fcts data, msgId
- **input**(sender), input(receiver) putting msgs into queue, put by mssg carrier into in(self)

Input ≡ If in(self) ≠ undef

then append in(self) to queue(self)

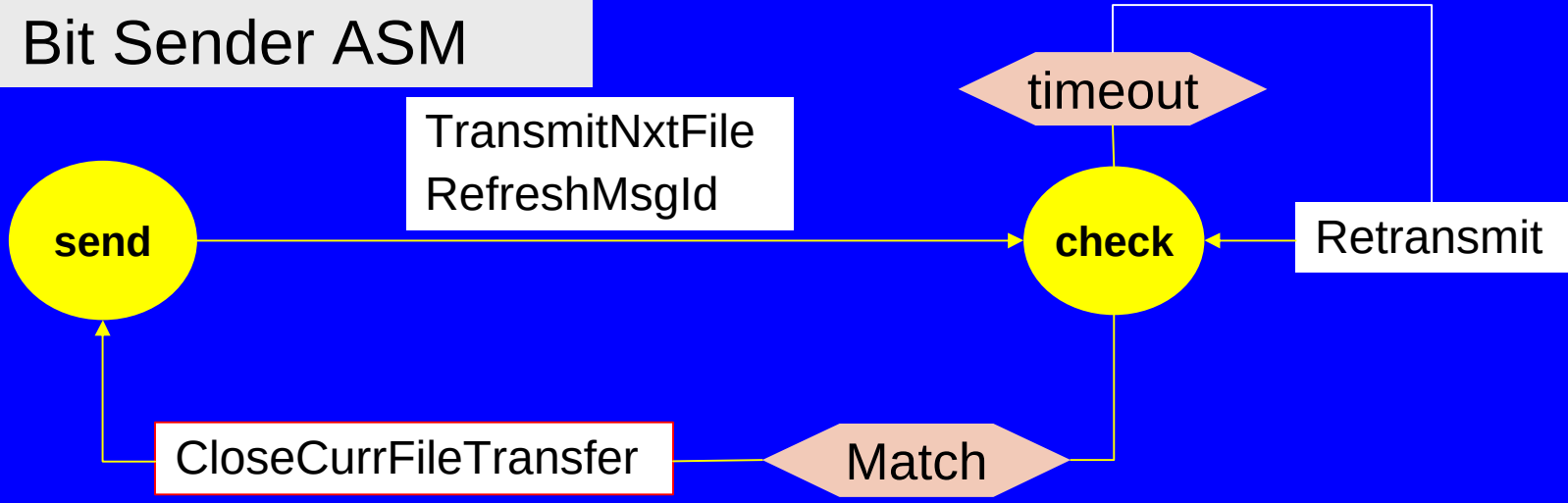
in(self):= undef

NB. queue(self) is shared bw the input agent and its target agent (sender or receiver). The input agents only append items to the end of the queue, sender and receiver only delete items from the beginning of the queue.

Special Functions for Alternating Bit

- sender and receiver have a controlled function bit(self): MsgID, called their alternating bit. Then one can set MsgID={0,1}.
  - An alternative is to treat the alternating bit as derived function bit = currNum mod2, issuing as MsgId currNum mod2. We will do this here and write also bit(m) instead of msgId(m).
  - We write flip for the fct which flips booleans.
  - A msg of form (file(i), i mod2) is called an i-msg, a msg (Null, i mod2) acknowledging the receipt of (file(i), i mod2) is called an i-ack.

Alternating Bit Sender ASM



SendingTime ≡ ctl=send

StartNxtFileTransfer ≡ TransmitNxtFile, RefreshMsgId, ctl:=check

TransmitNxtFile ≡ send (file(currNum+1), currNum+1 mod2) bit is flipped

RefreshMsgId ≡ currNum := currNum+1

CheckingTime ≡ ctl=check & queue ≠ empty

Match ≡ msgId(fst(queue)) = bit CloseCurrFileTransfer ≡ ctl := send

ClearCurrMsg ≡ delete fst(queue) from queue

RetransmitTime ≡ ctl=check & timeout

Retransmit ≡ send (file(currNum), bit), timeout:= false

dyn fcts with sender param

Input can fire independently of sender's ctl state

Alternating Bit Receiver ASM

Receive ≡ If ArrivalTime then
AcknowledgeReceipt, If Match then AcceptCurrFile, ClearCurrMsg

ArrivalTime ≡ queue ≠ empty

AcknowledgeReceipt ≡ send (Null, msgId(fst(queue)))

ClearCurrMsg, Match, RefreshMsgId as for sender
with receiver queue, bit, currNum

AcceptCurrFile ≡ file(currNum) := data(fst(queue)) , RefreshMsgId

Reassuming: The Alternating Bit Protocol ASM consists of the rules

Send , Check, Retransmit of sender Receive of receiver

unknown ("hidden") rules of a mssg carrier agent

Input(sender), Input(receiver) of two input agents

Input ≡ If in(self) ≠ undef then
append in(self) to queue(self), in(self):= undef

Alternating Bit Protocol Initialization

- at both sender, receiver
  - queue = empty
  - in = undef
  - no message is in transit to the agent, say InTransit = empty where by definition InTransit contains all sent, not lost and not yet arrived msgs (in their sending order), including in if in $\neq$ undef
- at sender
  - currNum = 0 = bit (sender) timeout = false
  - ctl = send (i.e. SendingTime is true)
- at receiver
  - currNum = 1= bit (receiver) file(x) = undef for all x

Def. InArrival(self) = InTransit(self) queue(self)

Compare ASM with Petri net for ABP (Fig.27.7) in W. Reisig: Elements of Distributed Algorithms

Recalling Asynchronous Multi-Agent ASMs

- An **Asynchronous Multi-Agent ASM** is a family of pairs (a, ASM(a)) of
  - agents a $\in$ Agent (a possibly dynamic set)
  - basic ASMs ASM (a)
- A **Run** of an asynchronous multi-agent ASM is a partially ordered set (M, $<$) of "moves" m of agents s.t.:
  - finite history: each move has only finitely many predecessors, i.e. {m' | m' $\leq$ m } is finite for each m$\in$ M
  - sequentiality of agents: for each agent a $\in$ Agent, his moves {m | m$\in$ M, a performs m} are linearly ordered by $<$
  - coherence: each (finite) initial segment X of (M, $<$) has an associated state σ (X) – the result of all moves in X with m executed before m' if m<m'– which for every maximal element m$\in$ X is the result of applying move m in state σ(X-{m})

Message Carrier, Timeout, Fairness Constraints on Runs

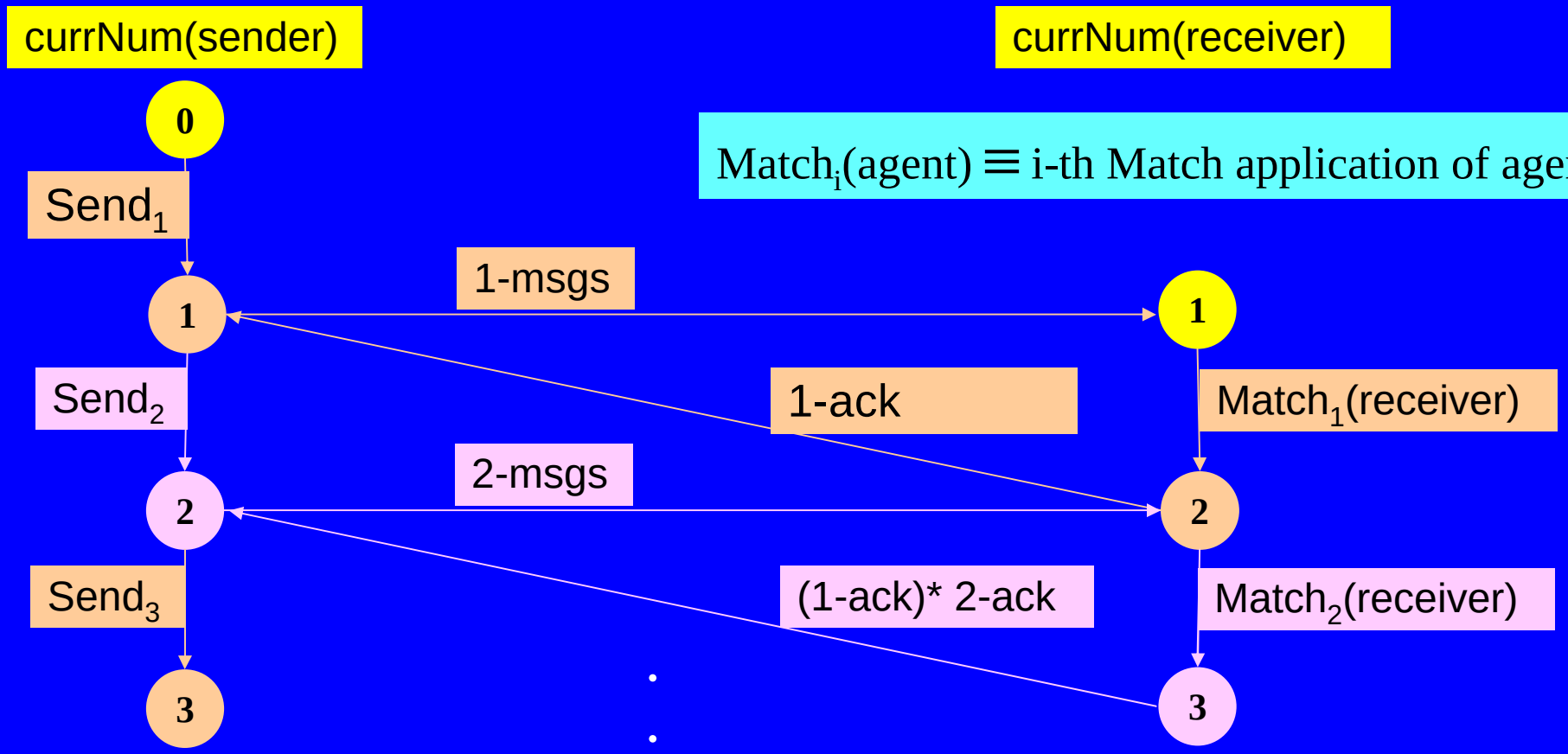
- **Timeout Assumption**: when for both sender and receiver
  - InArrival is empty
  - it is not SendingTime

  then timeout eventually becomes true. (Consider timeout as Δ between currtime and the moment of the last Send or Check action.)
- **Message Order Assumption**: messages which do not get lost during the transmission and are received as input at their target, arrive there one after the other in the order they have been sent.
- **Message Carrier Reliability Assumption**: For every tail ρ of each infinite run, if in ρ an agent applies send infinitely often, some of the msgs sent do not get lost & arrive as input at their target.
- **Agent Fairness Assumption**: in every tail ρ of each infinite run and for every agent other than the message carrier, if the agent is enabled infinitely often in ρ, then it will make a move in ρ.

Runs for Alternating Bit Protocol

- ABP Runs are defined as distributed runs of the sequential ASMs for sender, receiver, input(sender), input(receiver). The runs are assumed to respect
  - the Message Carrier, Timeout, Fairness Constraints, and the initialization defined above
  - the following ordering of corresponding sender/input/receiver moves, i.e. moves which are related by the same instance of a non-lost message m:
    - Send or Retransmit with send m
      comes before the corresponding Input(receiver) with in = m
      comes before the corresponding Receive with fst(queue) = m
    - Receive with fst(queue) = m
      comes before the corresponding Input(sender) with in = m
      comes before the corresponding Check with fst(queue) = m
- For simplicity of exposition we assume the moves to be labeled with their real-time moments

Phases in Alternating Bit Protocol Runs



$Match_i(agent) \equiv$ i-th Match application of agent

$Send_i \equiv$ i-th Send application

(a,b) = {c| not c ≤a & not c ≥ b}

$phase_i$(sender) = all moves from the one when $Send_i$ fires to the one before $Send_{i+1}$ fires

$[Send_i, Send_{i+1}) = [Send_i, Match_i(receiver)) \cup [Match_i(receiver), Match_i(sender)]$

only i-1-acknowledgements are sent

only i-acks are sent

only i-msgs sent: $Send_i$ is enabled or (currNum=i & $Send_{i+1}$ is not enabled

Phase Lemma for Alternating Bit Protocol Runs

- For every i ≥1, when the sender reaches phase i to fire $Send_i$ increasing currNum(sender) to i, the receiver has already moved to phase i, i.e. satisfies currNum(receiver)=i. Thus during sender phase i, receiver is in phase i or i+1.
- In $(Send_i, Match_i(receiver)]$ holds
  - bit and currNum are the same at sender and receiver, so that Match(receiver) fires only when bit(sender) = bit(receiver).
- In $(Match_i(receiver), Send_{i+1}]$ holds
  - bit(sender)≠bit(receiver), currNum(receiver) = currNum(sender)+1 so that Send fires only when bit(sender) ≠ bit(receiver).
- Proof : for i=1 by initialization. Every $Send_{i+1}$ move is preceded by a $Match_i$(sender) move whose i-ack must have been sent by a preceding Receive of an i-msg, which in turn comes after or at the i-th $Match_i$(receiver) move: that move, the only one which changes the receiver phase, brought receiver into phase i+1. From then until the $Send_{i+1}$ move holds bit(sender) ≠ bit(receiver).

InArrival Message Order Lemma for ABP Runs

- During sender phase i ≥1, InArrival(sender) can contain only i-1-acks followed by i-acks.
- During receiver phase i ≥1, InArrival(receiver) can contain only i-1-msgs followed by i-msgs.
- In every state of the run, InArrival(sender)InArrival(receiver) has form
  (i-1-MSG)*(i-MSG)* where MSG:{ack,msg}
  with bit-sequence flip(bit(sender))* bit(sender)*.
- Corollary. When sender has an ack with bit(sender) InArrival, then bit(sender) = flip(bit(receiver)).
  - Proof. Let ack be an i-ack. Then by InArrival Message Order Lemma sender is in phase i or i+1. Upon sending the first i-ack in $Match_i$(receiver), receiver has switched from phase i (by the Phase Lemma) to phase i+1 and flipped its bit from the sender bit. This proves the claim since send is either in phase i or hast just applied $Send_{i+1}$.

Proof of InArrival Message Order Lemma (1)

- Simultaneous ind. on phase changing moves $Send_i$, $Match_i$(receiver).
- Since j-msgs are sent in order j=1,2,3, ..., by the Message Order Assumption they can be received only in this order, so that also j-acks may be sent and arrive only in this order. Therefore when the first i-ack is matched by the sender, only i-acks can be left InArrival(sender), and only i-msgs can still be InArrival(receiver). The next sender move is $Send_{i+1}$, so that until move $Send_{i+2}$ (excluded), msgs newly transmitted to InArrival(receiver), yielding acks which may reach InArrival(sender), can be only i+1-msgs.
- The argument for i-msgs which are InArrival(receiver) and eventually get matched by $Match_i$(receiver) is symmetric.

Proof of InArrival Message Order Lemma (2)

- From the state of $Send_i$ (included) until the state of receiver move $Match_i$ (included) there can be no i-ack InArrival(sender). From the state in which receiver has passed to phase i+1 until $Send_{i+1}$ included, there can be no i+1-msg InArrival(receiver). Therefore in every state during the sender phase i, InArrival(sender)InArrival(receiver) is of form (i-1-MSG)*(i-MSG)* with MSG either ack or msg, therefore its bit-sequence of form flip(bit(sender))* bit(sender)*.

InArrival(receiver)-Data Lemma for ABP Runs

- In every reachable state of the run, InArrival(receiver) msgs contain the data of
  - file(currNum(sender)), if their msgId is bit(sender)
  - file(currNum(sender)) –1, otherwise
- Proof. If in state S, an i-msg m is InArrival(receiver), the receiver is in its phase i or i+1 (by InArrival Message Order lemma) and the phase of sender is i or i+1 (by phase lemma). Therefore if msgId(m) = bit(sender), bit(sender) & therefore currNum(sender) did not change after $Send_i$ until S. Otherwise bw $Send_i$ and S, $Send_{i+1}$ has fired because it is the only rule which changes currNum(sender).

File Transfer Correctness Lemma for ABP Runs

- In every reachable state of the run and for every file number x,
  - file(receiver)(x) = file(sender)(x) if it is defined
  - file(receiver)(x-1) is defined

    if x = currNum(receiver)>1
- Proof. Only AcceptCurrFile in Match(receiver) updates file(receiver), in a state where (by the phase lemma) sender and receiver have the same currNum and bit - which is the bit of the accepted msg, so that the InArrival(receiver)-data lemma implies the claim.
  - Initially currNum(receiver) =1. It is is increased only by RefreshMsgId in AcceptCurrFile as part of Match(receiver).

File Transfer Completeness Lemma for ABP Runs

- Every (data,id) sent by the sender resp. receiver will eventually enter and leave InArrival(receiver) resp. InArrival(sender).
- Every file which is sent by the sender is eventually received, an acknowledgement of its receipt is eventually received by the sender.
- Proof. The first claim follows from the ABP run constraints (on message carrier, timeout, fairness) and from the rules Retransmit, Input, ClearCurrMsg.
- If a file has been sent by $Send_i$, the first time it leaves queue(receiver) its msgId matches bit(receiver) (by phase lemma) so that
  - AcceptCurrFile in $Match_i$(receiver) stores the file at the receiver
  - AcknowledgeReceipt sends the first i-ack.

  Therefore, when the first i-ack eventually leaves queue(sender), it is received by $Match_i$(sender) firing CloseCurrFileTransfer.

Sliding Window: refining msgId, currNum, Match

- MsgID = N (set of file numbers), we write fileNum for msgId
  - (file(i), i) is called an i-msg, (Null, i) is called an i-ack
- currNum(self) is refined to a dynamic window (integer interval) of sent msgs, defined by controlled boundary fcts:
  - high(sender) refining the role of currNum in Send
  - low(sender) refining the role of currNum in Retransmit
  - not windowFull refining SendingTime where
    - winsize = max window size (implementation dependent constant)
    - windowFull ≡ high(sender) – low(sender) +1 = winsize

  NB. The refinement makes Send/Retransmit independent, although wlog we will team them (see below)
- bit-test Match ≡ msgId(fst(queue)) = bit refined to window test low ≤ fileNum(fst(queue)) ≤ high

Receiving Acks & Msgs Refined for Sliding Window

- CloseCurrFileTransfer is refined to an update of a predicate receivedAck to true, namely for fileNum(fst(queue)), thus
  - storing which file transfer messages in the sender window have been succesfully acknowledged, for
  - triggering SlideWindow(sender), when receivedAck(low), to increase low by 1

SlideWindow(sender) ≡

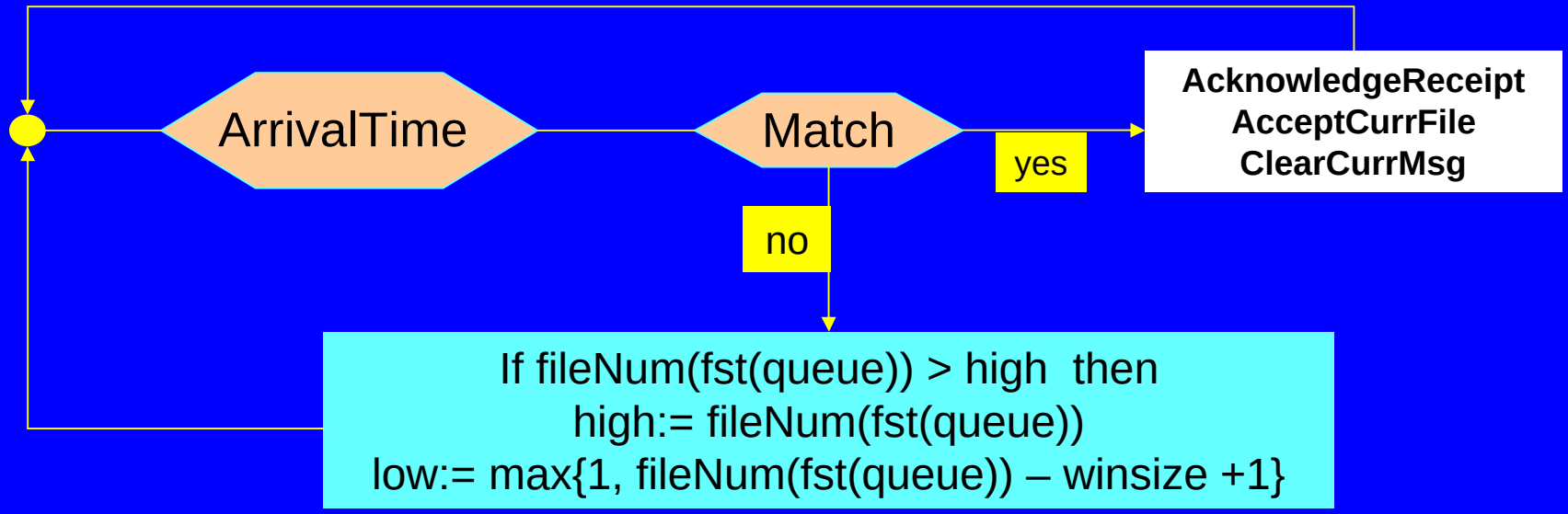
If receivedAck(low) then low := low+1

- A new rule SlideWindow(receiver)
  - upon arrival of a new msg m whose fileNum does not match (yet) the (upper) window bound high(receiver)

slides the receiver window

  - typically by increasing high by 1, so that fileNum(m) will match the receiver window bounds & is accepted

Teaming SlideWindow(receiver) and Receive



NB. Upon the first arrival of a newly transmitted file at the receiver - SlideWindow(receiver) fires (making the arriving file Match) and file(low(receiver)) has been stored in case low(receiver) is slided (due to the file number being >winsize)

Refining Sliding Window Protocol Initialization

- at both sender, receiver
  - InArrival = empty
  - low = 1, high = 0
- at sender
  - timeout = false
  - receivedAck(x) = false for all x
- at receiver
  - file(x) = undef for all x

Compare ASM below with Petri net for SWP (Fig.28.1/2) in W. Reisig: Elements of Distributed Algorithms

Sender ASM: Send, Check, Retransmit, Input, SlideWindow

SendingTime ≡ not windowFull

StartNxtFileTransfer ≡ TransmitNxtFile, RefreshMsgId deleting ctl := check

TransmitNxtFile ≡ send (file(high+1), high+1) replacing currNum

RefreshMsgId ≡ high := high+1 replacing currNum

CheckingTime ≡ queue ≠ empty deleting ctl = check

Match ≡ low ≤fileNum(fst(queue)) ≤high replacg msgId(fst(queue))=bit

CloseCurrFileTransfer ≡ receivedAck(fileNum(fst(queue))):= true
replacing ctl:=send

ClearCurrMsg ≡ delete fst(queue) from queue

RetransmitTime ≡ timeout deleting ctl = check

Retransmit ≡ send (file(low), low), timeout:= false replacing currNum,bit

SlideWindow(sender) ≡ If receivedAck(low) then low := low+1

Refining the Macros for Receiver ASM: Receive, Input(receiver), SlideWindow(receiver)

Receive ≡ If ArrivalTime & Match then
AcknowledgeReceipt, AcceptCurrFile, ClearCurrMsg

ArrivalTime ≡ queue ≠ empty

AcknowledgeReceipt ≡ send (Null, fileNum(fst(queue)))

ClearCurrMsg, Match as for sender

AcceptCurrFile ≡ file(fileNum(fst(queue))) := data(fst(queue))

replacing currNum & deleting RefreshMsgId

SlideWindow(receiver) ≡ If ArrivalTime &
fileNum(fst(queue)) > high then high:= fileNum(fst(queue))
low:= max{1, fileNum(fst(queue)) – winsize +1}

Compare the high-level Petri net formulation of this protocol in Fig.1 (pg. 390) of G. Franceschinis and M.Ribaud: Efficient Performance Analysis Techniques for Stochastic Well-formed Nets and Stochastic-Process Algebras. LNCS 1492 (1998) 386-437

Interleaving subagents for Sliding Window Protocol Runs

- In analogy to SlideWindow(receiver) being teamed with Receive,
  - we exclude independent rules among Send, Check, Retransmit, SlideWindow(sender) to fire simultaneously.

  We interleave the sender rules non-deterministically teaming them into a sender "module" (basic ASM) of form:

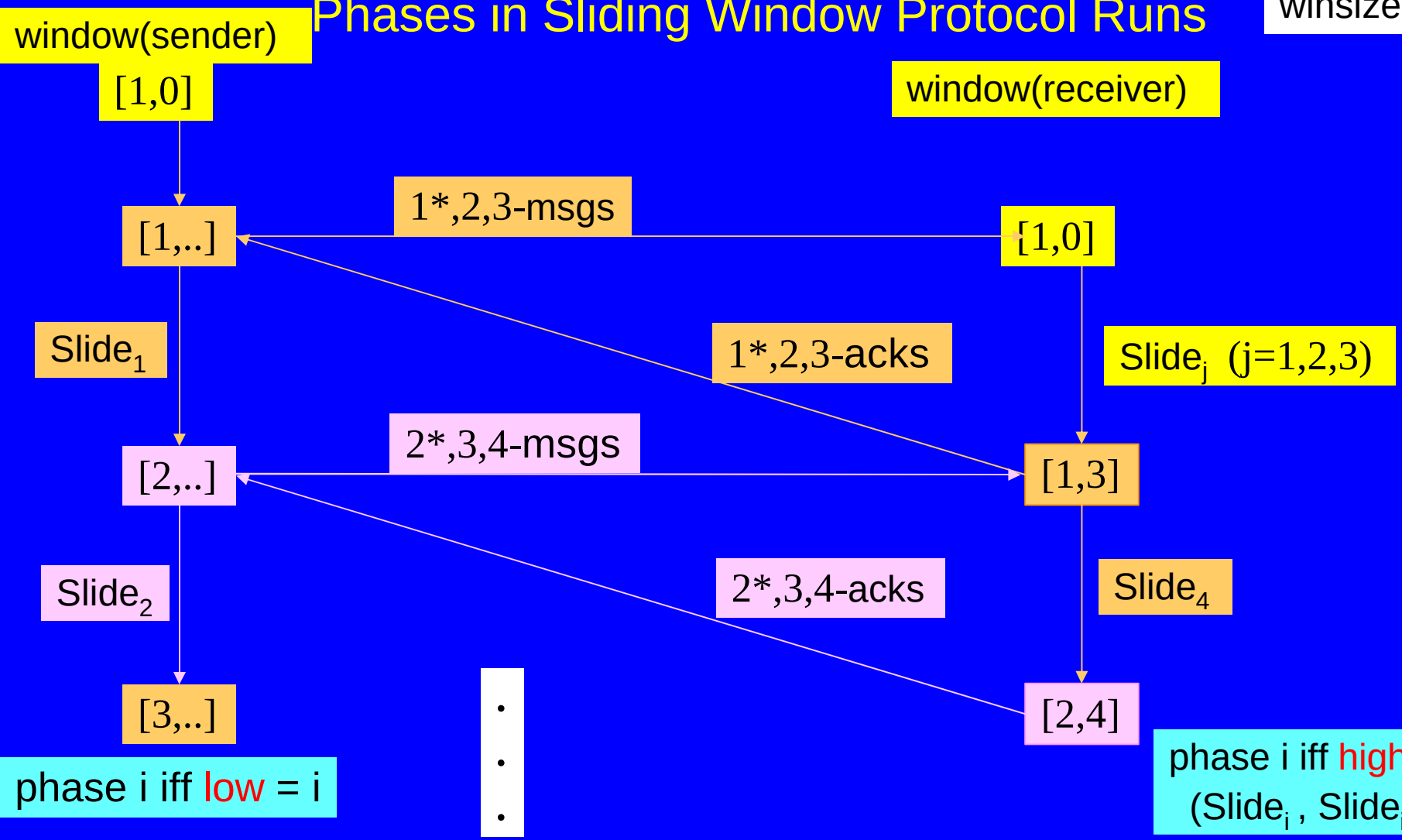
  choose R $\in$ {Send,Check,Retransmit,SlideWindow(sender)}
  in R

  Thus only sender, receiver, the message carrier, and the input devices are considered as independent agents.

  NB. For ABP only the independent rules Retransmit & Check can fire simultaneously.
- The Message Order Assumption can be dismissed for SWP runs since the identity of msgs is preserved by their fileNum.
- The successive increases of low by 1 in SlideWindow(sender) impose successive increases also of the receiver window only if the receiver window is full.

Phases in Sliding Window Protocol Runs

winsize = 3



Phase Lemma for Sliding Window Protocol Runs

- In phase i ≥1, the sender
  - can send finitely many i-msgs and for each 0<j<winsize at most one i+j-msg
  - can fire $Slide_i$ to enter phase i+1 only after the receiver has already made a $Slide_{i+j}$ move for some 0 ≤ j<winsize
    - followed by a Receive move where data(i-msg) = file(sender)(i) is stored in file(receiver)(i) and the first i-ack sent.
- Corollary. In every reachable state of the run and for every file number i,
  - file(receiver)(i) = file(sender)(i) (if it is defined)
  - file(receiver)(i) is defined when phase(receiver) ≥i
- Proof follows by an induction on runs.

File Transfer Completeness Lemma for SWP Runs

- For every (data,id) which is sent by the sender resp. receiver, an instance will eventually enter and leave InArrival(receiver) resp. InArrival(sender).
- Every file which is sent by the sender is eventually received, an acknowledgement of its receipt is eventually received by the sender.
- Proof. The first claim follows from the SWP run constraints (on message carrier, timeout, fairness, move order) and from the rules Retransmit, Input, SlideWindow(receiver), ClearCurrMsg.
- If an i-msg has been sent, by the phase lemma and the first claim at the latest when sender is ready to fire $Slide_i$, receiver has received a copy of an i-msg and stored file(i). When the first i-ack eventually leaves queue(sender), the transfer phase for file(i) terminates by applying CloseCurrFileTransfer which will be followed by an application of the SlideWindow(sender)-rule when low = i.

Building Kermit out of ABP/SWP

- [DaCruz 1987] used the alternating bit and sliding window protocols to define Kermit. Kermit was originally designed for modem communications; it has since been adapted for network communications as well. It is often installed with TCP/IP. It has
  - a session layer controlling sending/receiving files
  - a transport layer making the message carrier reliable via an ABP
  - a datalink layer formatting messages as strings to conform to communication network standards (on msg start marking, its length, number, type, datum, checksum, end): pure data refinement of msg data, types, numbers & related fcts
  - a presentation layer which cuts strings into sequences of short strings (typically of printable ASCII characters) to be sent through the network: a further detailing of the signature (data refinement)
- [Huggins 1995] shows how an ASM for the Kermit protocol can be specified and verified by appropriately reusing alternating bit and sliding window protocol machines and their correctness proofs, which inspired the ASMs and the proofs presented above.

Exercises

- Define the refinement relation which maps ABP runs to SWP runs.
  - Consider in particular the restriction to the case where except for the initially empty sender and receiver interval [1,0], windows always have length 1.
- Optimize the SlideWindow(sender) rule by sliding the window to the nearest position where receivedAck is false and justify the correctness.
- Refine the Sliding Window Protocol ASM by a machine with bound $2 \times$ winsize for message numbers.
  - Hint: Use fileNum(m) mod $2 \times$ winsize as msgId and adapt Match. [Walrand 1991], see also [Huggins 1995].

Exercises

- Show by an example that 2 $\times$ winsize message numbers are necessary for the sliding window protocol [Walrand 1991], see also [Huggins 1995].
- Sharpen the arguments for ABP/SWP correctness & completeness proofs in terms of the partial ordering of moves only, to make them work for asynchronous multi-agent ASMs without labeling moves by real-time moments.
- Adapt the sliding window protocol correctness proof for an asynchronous multi-agent ASM where the sender respectively receiver rules which could fire independently are not teamed.
  - Hint: investigate applications of Retransmit which could be in parallel with applications of SlideWindow(sender).

Exercises

- Show that in every reachable state of runs of the sliding window ASM the following window properties hold:
  - windows never exceed winsize: high – low +1 ≤ winsize always holds at both sender and receiver
  - the receiver window is never ahead of the sender window: low(receiver) ≤ low(sender), high(receiver) ≤ high(sender)
  - once the receiver window gets full, it remains full
  - for files with fileNum<low(sender), the sender has received an acknowledgement: 0 ≤i< low(sender) implies receivedAck(i).

References

- J.-R. Abrial and L. Mussat: Specification and Design of a Transmission Protocol by Successive Refinements Using B
  - Mathematical Methods in Program Development (Ed. M. Broy and B. Schieder), Springer 1996
- K. A. Bartlett, R. A. Scantlebury, P.T.Wilkinson: A note on reliable full-duplex transmission over half-duplex links.
  - Communications of the ACM 12 (5) 1969, 260-261
- F.DaCruz: Kermit: A File Transfer Protocol. Digital Press, 1987
- G. Franceschinis, M.Ribaud: Efficient Performance Analysis Techniques for Stochastic Well-formed Nets and Stochastic-Process Algebras.
  - LNCS 1492 (1998) 386-437 (in particular Fig.1 (pg. 390))

References

- J.K.Huggins: Kermit: Specification and Verification
  - In: Specification and Validation Methods (Ed. E. Börger). Oxford University Press, 1995, 247-293
- E. Börger, R. Stärk: Abstract State Machines. A Method for High-Level System Design and Analysis Springer-Verlag 2003, see http://www.di.unipi.it/AsmBook
- W.Reisig: Elements of Distributed Algorithms. Modeling and Analysis with Petri Nets. Springer 1998, Ch.27,28
- J. Walrand: Communication Networks: A First Course.
  - Aksen Associates 1991
- Kermit Web site http://www.columbia.edu/kermit