Egon Börger (Pisa)

Concurrent Abstract State Machines

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

See E. Börger and K.-D. Schewe: *Concurrent Abstract State Machines*.
Acta Informatica 53 (5), 469-492 (2016)

Goal of the lecture

Motivate and provide the definition of concurrent ASM runs

- Analysis of its relation to:
  - synchronous parallel computation model
  - interleaving model of concurrency
  - concurrency model of sequentially consistent runs (Lamport)
  - concurrency model of partial order ASM runs (Gurevich)
  - Petri net model of concurrency
  - Space-time view of distributed system runs (Lamport)
- Definition of concurrent ASM runs
  - illustrated by Lamport's distributed Mutual Exclusion protocol
  - compared to partial order ASM runs of MUTEXSKELETON
- Concurrency Postulate, Concurrent ASM Thesis and its proof
  - extending Sequential ASM Thesis and proof

NB. Knowledge of the definition of single-agent ASM runs is assumed.

Why concern about concurrent ASMs?

- The concept of sequential (single-agent) algorithm is well understood.
  - It is supported by a computation-model-independent definition, in terms of 3 natural postulates which imply the seq-ASM Thesis
    - formulated and proved by Gurevich in 1984 resp. 2000.
- There are many specific notions of concurrency
  - in the literature, in implementations in current hw/sw systems, in numerous specification and programming languages, etc.

  e.g. distributed algorithms, sequential consistency, process algebras, actor models, trace theory, Petri nets, etc.
- There is no comprehensive or generally accepted concurrency concept
  - despite of numerous extensions of the Sequential ASM Thesis to parallel or bulk synchronous machines, machines interacting during a step with the environment, quantum algorithms, database systems ...

Is there a comprehensive notion of concurrent ASMs to support the practice of computing (and a convincing Concurrent ASM Thesis)?

The question made more precise

Is there a notion of concurrent ASM runs s.t.

- it allows the practitioner to *faithfully model* concurrent real-life systems
  - for a *rigorous analysis* ('ground model' concern)
  - for a *reliable implementation* by stepwise detailing to code ('refinement concern')
- it is comprehensive ('general enough'), i.e. comprises the major concepts of concurrent runs of families of sequential processes which may access a common memory or share info via communication
- it preserves the successful *most general concept of state/change* of sequential ASMs
  - integrating shared data into the control-flow view
- it preserves the *parallel computation model* for single-agent ASMs
  - relegating behaviourally irrelevant sequentialization to refinements

Syntactical definition of Multi-Agent ASMs

A *single-agent* ASM P executes its rule P step by step, sequentially. Where stand-alone, the executing agent a is suppressed notationally.

A multi-agent ASM $\mathcal{M}$ is a family $(pgm_a)_{a \in Agent}$ of single-agent ASMs pgm_a each associated with an agent a which

- executes, *at its own pace*, its rule pgm_a
  - where $a =$ **self** so that different agents may have the same pgm
- interacts with the other agents via reading/writing in designated (input/output or shared) locations

Remark. The set $Agent$ and the program association function $pgm : Agent \rightarrow AsmRule$ may be dynamic

- for run-time creation/deletion of agents and for program changes

NB. One can view the signature of $\mathcal{M}$ as the union Σ (global view) of the signatures Σ_a (local view) of programs pgm_a of $a \in Agent$.

We also write (a, pgm_a) for pgm_a and $ag(P) = a$ if $P = pgm_a$.

SyncAsm behavior is definable by single-agent ASMs

A *synchronous multi-agent ASM* is a multi-agent ASM where the steps of the components are synchronized by a global clock.

Since ASMs support unbounded synchronous parallelism—their syntax permits to use the quantifier **forall**—the behavior of synchronous machines can be defined in terms of the behavior of a single-agent ASM.

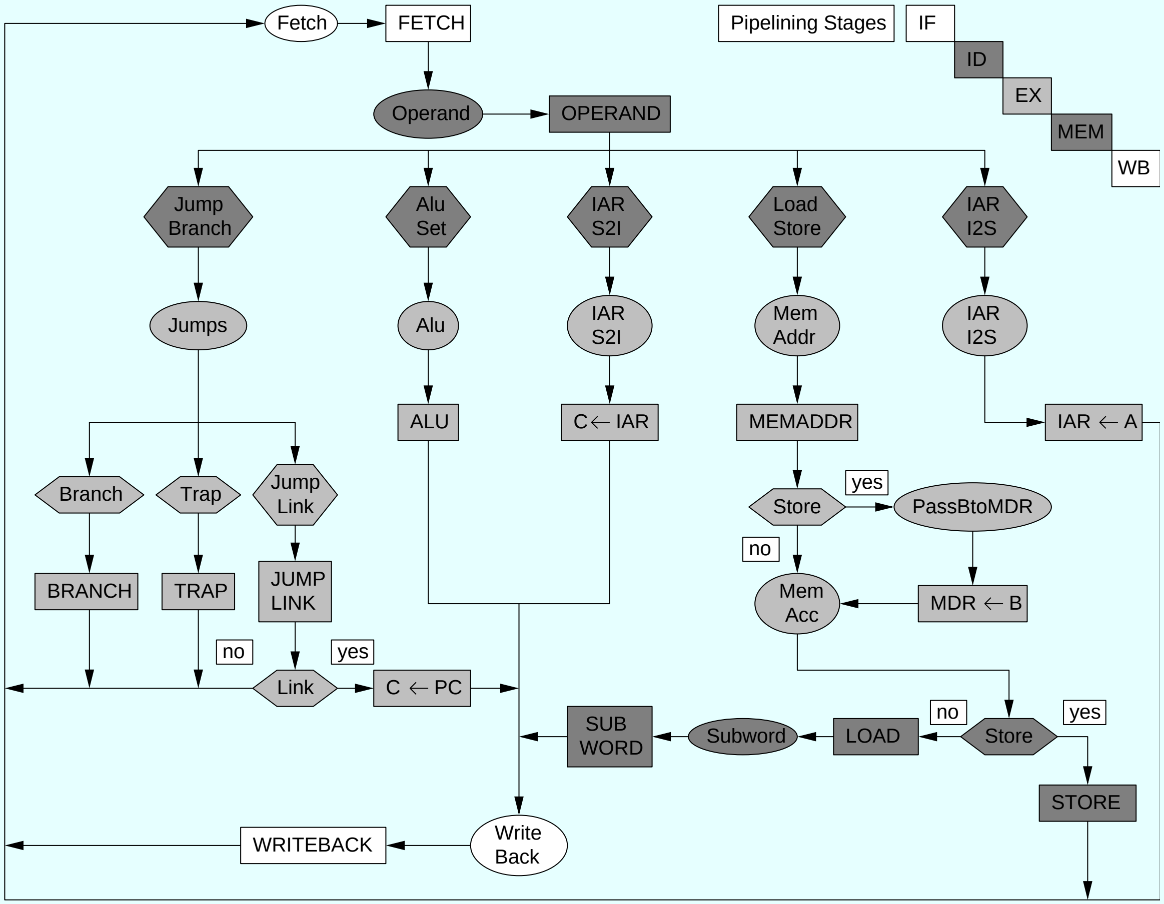
Definition. A synchronous multi-agent ASM is a multi-agent ASM $\mathcal{M} = (a, pgm_a)_{a \in Agent}$ with the following global-clock-governed behavior:

$$\textsc{SyncAsm}(\mathcal{M}) =$$
$$\textbf{forall } a \in Agent \textbf{ do } pgm_a$$

NB. Synchronous multi-agent machines turned out to be useful to rigorously describe parallel processing features.

Example of a SyncAsm: Pipelining of DLX

Serial DLX ground model, a control-state (single-agent) ASM with five successive pipelining stages:[1] IF, ID, EX, MEM, WB

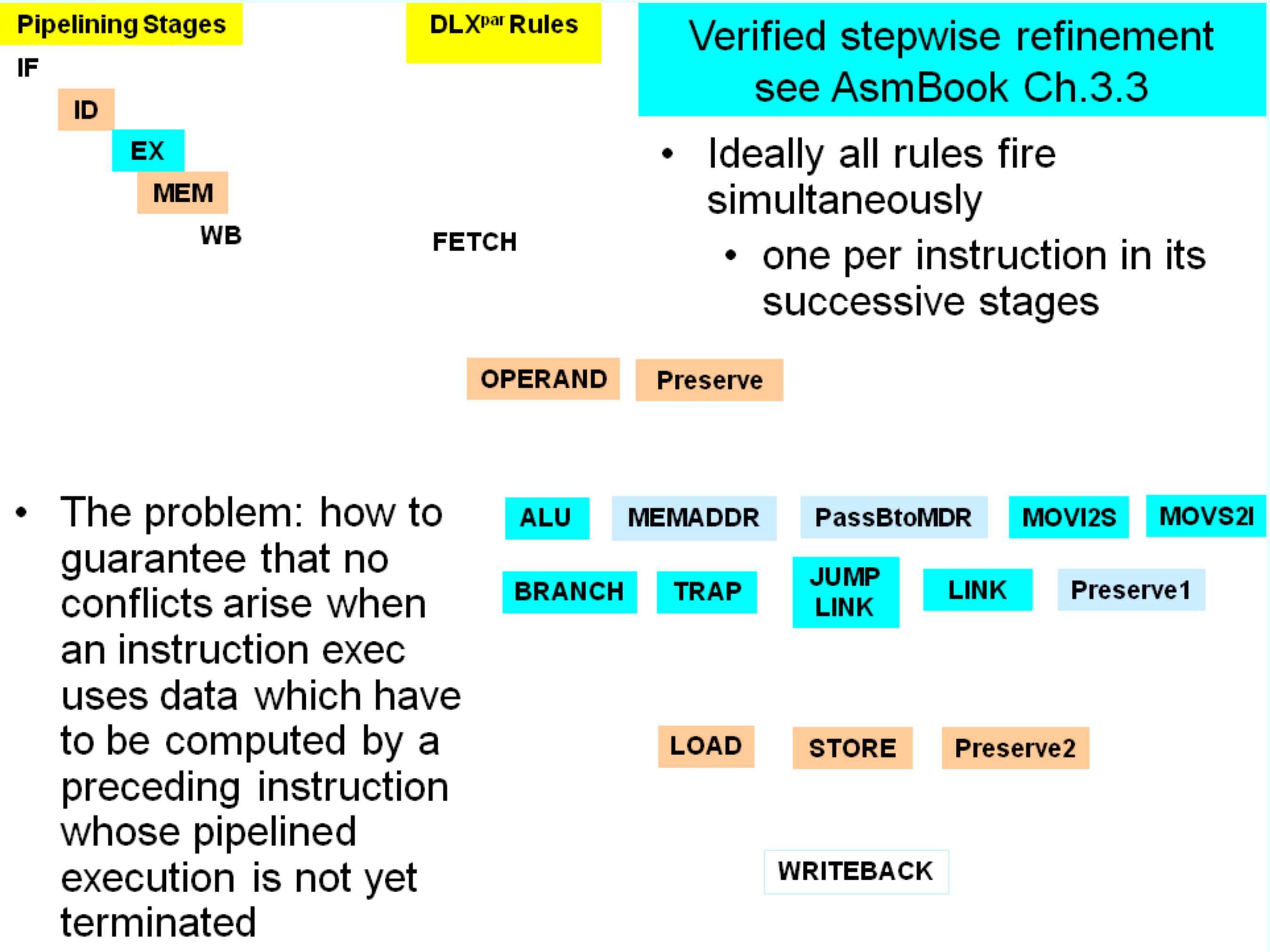


---

[1] Figure from AsmBook, © 2003 Springer-Verlag Berlin Heidelberg, reused with permission

A SyncAsm parallelizing DLX pipelining stages

A refinement relying upon compiler assumptions to avoid conflicts



Pipelining Stages

IF

ID

EX

MEM

WB

DLX^{par} Rules

FETCH

OPERAND Preserve

ALU MEMADDR PassBtoMDR MOVI2S MOVS2I

BRANCH TRAP JUMP LINK LINK Preserve1

LOAD STORE Preserve2

WRITEBACK

Verified stepwise refinement
see AsmBook Ch.3.3

- Ideally all rules fire simultaneously
  - one per instruction in its successive stages

- The problem: how to guarantee that no conflicts arise when an instruction exec uses data which have to be computed by a preceding instruction whose pipelined execution is not yet terminated

Interleaved behavior is definable by single-agent ASMs

Interleaving model of 'concurrency': in each step one agent is chosen to perform a step. Every choice yields a correct interleaving run.

$\textsc{InterleavingAsm}((a, pgm_a)_{a \in Agent}) =$

  choose $a \in Agent$ **do** pgm_a

Characteristics of $\textsc{InterleavingAsm}$-runs R:

- R linearly orders the actions of the ('concurrently' running) agents
- R treats each step of pgm_a as an atomic action
- R appears to each a as single-agent ASM run $View_a(R)$
  - looking at the steps of the other agents in R as environment steps

NB. Single-agent a-runs are defined[2] s.t. after each a-move and before the next a-move the $env(a)$ can make a move to update monitored or shared locations.

---

[2] in the AsmBook and the ModelingCompanion

Single-agent $View_a(R)$ of $\textsc{InterleavingAsm}$ runs R

- let R be a run of $\textsc{InterleavingAsm}((a, pgm_a)_{a \in Agent})$,
- let $m_n(a)$ be the n-th move of a in R and $S_{n,a}$ be the state in R where a makes this move,
- let $M_{n,a} = e_1, \ldots, e_k$ be the (possibly empty) sequence of moves of other agents in R between $m_n(a)$ and $m_{n+1}(a)$ (if $m_{n+1}(a)$ is defined),
- let $env(a)$ be the set $Agent \setminus \{a\}$ of other agents.

The $View_a(R)$ is the single-agent ASM run of a which results from viewing $M_{n,a}$ as the $env(a)$-step which follows $m_n(a)$ in an a-run.

Therefore in every $View_a(R)$-step

$$S_{n,a} \rightarrow_{m_n(a)} S'_{n,a} \rightarrow_{m_n(env(a))} S_{n+1,a}$$

the effect of the n-th move of the $env(a)$ is defined by

$$S'_{n,a} \rightarrow_{e_1} \ldots \rightarrow_{e_k} S_{n+1,a}$$

Lamport's sequential consistency concept for concurrency

Definition. A set of runs of sequential processes is sequentially consistent iff there exists an interleaving of the runs into a single sequence of operations (called a *witness*) that is

> 'a legal execution of a single-processor system, meaning that each value read is the most recently written value'

assuming a definition of the possible (initial) values of a read that precedes any write.

- operations are viewed as pairs $< op, val >$ where $op \in \{read, write\}$ and val is the value read or written by op

Sequential consistency permits witnesses with different results

- but it excludes completely unpredictable concurrent behavior.

NB. Gurevich's *partial order ASM runs are more stringent* (see below)

- providing seq-consistency witnesses but excluding different results.

Exl. for sequential consistency witnesses with different results

- Let $\textsc{RacyWrite}$ consist of agents a_i $(i = 1, 2)$ with rule

```
if $mode_{a_i} = start$ then                 -- controlled 0-ary fct $mode_{a_i}$
    if $f \neq i$ then $f := i$                   -- shared 0-ary fct $f$
    $mode_{a_i} := stop$
```

- Consider the 1-step runs of a_i, started in an initial state of a_i where $mode_{a_i} = start$ and $f \in \{0, 1, 2\}$, consisting of one a_i-move m_i.
- $\{m_1, m_2\}$ is a sequentially consistent set of runs of the two $\textsc{RacyWrite}$-components with two witnesses (runs with **seq**uential move executions) yielding different results:

  m_1 **seq** m_2 yielding $f = 2$ and m_2 **seq** m_1 yielding $f = 1$

NB. *There is no partial order* $\textsc{RacyWrite}$ *ASM run where each agent makes a step* (see the comparison below).

Example without sequential consistency witness: IRIW

- Let IRIW (Independent Read Independent Write) consist of agents a_i $(1 \leq i \leq 4)$ with respective rule
  $x := 1 \mid y := 1 \mid Read(x)$ **seq** $Read(y) \mid Read(y)$ **seq** $Read(x)$
- There is *no sequentially consistent IRIW-run set* where (i) initially $x = y = 0$, (ii) each agent makes once each possible move and (iii) eventually a_3 reads $x = 1, y = 0$ and a_4 reads $x = 0, y = 1$.

Proof: such a run would imply a 'must-come-before' move order $<$:

$x := 1 < (a_3, Read(x))$ -- a_3 should read $x = 1$, initially $x = 0$

$< (a_3, Read(y))$ -- by sequential order of a_3-moves

$< y := 1$ -- since a_3 should read $y = 0$

$< (a_4, Read(y))$ -- since a_4 should read $y = 1$ and initially $y = 0$

$< (a_4, Read(x))$ -- by sequential order of a_4-moves

but then a_4 reads 1, contradicting that it should read $x = 0$.

Extending sequential consistency to ASM runs

- read/write operations $< op, val >$ generalized to ASM steps
- alternate agent (internal) and environment (external) steps in runs

A witness of a sequentially consistent set of ASM runs is defined as

- an interleaving R of the given single-agent a-runs R_a where R_a-steps $m_n(a), m_n(env(a))$ and the corresponding $View_a(R)$-steps yield equivalent updates

This means that for every a and n the sequence $M_{n,a} = e_1, \ldots, e_k$

- of *moves of other agents* in R between the n-th and the $n+1$-th move of agent a in R

yields the same result as the n-th environment move updates in R_a, i.e. such that each step with (not final) move $m_n(a)$ in state $S_{n,a}$

$$S_{n,a} \rightarrow_{m_n(a)} S'_{n,a} \rightarrow_{m_n(env(a))} S_{n+1,a} \rightarrow_{m_{n+1}(a)} S'_{n+1,a}$$

implies $S'_{n,a} \rightarrow_{e_1} \ldots \rightarrow_{e_k} S_{n+1,a}$

Gurevich's partial order ASM runs: the definition

Let $\mathcal{M}$ be a multi-agent ASM. A partial order $\mathcal{M}$-run (called also distributed ASM run or po-run of $\mathcal{M}$) is a partially ordered non-empty set $(M, \leq)$ of *moves* m of its agents $ag(m) \in Agent$ coming with an initial segment function σ that satisfies the following conditions:

finite history: each move has only finitely many predecessors, i.e. $\{m' \in M \mid m' \leq m\}$ is finite for each $m \in M$,

sequentiality of agents: for each agent a the set of its moves in M is linearly ordered, i.e. $ag(m) = ag(m')$ implies $m \leq m'$ **or** $m' \leq m$,

coherence: each finite initial segment I of $(M, \leq)$ has an associated state $\sigma(I)$ —interpreted as the result of all moves in I with m executed before m' if $m < m'$ — which for every maximal element $m \in I$ is the result of applying move m in state $\sigma(I - \{m\})$.

A move m is an application of the agent's ASM rule $pgm(ag(m))$.

NB. We do not restrict programs or agents to finitely many.

Reminder on partial orders

- A *partial order* $\leq$ is a reflexive ($x \leq x$), antisymmetric ($x \leq y \leq x$ implies $x = y$) and transitive ($x \leq y \leq z$ implies $x \leq z$) relation.
- A *partially ordered set* is a set M with a partial order $\leq \subseteq M \times M$.
- $x < y$ denotes $x \leq y$ **and** $x \neq y$.
- A subset I of M is an *initial segment*, if it is downward closed, i.e. if $x \in I$ and $y \leq x$, then $y \in I$.
- $x \in S$ is *maximal* in S iff $y \geq x$ implies $y = x$ for all $y \in S$.
- $x \in S$ is *minimal* in S iff $y \leq x$ implies $y = x$ for all $y \in S$.
- A *linearization* of a partially ordered set M is a sequence $x_0, x_1, \ldots$ (read: an order) of all elements of M which respects the partial order, i.e. such that $x_i \leq x_j$ implies $i \leq j$.

Some simple properties of partial orders

- *Every non-empty finite partially ordered set has minimal elements*.
- *Every po-run M contains minimal moves*.
  - Proof. Let $I_m = \{m' \in M \mid m' \leq m\}$ for some $m \in M$. Since I_m is finite (by the finite history condition), it has minimal elements, which are also minimal in M.
- *Every finite partially ordered set M can be linearized*. Linearizations can be obtained by
  - starting with a minimal element x_0 in M
  - choosing in step $n+1$ a minimal element x_{n+1} in $M \setminus \{x_0, \ldots, x_n\}$
  - until M is exhausted.

Partial order ASM run example IndependentWrite

Consider IndependentWrite with two agents a, b, rule $f(\textbf{self}) := 1$ and initial state $\sigma(\emptyset)$ where $f(a) = f(b) = 0$. Let m_a resp. m_b be a move of a resp. b. The induced partial order is $\{(m_a, m_a), (m_b, m_b)\}$ with initial segments $\emptyset, \{m_a\}, \{m_b\}, \{m_a, m_b\}$ and σ illustrated by:

$$\begin{tikzcd}
 & & f(a) = f(b) = 0 & & \\
 & & \emptyset \arrow[dl] \arrow[dr] & & \\
f(a) = 1 \quad f(b) = 0 & \{m_a\} \arrow[dr] & & \{m_b\} \arrow[dl] & f(a) = 0 \quad f(b) = 1 \\
 & & \{m_a, m_b\} & & \\
 & & f(a) = f(b) = 1 & &
\end{tikzcd}$$

Characteristics of partial order ASM runs

NB. In $\textsc{IndependentWrite}$ the updates $f := 1$ by a resp. b are independent of each other because the two agents a, b come with separate states (i.e. interpretation of f as agent-dependent fcts f_a, f_b).

Let I be any finite initial segment of a partial order ASM run.

- *Linearization Property*.
  All linearizations of I yield runs with the same final state $\sigma(I)$.
- *Unique Result Property*.
  Two partial order ASM runs with same moves and same initial state yield for every finite initial segment the same associated state, i.e. $\sigma(I) = \sigma'(I)$ for each finite initial segment I.
- *Sequential Consistency Property.*
  Each linearization l of I yields a witness for the sequential consistency of the set of sequential I-subruns (one sequential run for each agent which makes a move in I).

Proof of the linearization property

- Let $m_0, m_1, \ldots, m_{n-1}$ be any linearization of I. To show: $S_n = \sigma(I)$ where $S_0 = \sigma(\emptyset)$ and $S_{k+1} =$ the result of performing move m_k in S_k.
- Let $I_k = \{m_0, \ldots, m_{k-1}\}$ for $0 \leq k \leq n$ to stepwise construct I.

Extend $S_0 = \sigma(I_0)$ to $S_k = \sigma(I_k)$ by induction on $0 \leq k \leq n$.

- $I_{k+1} = \{m_0, \ldots, m_k\}$ is an initial segment.
  - Proof. Let $m \leq m_i$ with $m_i \in I_{k+1} \subseteq I$. Then $m \in I$ (initial segment) so that $m = m_j$ for some $j < n$. Thus $m_j \leq m_i$ whereby $j \leq i \leq k$ so that $m = m_j \in I_{k+1}$.
- m_k is maximal in I_{k+1}.
  - Proof. Let $m_k \leq m$ for some $m = m_i \in I_{k+1}$ with $i \leq k$. But $m_k \leq m_i$ implies $k \leq i$. Thus $k = i$ so that $m_k = m_i = m$.
- By induction $S_k = \sigma(I_k) = \sigma(I_{k+1} \setminus \{m_k\})$. Thus state S_{k+1}, result of move m_k in S_k, by the coherence condition equals $\sigma(I_{k+1})$.

Proof of the unique result property

Proof by induction on the initial segment $\mathit{size}(I)$, using the proof for the linearization property.

- Base step $\mathit{size}(I) = 0$: $\sigma(I_0) = S_0 = \sigma'(I_0)$ by assumption.
- Induction step $\mathit{size}(I) = k + 1$: let $I = I^* \cup \{m\}$, an initial segment extending some finite initial segment I^* by a move m.
  - Then m is maximal in I wrt both partial orders, say $\leq$
    - because there is no $m' \in I$ with $m' > m$ (since otherwise $m' \in I^*$ so that the downward closure would imply $m \in I^*$).
  - By induction hypothesis $\sigma(I^*) = \sigma'(I^*)$ so that the coherence condition implies

$$\sigma(I) = \mathit{result}(m, \sigma(I \setminus \{m\})) = \mathit{result}(m, \sigma(I^*))$$
$$= \mathit{result}(m, \sigma'(I^*)) = \sigma'(I \setminus \{m\}) = \sigma'(I)$$

RACYWRITE has no not-single-agent partial order ASM runs

```
RACYWRITE =                                  -- two agents $a_i$ with $i = 1, 2$
  if $mode_{a_i} = start$ then
    if $f \neq i$ then $f := i$              -- NB. $f$ is a *shared* fct
    $mode_{a_i} := stop$
```

- There is no partial order ASM run of RACYWRITE, started in $mode_{a_i} = start$, where each a_i makes a move m_i.
  - As for INDEPENDENTWRITE, initial segments are $\emptyset$, $\{m_1\}$, $\{m_2\}$, $\{m_1, m_2\}$ and every state assignment σ satisfies $f = i$ in $\sigma(\{m_i\})$.
  - m_1 **seq** m_2 and m_2 **seq** m_1 are *two linearizations* of $\{m_1, m_2\}$ *with different final state*.

  Only single-agent 1-step runs are po-runs of RACYWRITE.

Conclusion: There may be multi-agent sequentially consistent runs but no partial order runs.

Characterizing the computational power of po-runs

Consider finitely composed concurrent ASMs (satisfying stipulations Gurevich made for partial order ASM runs), i.e. multi-agent ASMs $\mathcal{C} = (asm_a)_{a \in Agent}$ such that

- each component asm_a is an instance **amb** a **in** P of an ASM rule P from a *finite program base $\mathcal{B}$ of*
  - (possibly non-deterministic) *sequential* ASMs
  - *programs which may create new agents* (potentially infinitely many)
    - via rules **let** $a = \mathbf{new}\ (Agent)$ **in** r
- initially there are only finitely many $Agent$s

Theorem (Börger/Schewe 2019). Finitely composed concurrent ASMs with po-run definable concurrent runs are recursive ASMs.

Conclusion: Since recursive ASMs are a subclass of concurrent ASMs (see references below), partial order ASM runs are not general enough to fully capture true concurrency.

Petri nets: a subclass of non-deterministic sequential ASMs

- static background (finite graph structure $\mathcal{G}$ with $\mathit{Transition}$ rules)
- global state:
  - in classical Petri nets a $\mathit{marking}$ of Places of $\mathcal{G}$ by (possibly coloured) tokens, i.e. a dynamic fct $\mathit{marking} : \mathit{Place} \rightarrow \mathit{Token}$
  - in predicate/transition nets a $\mathit{marking}$ of each place by a set
    - the union of these sets represents the dynamic universe of the net
- one agent with non-deterministic sequential ASM semantics:

$$\textsc{PetriBehavior}(\mathcal{G}, \mathit{Transition}) =$$

$$\textbf{choose}\ T \subseteq \mathit{Transition}\quad \textbf{if}\ \mathit{Enabled}(T)\ \textbf{then}\ \textsc{Fire}(T)$$

  - in the interleaving or partial order view T is a singleton set $\{t\}$
  - in the lock-step or a concurrent ASM view (see definition below) T is a finite set

NB. The static finite set $\mathit{Transition}$ has only finitely many subsets so that $\textsc{PetriBehavior}$ is in fact a *non-deterministic sequential ASM*.

What it means to $\textsc{Fire}$ an $\mathit{Enabled}$ single Petri net transition

The (atomic!) actions to $\textsc{Fire}(t)$ a single $\mathit{Enabled}$ $t \in \mathit{Transition}$ are:

- in classical Petri nets
  - to ‘remove’ from every pre-place $marking(p)$ a specified number $m_{t,p}$ of tokens resp. to ‘insert’ into every post-place $marking(q)$ a specified number $n_{t,q}$ of tokens. Thus rule $\textsc{Fire}(t)$ yields a set of updates:
    - $((marking, p), marking(p) - m_{t,p})$ for each $p \in \mathit{PrePlace}(t)$
    - $((marking, q), marking(q) + n_{t,q})$ for each $q \in \mathit{PostPlace}(t)$
  - if $\mathit{Enabled}(t)$, a conjunction of conditions $m_{t,p} \leq marking(p)$
- in predicate/transition nets a generalization
  - of $+$/- on tokens to union/difference operations on sets
  - of $\leq$ in $\mathit{Enabled}(t)$ to the subset relation

NB. If p is pre-place and a post-place of t, the updates are *partial updates with cumulative effect*.

What it means to $\textsc{Fire}$ an *Enabled* set of Petri net transition

$Enabled(T)$ checks whether all $t \in T$ can be fired simultaneously (in one step), i.e. whether there are enough resources for all of them to fire together without resource access conflict:

- Providing resources as required by a single transition t

  $$\textbf{forall } p \in PrePlace(t)\ m_{t,p} \leq marking(p)$$

  is sharpened to a *cumulative enabling condition* which provides simultaneously all resources required by any transition in T.
  - Let $T_p = \{t \in T \mid p \in PrePlace(t)\}$ (transitions with pre-place p). Then the cumulative enabling condition becomes
    - in traditional Petri nets $\sum_{t \in T_p} m_{t,p} \leq marking(p)$
    - in predicate/transition nets generalized in terms of $\bigcup$ and $\subseteq$

$\textsc{Fire}(T)$ cumulatively applies the partial updates computed by $\textsc{Fire}(t)$ for each $t \in T$.

Claimed special character of Petri net actions

Claimed distinct character of Petri nets, expressed by two objections to 'cumulative location updates' analysis (quotes from a review):

> This concept of "concurrency" is legitimate, but very special. Many models of concurrency, e.g. process algebras and Petri nets, purposely *avoid* notions such as *shared memory* locations and, consequently, the difference between read- and write-conflicts. Petri nets *distinguish* however *access conflicts*, modeled in terms of self loops, and consumption conflicts (alternatives).
>
> On the most elementary level of behavior description, *reading and writing of variables as in ASM fundamentally differs from insertion and removal of tokens as in Petri nets*. This level of description [is] decisive for a precise notion of atomicity, concurrent access to variables, (weak) fairness, etc.

Token insertion/removal are atomic read&write actions

- checking $Enabled$ness reads every involved location $(marking, p)$
  - $Enabled(t)$ reads $(marking, p)$ for each pre-place p of t
  - $Enabled(T)$ reads for each $t \in T$ each $(marking, p)$ for all pre-places p of t
- every removal resp. insertion action writes to every involved location
  - updating it for a single t by a $+/-$ operation or a cumulative partial $+/-$ update
  - updating it for a set T by cumulative partial $+/-$ updates of all locations involved by removal/insertion in at least one $t \in T$

NB. The writings for removal/insertion, concerning possibly multiple pre-/post places, involve also reading those locations and these combined read&write happen simultaneously as ONE atomic action

- same as for—indeed ‘very special‘—single-agent ASMs
  - differently from reads/writes of—much more general!—concurrent ASMs (see below)

Petri net access conflicts = write conflicts of shared locations

- Consider Petri nets as multi-agent systems $(t_a)_{a \in Agent}$.
- Places which are common to multiple transitions $t_{a_i} \in T$ are locations that are *shared* among their agents a_i.
  - The cumulative enabling and update interpretation for concurrent reads of resp. for concurrent writes to shared locations
    - namely by $Enabled(T)$ resp. $\textsc{Fire}(\mathrm{T})$, defined in terms of 'concurrent access to variables'—i.e. simultaneous atomic reads resp. atomic read & write actions for shared $Place$s
      - with 'precise notion of atomicity'

  reflects the 'access conflict' concern governing shared locations which characterizes the behavior of Petri nets.

Conclusion: There is nothing in Petri net actions that 'fundamentally differs' from usual atomic reading resp. updating values of locations.

Instances of nd-sequential ASM PETRIBEHAVIOR

choose $T \subseteq Transition$ **if** $Enabled(T)$ **then** $\textsc{Fire}(T)$

- The definition directly reflects the *lockstep* model where in each step a set of independently fireable (conflict-free) transitions is fired.
- The *Interleaving* model restricts T to singleton sets ($\mid T \mid = 1$).
  - *moves may be partially ordered* (by token-flow, called 'causal dependency') so that incomparable moves can occur in any order
    - or '*occur concurrently*': case of a set T satisfying $Enabled(T)$

Conclusion: Petri net behavior *is not general enough to fully capture true concurrency* (though it may be useful for analyzing control flow issues). Its nd-seq ASM definability is a special case of:

Theorem (Börger/Schewe 2019) For each finite concurrent ASM $\mathcal{C} = (a_i, asm_i)_{i \in I}$ of nd-seq components whose concurrent runs are definable by partial-order runs one can construct an nd-seq ASM $\mathcal{S}_{\mathcal{C}}$ such that the concurrent runs of $\mathcal{C}$ and the runs of $\mathcal{S}_{\mathcal{C}}$ are equivalent.

What is the intuition of truly concurrent runs?

Concur runs are computations of multiple autonomous agents which

- execute each a sequential process,
- run together asynchronously, each with its own clock,
- interact with (and know of) each other only via reading/writing values of designated locations.

Such ‘designated locations’ are input/output or shared locations

- NB. interaction concept should include both synchronous and asynchronous message passing mechanisms
  - sending/receiving appears as writing/reading messages to/from shared or mailbox like locations, e.g.
    - shared channel & data vars in Communicating Action Systems
    - input pool for communication in S-BPM

Concurrent runs = sequences of discrete snapshots of when agents of asynchronously running, autonomous, sequential computations interact

The Concurrency Postulate (Börger/Schewe 2016)

- A concurrent system (or process) is given by a set $\mathcal{A}$ of agents a each of which is equipped with a sequential algorithm $alg(a)$ that is executed autonomously by the agent.
- A concurrent $\mathcal{A}$-run is a sequence $S_0, S_1, \ldots$ of 'interaction states':
  - S_0 is an initial state
  - S_{n+1} is obtained from S_n *by combined single interacting moves* of a finite set A_n of (autonomous!) agents a which
    - started the execution of their current $alg(a)$-move by a read interaction ('receive action') in state S_{j_a} ($j_a \leq n$ depending on a),
    - complete this (otherwise internal, purely local) move by a write interaction ('send action') in state S_n.

NB. *'Combining interacting moves of autonomous agents' in* $S_0, S_1, \ldots$ expresses the concurrent view of multi-agent system runs.

Characteristics of sequential vs concurrent processes

- 3 characteristics of seq algorithmic processes (ASM thesis postulates):
  - states are *structures* (sets of objects with functions of any type)
  - state transformation by *atomic steps*
    - see simultaneous atomic read & write in single-agent ASM steps
  - *resource bound*: steps can depend upon (by a read action) and affect (by a write action) only finitely many terms which represent memory locations and depend only on the process
- in concurrent processes multiple agents may
  - interact via reads/writes of *interaction* (i.e. in/shared/out) locations,
  - perform their reads/writes *asynchronously*, reading in one and writing to another interaction state, each agent at its own speed
    - whereas atomic single-agent ASM read/write steps alternate (are synchronized) with atomic environment read/write steps

Therefore, for concurrent ASM runs we must open the atomicity of single-agent ('sequential') read/write steps.

From single-agent to concurrent ASM read/write steps

- In an atomic *single-agent ASM step*, reading and writing of locations are atomic actions which do not interfer with each other.
- For a concurrent ASM step we *separate atomic reading* (in particular of input/shared locations) *from successive atomic writing* (in particular to output/shared locations)
  - without specific assumptions on the granularity of the read/write ops
  - preserving the capability of ASMs to capture algorithmic computations at any desired (low or high) level of abstraction
- as a result, agents can read in one interaction state
  - e.g. making local copies of interaction data

  and perform the corresponding write in another interaction state
  - e.g. by a writing back step to interaction locations

  unless they perform a simultaneous atomic read/write step
  - which appears in the concurrent run as a global step

Definition of concurrent ASM runs

Let $\mathcal{A}$ be a multi-agent ASM $(a, pgm(a))_{a \in A}$. A concurrent ASM run of $\mathcal{A}$ is a *sequence* $S_0, S_1, \ldots$ *of interaction states* together with a sequence $A_0, A_1, \ldots$ of subsets of agents in A such that

- S_{n+1} is obtained by combining the moves of the agents in A_n which
  - interact by writing some output and shared locations
  - started their current interaction step by reading their monitored and shared locations in some possibly preceding state S_{j_a} (i.e. $j_a \leq n$).
- The run terminates in state S_n if the interaction is incompatible, i.e. if the union of the sets of updates provided by the agents in A_n is inconsistent.

By this definition, a single-agent atomic read/write step, concerning interaction locations of a in a state S, can be split into reading the input and shared locations of a in S, performing local updates and writing back the output and shared locations of a in a later state S'.

Single-agent ASM runs are generalized with read-cleanness

- concurrent ASM runs *generalize single-agent ASM runs*
  - where internal moves, which depend on monitored and shared locations, alternate with updates of those locs by env moves
- concurrent ASM runs guarantee Lamport's *read-cleanness* requirement that even when reading and writing of a location are overlapping, the reading provides a clear value, either taken before the writings start or after they are terminated
  - updates made to a shared or monitored location loc between
    - the state S_{j_a} where agent a interacts with other agents to read this loc to perform its current (possibly only local) move and
    - the next state S_n $(j_a \leq n)$ where a interacts again with other agents, namely to write its output and shared locations, thus completing the execution of its local moves triggered in S_{j_a}

    become visible to a only in state S_{n+1} where it may start its next interaction (not purely local) move

A way to formalize the concurrent ASM run step scheme

A nd-seq ASM $\textsc{ConcurStep}(pgm(a))$ permits agent a to choose

- whether to perform a global step
  - whose updates of globally visible (shared or output) locations are synchronized with globally visible updates by other processes
    - updates which are subject to the consistency constraint

  i.e. to directly perform a global atomic read&write step (including interaction locations) of $pgm(a)$ in the current run state
- or to perform a series of local steps affecting private memory, using local copies for the previously read values of globally visible locations
  - to only later interact again with other processes by
    - writing back to globally visible locations and
    - reading again possibly changed values of monitored/shared locs

  i.e. to trigger performing in the run three consecutive atomic actions:

  read&$\textsc{SaveGlobalData}$, $\textsc{LocalWriteStep}$, $\textsc{WriteBack}$

  asynchronously, in different states (read: at clock ticks of a)

Formula for the concurrent ASM run step scheme

Then one can define S_{n+1} formally

- as obtained from S_n by applying to S_n all the update sets U_a any agent $a \in A_n$ computes in S_n using $\textsc{ConcurStep}(pgm(a))$.

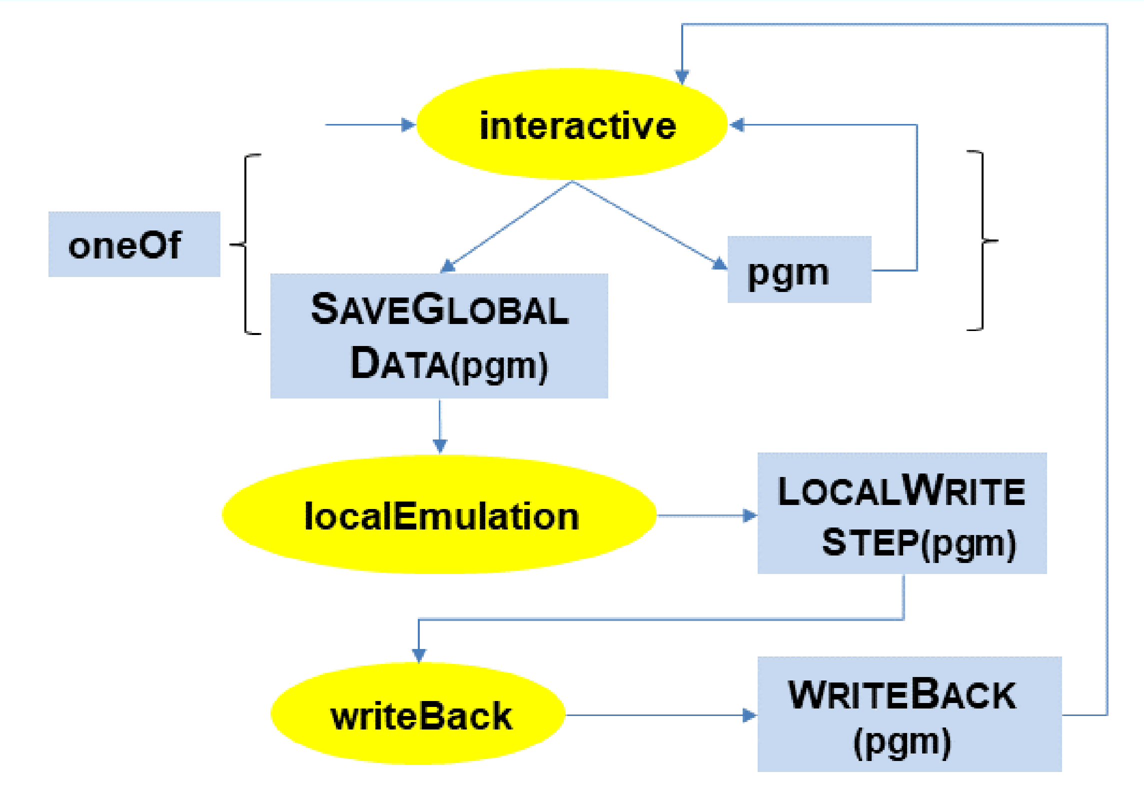
Expressed equationally:

$$S_{n+1} = S_n + \bigcup_{a \in A_n} U_a$$

NB. Here we use bootstrapping: instances $\textsc{ConcurStep}(pgm(a))$ of a single-agent ASM are used to define the behavior of concurrent ASM steps

- computing the updates in S_n either by $pgm(a)$ or by one of the simulation components
  - asynchronously at the agent's clock tick

Three-step version of CONCURSTEP(*pgm*)



- In a 2-step version LOCALWRITESTEP and WRITEBACK are joined into a 1-step execution (eliminating one control state).
- In a multi-step version LOCALWRITESTEP may perform a multi-step subcomputation; only the WRITEBACK-step becomes globally visible.

Interactive and local states in concurrent ASM runs

When $a \in A_n$ contributes to build S_{n+1} by executing one of the $\textsc{ConcurStep}$s of $pgm(a)$, it is in one of three (resp.two) modes:

- in $mode(a) = interactive$, a reads in state S_n the data (also in interaction locs) needed to perform a $pgm(a)$-step. To build S_{n+1}:
  - either a directly computes its update set U_a, in S_n, and applies it to S_n, possibly updating also some interaction locations
  - or a does $\textsc{SaveGlobalData}$ locally, switching to local emulation
- in $mode(a) = localEmulation$, a computes a local copy of U_a, using the saved global data, & switches to $\textsc{WriteBack}$ to interaction locs
- in $mode(a) = writeBack$, a will $\textsc{WriteBack}$ to those (globally visible) interaction locations whose values it has updated locally (by executing an assignment $f_a(s) := t$ in $mode = localEmulation$).

NB. Mathematically, the local updates of $\textsc{SaveGlobalData}$ and $\textsc{LocalWriteStep}$ appear as part of updates of a global state S_n

- other formalizations are possible which hide local updates

Local simulation components of $\textsc{ConcurStep}(a)$

$\textsc{SaveGobalData}(pgm)$ and $\textsc{WriteBack}(pgm)$ transfer values between globally visible interaction functions f (input/shared/out) and new local copies $f_{\textsf{self}}$ which are used for the local simulation.

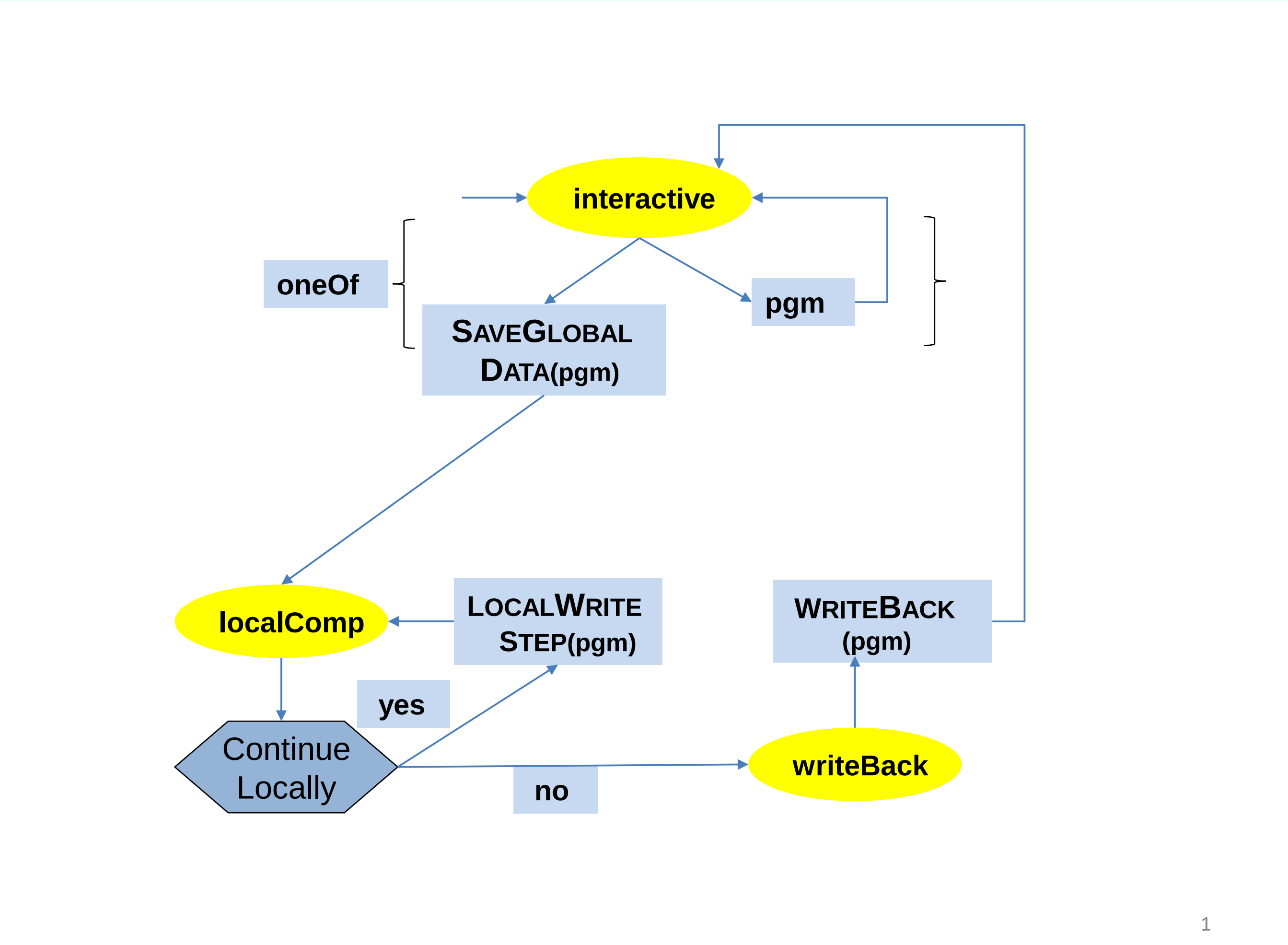
```
$\textsc{LocalWriteStep}(pgm) =$
    $pgm[f/f_a]$                -- replace each interaction fct $f$ by a local copy $f_a$
    add in the modified program in parallel to each $f_a(s) := t$
        $\textsc{Record}(updData(f_a, s, t), GlobalUpd)$
where $a = ag(pgm)$

$\textsc{SaveGobalData}(pgm) =$ forall $f \in Monitored \cup Shared$ $f_a := f$
$\textsc{WriteBack}(pgm) =$
    forall $updData(f_a, s, t) = ((f_a, args), val) \in GlobalUpd$
        $f(args) := val$
    $GlobalUpd := \emptyset$            -- $GlobalUpd$ assumed to be initially empty
```

Two-step version of $\textsc{ConcurStep}(pgm)$

```
choose $P \in \{\textsc{Read\&WriteStep}(pgm), \textsc{ReadStep}(pgm)\}$ do $P$
if $mode = localEmulation$ then
    $\textsc{LocalEmulation}(pgm)$
    $mode := interactive$
– – – – – – – – – – – – – – – – – – – – – – – – –
$\textsc{Read\&WriteStep}(pgm) =$
    if $mode = interactive$ then $pgm$                    -- NB. no $mode$ change
$\textsc{ReadStep}(pgm) =$                                   -- NB. $mode$ change
    if $mode = interactive$ then
        $\textsc{SaveGlobalData}(pgm)$
        $mode := localEmulation$
$\textsc{LocalEmulation}(pgm) =$                             -- emulation of write step
    $\textsc{LocalWriteStep}(pgm)$ seq $\textsc{WriteBack}(pgm)$
```

NB. Turbo ASM operator **seq** guarantees atomic 1-step execution.

Multi-step version of $\textsc{ConcurStep}(pgm)$



NB. One may refine $\textsc{Record}(updData(f_a, s, t), GlobalUpd)$ (or $\textsc{WriteBack}$) to handle also local update overwritings

Concurrent ASM runs in CoreAsm

CoreAsm (an ASM interpreter, see `https://github.com/coreasm`) simulates concurrent ASM runs by trying to select for each 'step' a subset $A \subseteq Agent$ of agents which can make a move: if alltogether they yield an inconsistent update set, another subset of agents is tried out, until all possible combinations haven been tried.

The agent selection can be specified and implemented

- as a pure choice function $select_{Agent}$ (nondeterministic case, currently implemented in CoreAsm) or
- to choose a minimal or a maximal set or
- to choose a set satisfying some priority condition on the agents, etc.

Concurrent ASM Thesis and Proof

Plausibility Theorem (Börger/Schewe 2019). Each concurrent ASM satisfies the Concurrency Postulate.

Characterization Theorem (Börger/Schewe 2019). Each concurrent process $(a, alg(a))_{a \in A}$ which satisfies the Concurrency Postulate can be lock-step simulated by a concurrent ASM $(a, P_a)_{a \in A}$, i.e. s.t. the sets of agents and their steps in corresponding runs are in a 1-to-1 relation.

The proof of the characterization theorem uses Gurevich's proof of the Sequential ASM Thesis for the components of concurrent algorithms:

- **Abstract State**: states are algebraic structures.
- **Sequential Time**: successor states are computed by applying an atomic-step function.
- **Bounded Exploration**: each step is determined by the interpretation of a finite set of terms which depends only on the algorithm.

Space-Time view of distributed system runs (Lamport)

Space-time view of distributed multiprocess system runs

- ‘allows one to implement any desired ... multiprocess synchronization’ without any central synchronizing process or central storage
  - s.t. one can

    derive global concurrent run info from local observations

    in the sequential component runs of a multi-agent system $\mathcal{P} = (P_i)_{i \in I}$ of sequential processes P_i
    - for example in a distributed mutual exclusion protocol that a $\mathit{ResourceCanBeGrantedTo}$ a process

The space-time view yields concurrent ASM runs.

In contrast, *axiomatically specified partial-order ASM runs* may not exist at all, and if they exist it may be hard to find ways to compute them.

- We illustrate this here by comparing the $\textsc{MutexLamport}$ algorithm with a corresponding $\textsc{MutexSkeleton}$ ASM.

Idea underlying the space-time view of concurrent runs

Idea: *parameterize component actions by* a logical time parameter —a local $stepCount_{P_i}$ for each sequential component P_i—s.t.

- the local clock ticks at each component step
  - reflecting P_i's sequential execution order
- clock values of corresponding component moves $Send/Receive(msg)$ are linked to **locally reflect** the following stipulation:
  - $Send(msg)$ by P_i to P_j **HappensBefore** $Receive(msg)$ by P_j

The link is expressed by the following ClockCondition for moves m, m' of any components $ag(m)$, $ag(m')$:

- $m\ HappensBefore\ m'$ implies

$$stepCount_{ag(m)}(m) < stepCount_{ag(m')}(m')$$

The ClockCondition can then be used to make $HappensBefore$ locally observable.

ExI: concurrent ASM runs for space-time view of Mutex

Let $\mathcal{P}$ be a distributed system

- of *spatially separated component processes* P_i $(i \in I)$
- which *communicate* with one another by sending/receiving messages
  - in particular to request/release an exclusive resource executing steps of a mutual exclusion protocol MutexLamport with 5 actions:
    - RequestResource and ReleaseResource
    - FetchMsgs (transfer from $mailbox$ to local memory)
    - ReceiveRequestMsg and ReceiveReleaseMsg.

This *exclusively grants a resource* to component processes P_i s.t.

- every RequestResource by P_i is eventually granted to P_i if every process that has been granted the resource eventually relases it
- whenever the resource has been granted to P_i, P_i must ReleaseResource before it can be granted to another P_j
- requests by different processes are granted in the order they are made

Algorithmic idea for MutexLamport

In a concurrent run of MutexLamport, every component process P_i:

- may at certain points in a run RequestResource
  - by sending its *req*uest to every other process
- observes by ReceiveRequestMsg and ReceiveReleaseMsg steps all resource requests/releases made by other processes until
  - P_i in local memory $HasSeenAllReqsCompetingWith(req)$
  - P_i sees in loc mem that $SatisfiedAllReqsCompetingWith(req)$
- is assumed to eventually ReleaseResource
  - including to send a release message to every other process

  each time it has been granted the resource

Let $HappensBefore$ be the transitive closure of the (union of the) linear orders of P_i-steps and the partial order where a $Send(msg)$-step of sender P_i precedes the corresponding $Receive(msg)$-step of receiver P_j.

How to locally implement the ClockCondition

$m\ HappensBefore\ m' \Rightarrow stepCount_{ag(m)}(m) < stepCount_{ag(m')}(m')$

- for P_i-moves m, m': use $\textsc{Increment}(stepCount_{P_i})$ at every P_i-step
- for corresponding $Send/Receive(msg)$ moves m, m':
  - associate every $msg \in Msg$ with a 'logical' $sendTime(msg)$
    - namely $stepCount_{ag(m)}(m)$ when move $m = Send(msg)$ is made
  - for each msg found in $mailbox_{receiver}$
    - namely by a $receiver$'s $\textsc{FetchMsgs}$ move m

    $\textsc{Adjust}\ stepCount_{receiver}(m)$ to a value $> sendTime(msg)$
  - define the proper $Receive(msg)$ moves as $\textsc{ReceiveRequestMsg}$ or $\textsc{ReceiveReleaseMsg}$ step for a msg $\textsc{FetchMsgs}$ has transfered to local memory, $\textsc{Adjust}$ing $stepCount_{receiver}$

This makes $Send(msg)\ HappensBefore\ Receive(msg)$ locally observable in $\textsc{Receive}[\textsc{Request}/\textsc{Release}]\textsc{Msg}$ steps by $stepCount_{sender}(Send(msg)) < stepCount_{receiver}(Receive(msg))$

How to locally compute $\mathit{SatisfiedAllReqsCompetingWith}(req)$

- Each P_i manages a set ReqMsg of (sent or received and not yet released) request messages, i.e. msg of $type(msg) = request$.
- ReqMsg must be linearly ordered by a locally observable order
  - the order of requests by their $\mathit{sendTime}(req)$ is known locally
  - to *order also concurrent req moves* (not ordered by $\mathit{HappensBefore}$) use any static linear order $<_{\mathcal{P}}$ every process knows
- NB. Similarly one can linearize the entire $\mathit{HappensBefore}$.

$req\ \mathit{GoesBefore}\ req'$ **iff**

$$\mathit{sendTime}(req) < \mathit{sendTime}(req') \textbf{ or}$$
$$(\mathit{sendTime}(req) = \mathit{sendTime}(req')$$
$$\textbf{and } sender(req) <_{\mathcal{P}} sender(req'))$$

$\mathit{SatisfiedAllReqsCompetingWith}(req)$ **iff**

$$\textbf{forall } req' \in \mathit{ReqMsg} \setminus \{req\} \quad req\ \mathit{GoesBefore}\ req'$$

Assumptions on communicating agents

- Each P_i communicates directly with each other P_j.
  - i.e. $\textsc{Send}(msg, \textbf{to } P)$ eventually leads to $msg \in mailbox_P$.
- Message passing is reliable:
  - No message loss: every sent message is eventually $Received$ by the destination process (via a $\textsc{FetchMsgs}$ step).
  - No message overtaking: messages are received in sending order
    - more precisely: msgs sent by P_i to P_j are $Received$ by P_j in their sending order.
- Component actions of $\textsc{MutexLamport}$ are atomic.
  - Thus read/write steps need not be separated in concurrent runs. Only $mailbox$ operations Send/Receive are interactive.
    - Communication agents are abstracted away (wlog).
- Every enabled $Receive(msg)$ move is eventually performed.

For defining $HasSeenAllReqsCompetingWith(req)$ we exploit that msgs are not lost and furthermore received in sending order.

How to locally compute $HasSeenAllReqsCompetingWith(req)$

- We say that a $request$ of P_i $competes$ with a req' of another P_j if req' $GoesBefore$ req, in which case req' has been made before or at the same time as req (i.e. $sendTime(req') \leq sendTime(req)$.
- Therefore, when P_i has a pending $req \in ReqMsg$, this req does not compete with any $laterReq$ sent by P_j, i.e. timestamped with $sendTime(laterReq) > sendTime(req)$.
- Since msgs are $Received$ in their sending time order, this implies that P_i knows its req-competitors from P_j once it has $Received$ a $laterMsg$ from P_j.

Competitor Knowledge Lemma. When P_i has a pending $req \in ReqMsg$ and has $Received$ from another P_j any $laterMsg$

- i.e. with $sendTime(laterMsg) > sendTime(req)$

then in some preceding states P_i must have seen—$Received$ and INSERTed into $ReqMsg$—any req' from P_j its own req competes with.

Defining $\mathit{ResourceGrantedTo}$

The Competitor Knowledge Lemma can be applied if

- after sending a request, P_i receives from each other P_j a $\mathit{laterMsg}$.

To guarantee this, we let each ReceiveRequestMsg move send a timestamped ackowledgment to the sender of the request (see below).

$\mathit{HasSeenAllReqsCompetingWith}(\mathit{req})$ **iff**

  forall $P \in \mathcal{P} \setminus \{$**self**$\}$ **thereissome** msg **with** $\mathit{sender}(\mathit{msg}) = P$

    and $\mathit{Received}(\mathit{msg})$ **and** $\mathit{sendTime}(\mathit{msg}) > \mathit{sendTime}(\mathit{req})$

$\mathit{ResourceGrantedTo}(P)$ **iff**

  forsome $\mathit{req} \in \mathit{ReqMsg}$ **with** $\mathit{sender}(\mathit{req}) = P$ -- an own req

    $\mathit{HasSeenAllReqsCompetingWith}(\mathit{req})$ **and**

      $\mathit{SatisfiedAllReqsCompetingWith}(\mathit{req})$

Recap $\textsc{MutexLamport}$ signature

- $Msg =$ initially empty set of messages created by protocol actions
  - msg content comprehends $sender$, $sendTime$ and $type$, formally:

    $sendTime : Msg \to NAT$

    $type : Msg \to \{request, release, ack\}$

    $sender : Msg \to \{P_i \mid i \in I\}$
- Controlled functions (instantiated for each P_i by **self** $= P_i$)

  $stepCount : NAT$ -- step counter, assumed to be initially ≥ 0

  $mailbox \subseteq Msg$ -- assumed to be initially empty

  $ReqMsg \subseteq \{m \in Msg \mid type(m) = request\}$ -- initially empty
- Auxiliary sets used by $\textsc{FetchMsg}$ to internally store $mailbox$ content

  $Received \subseteq Msg$ -- assumed to be initially empty

  $MsgForProperReceive \subseteq Msg$ -- assumed to be initially empty
- static orders $<_{\mathcal{P}}$ of components P_i and $<$ of NATural numbers and static set $\mathcal{P}$.

Defining $\textsc{FetchMsgs}$

```
forall $m \in mailbox$
    $\textsc{Insert}(m, Received)$                    -- transfer to local memory
    $\textsc{Adjust}(stepCount, mailbox)$
                    -- to become $> sendTime(m)$ for every $m \in mailbox$
    if $type(m) \neq ack$ then $\textsc{Insert}(m, MsgForProperReceive)$
    $\textsc{Delete}(m, mailbox)$            -- usual meaning of $\textsc{Insert}/\textsc{Delete}$
where
$\textsc{Adjust}(stepCount, mailbox) =$
    $stepCount := 1 + max(stepCount, maxSendTime(mailbox))$
$maxSendTime(mailbox) =$
    $max(\{sendTime(m') + 1 \mid m' \in mailbox\})$
```

This rule guarantees the ClockCondition for $Send/Receive(msg)$ moves.

Defining RequestResource

```
$\textsc{RequestResource} =$
  $\textbf{let}\ req = \textbf{new}\ (Msg)$
    $\textsc{DecorateAsReq}(req)$
    $\textbf{forall}\ P \in \mathcal{P} \setminus \{\textbf{self}\}\ \ \textsc{Send}(req, \textbf{to}\ P)$
    $\textsc{Insert}(req, ReqMsg)$
    $stepCount := stepCount + 1$
  $\textbf{where}$
    $\textsc{DecorateAsReq}(m) =$
      $type(m) := request$
      $sender(m) := \textbf{self}$
      $sendTime(m) := stepCount$
```

$\textsc{Send}(m, \textbf{to}\ P)$ triggers $\textsc{Insert}(m, mailbox_P)$ to eventually happen.

Defining ReleaseResource

```
$\textbf{let } msg = \textbf{new } (Msg)$
    $\textsc{DecorateAsRelease}(msg)$
    $\textbf{forall } P \in \mathcal{P} \setminus \{\textbf{self}\} \ \textsc{Send}(msg, \textbf{to } P)$
    $\textsc{DeleteOwnReqs}(ReqMsg)$
    $stepCount := stepCount + 1$
$\textbf{where}$
    $\textsc{DecorateAsRelease}(m) =$
        $type(m) := release$
        $sender(m) := \textbf{self}$
        $sendTime(m) := stepCount$
    $\textsc{DeleteOwnReqs}(ReqMsg) =$
        $\textbf{forall } req \in ReqMsg \textbf{ with } sender(req) = \textbf{self}$
            $\textsc{Delete}(m, ReqMsg)$
```

Defining ReceiveRequestMsg

```
$\textsc{ReceiveRequestMsg} =$
  forall $msg \in MsgForProperReceive$ with $type(msg) = request$
    $\textsc{Insert}(msg, ReqMsg)$
    $\textsc{Acknowledge}(msg)$
    $stepCount := stepCount + 1$
    $\textsc{Delete}(msg, MsgForProperReceive)$
  where
    $\textsc{Acknowledge}(msg) =$
      let $m =$ new $(Msg)$
        $\textsc{DecorateAsAck}(m)$
        $\textsc{Send}(m,$ to $sender(msg))$
```

Defining $\textsc{ReceiveReleaseMsg}$

```
$\textsc{ReceiveReleaseMsg} =$
  forall $msg \in MsgForProperReceive$ with $type(msg) = release$
    $\textsc{DeleteSenderReqs}(ReqMsg)$
    $stepCount := stepCount + 1$
    $\textsc{Delete}(msg, MsgForProperReceive)$
  where
    $\textsc{DeleteSenderReqs}(ReqMsg) =$
      forall $req \in ReqMsg$ with $sender(req) = sender(msg)$
        $\textsc{Delete}(req, ReqMsg)$
```

Defining the $\textsc{MutexLamport}$ program

We call a process P equipped with $\textsc{MutexLamport}$ if P executes a sequential $asm(P)$ which

- increments $stepCount_P$ at each step
- may use ('call') any rule of its instance of $\textsc{MutexLamport}$

$$\textsc{MutexLamport} =$$
$$\{\textsc{RequestResource}, \textsc{ReleaseResource}, \textsc{FetchMsgs},$$
$$\textsc{ReceiveRequestMsg}, \textsc{ReceiveReleaseMsg}\}$$

Correctness Property. Let $\mathcal{P}$ be a multi-agent ASM of processes P_i equipped with $\textsc{MutexLamport}$. Then

- every concurrent ASM run of $\mathcal{P}$ satisfies the above mutual exclusion requirements.

NB. Different linearizations $GoesBefore$, applying different $<_{\mathcal{P}}$ to the partial order $HappensBefore$, yield different resource grant results.

MutexLamport **correctness proof (1)**

To show: *Every* RequestResource *is eventually granted*.

- Let P_i execute RequestResource, putting a request msg into $ReqMsg$ and sending it to every other process P_j.
- Since no msg is lost, in some later state S P_i has $Received$ from every other P_j an $ackMsg$ with $sendTime(ackMsg) > sendTime(req)$, so that P_i in state S $HasSeenAllReqsCompetingWith(req)$.
- By Competitor Knowledge Lemma in some preceding states, P_i has Inserted into $ReqMsg$ every req' its own req competes with.
- If in S there is no such $req' \in ReqMsg$, then by its definition $ResourceGrantedTo(P_i)$ is true in S.
- By induction, when $ResourceGrantedTo(sender(req'))$, by assumption $sender(req')$ will eventually ReleaseResource so that eventually $ResourceGrantedTo(P_i)$ becomes true (via the rule ReceiveReleaseMsg).

$\textsc{MutexLamport}$ correctness proof (2)

To show: *Whenever the resource has been granted to P_i, P_i must* $\textsc{ReleaseResource}$ *before it can be granted to another P_j.*

- Assume $ResourceGrantedTo(P_i)$ and $ResourceGrantedTo(P_j)$ in S.
- Then by definition both processes have an own $req_i \in ReqMsg_{P_i}$ resp. $req_j \in ReqMsg_{P_j}$ that $GoesBefore$ any other req' in their $ReqMsg$.
- Since $GoesBefore$ is a total order of request messages, either req_i $GoesBefore$ req_j or req_j $GoesBefore$ req_i.
- Case 1. req_i $GoesBefore$ req_j. Then req_j competes with req_i (by definition) so that by the Competitor Knowledge Lemma, in some state preceding S, P_j has inserted req_i into $ReqMsg_{P_j}$.
- Since by assumption P_i in state S did not yet $\textsc{ReleaseResource}$, $req_i \in ReqMsg_{P_j}$ still holds in S. Therefore req_j $GoesBefore$ req_i (because $SatisfiedAllReqsCompetingWith(req_j)$), a contradiction.
- Case 2. Symmetric in i, j.

MutexLamport **correctness proof (3)**

To show: *Requests by different processes are granted in the order they are made.*

- Let req_i, req_j be made by P_i resp. P_j with $req_i\ GoesBefore\ req_j$ and without any other request between these two.
- Let $ResourceGrantedTo(P_j)$ be true for req_j in some state S and $ResourceGrantedTo(P_i) = true$ for req_i in another state S'.
- Since req_j competes with req_i, by the Competitor Knowledge Lemma P_j has inserted req_i into $ReqMsg_{P_j}$ in some state before S.
- Since $SatisfiedAllReqsCompetingWith(req_j)$ holds in S (by assumption), in this state $req_i \notin ReqMsg_{P_j}$ must be true.
- req_i can have been deleted from $ReqMsg_{P_j}$ only by a ReceiveRelaseMsg step of P_j, which must have been triggered by a ReleaseResource step of P_i before state S.

MUTEXLAMPORT vs partial order runs of MUTEXSKELETON

MUTEXLAMPORT provides a scheme to *compute local behavior which satisfies the given 'global' state/concurrent run requirement*.

MUTEXSKELETON =

```
if $owner = none$ then $owner :=$ self        -- GRAB move
if $owner =$ self then $owner := none$        -- RELEASE move
```

- for a SYNCASM of components with the MUTEXSKELETON rule
  - rule guards exclude to GRAB the resource when it has an *owner*
  - but if two processes simultaneously try to GRAB the available resource, the run simply terminates (update consistency condition)
- in po-runs of multiple agents with MUTEXSKELETON rule, the coherence axiom excludes independent (not-ordered) GRAB moves
  - by restricting local steps to be compatible with the global state req

  but by itself provides no insight how to *compute* the postulated global po-initial-segment states from local component behavior.

Analysis of po-runs helps to compute initial segment states

A detailed analysis of po-runs may disclose how one can use the partial order to compute the states postulated by the coherence axiom.

We illustrate this here for the $\textsc{MutexSkeleton}$ ASM.[3]

- Observation: independent (not-ordered) $\textsc{Grab}$ moves permit linearizations of initial run segments with different resulting states, i.e. different $owner$ values.
- Therefore, to exclude different resulting states (read: $owner$ values)
  - not only every not-last $\textsc{Grab}$ move m must come in pair with a $\textsc{Release}$ move m' of the same agent
    - as guaranteed already by the rule guards
  - but furthermore each other such $\textsc{Grab}/\textsc{Release}$ pair must come entirely either before m or after m'

Then, the *global segment states can be computed following the total order the coherence condition imposes on* $\textsc{MutexSkeleton}$ *moves*.

---

[3] We follow the analysis Robert Stärk presented in his lectures at ETH Zürich in 2004.

The given multi-agent ASM with $\textsc{MutexSkeleton}$ rule

In the following

- let $\mathcal{M}$ be any multi-agent ASM where each agent has the rule $\textsc{MutexSkeleton}$
  - possibly also other rules which however do not affect the values of the $owner$ location, though they may depend on them
- let $(M, \leq)$ be a partial order of $\textsc{MutexSkeleton}$-moves of agents of $\mathcal{M}$ which satisfies Finite History and Sequentiality of Agents
  - to simplify the wording of the analysis we disregard the effect of other local moves because by assumption they do not change the global state part—here the location $owner$—the global behavior requirement is about
- consider runs where initially $owner = none$

Revealing the MutexSkeleton move order in po-runs

The coherence axiom can be satisfied for $(M, \leq)$ iff

there is a strictly ordered sequence $m_0, m_1, \ldots$ of moves in M s.t. for all $i = 0, 1, \ldots$ the following holds:

- monotonicity: m_0 is the least move in M and $m_i < m_{i+1}$
- Grab/Release come in pairs: m_{2i}/m_{2i+1} are Grab/Release moves of a same agent $ag(m_{2i})$
- between moves of a Grab/Release pair and between a Release and the next Grab there is no other move in M
- all moves $m \in M$ are ordered wrt m_i, i.e. $m \leq m_i \vee m_i < m$
- last move condition (in case there is a last move):
  - if m_{2n} is the last element in the sequence, then each move $m > m_{2n}$ is made by another $ag(m) \neq ag(m_{2n})$
  - if m_{2n+1} is the last element in the sequence, then m_{2n+1} is the greatest sequence element ($\geq m$ for each $m \in M$)

Compute σ-states from $\textsc{MutexSkeleton}$ move sequence

- The initial state $\sigma(\emptyset)$ is defined by $owner = none$.
- Let $I \neq \emptyset$ be any finite initial segment of M. Then by the downward closure of I the least move m_0 in the given sequence is element of I so that the following function $last$ is well defined:

  $last(I) = max\{i \in NAT \mid m_i \in I\}$ -- index of last move in I
- Define the value of $owner$ in $\sigma(I)$ (read: the global state) by:

  $$owner = \begin{cases} ag(m_{last(I)}) & \textbf{if } last(I) \text{ is even} \\ none & \textbf{else} \end{cases}$$
- To show: For every maximal element $m \in I$ and $J = I \setminus \{m\}$:

  applying move m to $\sigma(J)$ yields $\sigma(I)$.

For the proof we distinguish two cases.

Proving coherence $\sigma(J) \Rightarrow_m \sigma(I)$: Case $m = m_{last(I)}$

If m is the greatest sequence element $m_{last(I)}$ in I, then $last(J) = last(I) - 1$ holds by monotonicity for the preceding $m_{last(J)}$.

- If $last(J)$ is even, $owner = ag(m_{last(J)})$ holds in $\sigma(J)$ (by definition).
  - Therefore $m_{last(J)}$ was a $\textsc{Grab}$ move, so that the next move $m_{last(I)} = m$ in the sequence is a $\textsc{Release}$ move of the same agent $ag(m_{last(J)}) = ag(m_{last(I)}) = ag(m)$.
  - Thus m applied to $\sigma(J)$ yields $owner = none$, as defined for $\sigma(I)$ with odd index $last(I)$.
- If $last(J)$ is odd, $owner = none$ holds in $\sigma(J)$ (by definition).
  - Therefore $m_{last(J)}$ was a $\textsc{Release}$ move so that the next move $m_{last(I)} = m$ in the sequence is a $\textsc{Grab}$ move of the same agent.
  - Thus m applied to $\sigma(J)$ yields $owner = ag(m) = ag(m_{last(I)})$, as defined for $\sigma(I)$ with even index $last(I)$.

Proving coherence $\sigma(J) \Rightarrow_m \sigma(I)$: Case $m \neq m_{last(I)}$

The case assumption $m \neq m_{last(I)}$ implies $last(I) = last(J)$ (by the definition of the $last$ function).

- The total order condition $m \leq m_{last(I)}$ **or** $m_{last(I)} < m$ implies that m comes after $m_{last(I)}$ (i.e. $m_{last(I)} < m$)
  - because $m \neq m_{last(I)}$ and because m is maximal in I.
- Case 1. $last(I)$ is even. Let $a = ag(m_{last(I)})$.

  Then $owner = a$ holds in $\sigma(I)$ and in $\sigma(J)$ (by definition because $last(I) = last(J)$ are even) and $m_{last(I)}, m_{last(J)}$ are $\textsc{Grab}$ moves. Furthermore $ag(m) \neq a$.
  - If $ag(m) = a$, by the last move condition move $m > m_{last(I)}$ is made by a different $ag(m) \neq ag(m_{last(I)}) = a$, a contradiction.

  Therefore rule $\textsc{MutexSkeleton}$ applied by $ag(m)$ to $\sigma(J)$ yields an empty update set (read: does not change the state), resulting in $\sigma(I)$.

Proving coherence $\sigma(J) \Rightarrow_m \sigma(I)$: Case $m \neq m_{last(I)}$ (Cont'd)

- Case 2. $last(I)$ is odd.
  - Then $owner = none$ holds in $\sigma(I)$ so that $m_{last(I)}$ is a $\textsc{Release}$.
  - Due to $m_{last(I)} < m$, $m_{last(I)}$ is not the last move in the sequence so that the next move $m_{last(I)+1}$, a $\textsc{Grab}$ move, is in the sequence.
  - The total order condition implies

    $$m < m_{last(I)+1} \textbf{ or } m \leq m_{last(I)+1}$$

    - $m < m_{last(I)+1}$ implies $m_{last(I)} < m < m_{last(I)+1}$, contradicting the property that there is no move between a $\textsc{Release}$ and the next $\textsc{Grab}$ move.
    - $m_{last(I)+1} \leq m \in I$ implies (by downward closure) that $m_{last(I)+1} \in I$, contradicting the definition of the $last$ function.

Therefore Case 2 is not possible.

Constructing $\textsc{MutexSkeleton}$ move sequence by induction

Case: i=0.

- Let m_0 be a minimal move of M. To prove: $m_0 \leq m$ for each $m \in M$.
- Proof. Let m_1 be a minimal move of $I_m = \{m' \in M \mid m' \leq m\}$. Then $m_0 = m_1$ because otherwise:
  - $m_1 \not\leq m_0$ since $m_1 \leq m_0$ implies $m_1 = m_0$ by minimality of m_0.
  - $m_0 \not< m_1$ since $m_0 < m_1$ implies $m_0 \not\leq m$ (by minimality of m_1) so that either $m < m_0$ (contradicting the minimality of m_0) or m, m_0 are incomparable so that $ag(m) \neq ag(m_0)$ and firing them in an initial state in different order yields linearizations with different result $owner$ (contradicting the linearization property of the given po-run).
  - $ag(m_0) \neq ag(m_1)$ because othw m_0, m_1 are ordered ($m_1 \leq m_0$ or $m_0 \leq m_1$) by the sequentiality of agents.
  - Then $\{m_0, m_1\}$ is an initial segment and m_0 **seq** m_1, m_1 **seq** m_0 yield states with different $owner$ $ag(m_0)$ resp. $ag(m_1)$ (the first move must be a $\textsc{Grab}$), contradicting the linearization property.

Constructing $\textsc{MutexSkeleton}$ move sequence (2)

Case: i=0 (Cont'd).

It remains to show: $m_0 = m_1$ *implies* $m_0 \leq m$.

- Obvious if $m_0 = m$. Therefore assume $m_0 \neq m$.
  - By minimality of m_0, initial state condition $owner = none$ and the guards of $\textsc{MutexSkeleton}$, m_0 must be a $\textsc{Grab}$ move and it must be the first $\textsc{Grab}$ move in the given po-run.
  - Wlog we can consider the case that also m is a $\textsc{Grab}$ move (otherwise consider the corresponding preceding $\textsc{Grab}$ move).
  - Since $m \not< m_0$ (by minimality of m_0) and by the guards of $\textsc{MutexSkeleton}$, move m can only happen after $ag(m_0)$ did $\textsc{Release}$ the resource it did $\textsc{Grab}$ by its move m_0.
    - A concurrent execution (assuming $m_0 \not< m$) would lead to linearizations of m_0, m with different $owner$ result $ag(m_0)$ resp. $ag(m)$.

    This implies $m_0 < m$.

Constructing MutexSkeleton move sequence (3)

Induction step: definition of m_{2i+1}. Let $a = ag(m_{2i})$. By induction hypothesis m_{2i} is a Grab move so that $owner = a$ is the state result of every linearization of $\{m \in M \mid m \leq m_{2i}\}$.

Case 1: a makes no more move after (read: $>$) m_{2i}. Then every move $m > m_{2i}$ in M can only be a move of another agent.

Case 2: Else. Let m_{2i+1} be the next move of a in the po-run, formally: the least element of $\{m \in M \mid m_{2i} < m$ **and** $ag(m) = a\}$.

- Then a makes no move bw m_{2i} and m_{2i+1} (by minimality of m_{2i+1}).

To prove that m_{2i+1} is a Release move we must show that a makes no move not-before m_{2i} but strictly before m_{2i+1}. Let β be a linearization of $\{m \in M \mid m \not\leq m_{2i}$ **and** $m < m_{2i+1}\}$.

- By ind hypo $m \leq m_{2i}$ or $m_{2i} < m$, so that $m \in \beta$ implies $m_{2i} < m < m_{2i+1}$, which implies that $ag(m) \neq a$.

Thus β does not update $owner = a$ and m_{2i+1} sets it to $none$.

Constructing $\textsc{MutexSkeleton}$ move sequence (4)

To show: Every $m \in M$ is ordered wrt ($\leq$ or $\geq$) m_{2i+1}.

- By ind hypo $m \leq m_{2i}$ or $m_{2i} < m$.
- Case 1: $m \leq m_{2i}$. This implies $m \leq m_{2i+1}$ since $m_{2i} < m_{2i+1}$ (by definition of m_{2i+1}.
- Case 2: $m_{2i} < m$. We derive a contradiction from the assumption that m is not ordered wrt m_{2i+1}.

Assumption: the set U of wrt m_{2i+1} unordered moves bw m_{2i} and m

$$U = \{m' \in M \mid m_{2i} < m' \leq m \textbf{ and } m' \text{ is not ordered wrt } m_{2i+1}\}$$

is not empty. Then it has a minimal element $m_0 \in U$.

We show that m_{2i} and m_0 have different agents and that $I_{m_{2i+1}} \cup \{m_0\}$ is a finite initial segment of M, so that two linearizations ending by m_{2i+1} **seq** m_0 resp. m_0 **seq** m_{2i+1} can be defined with different result $ag(m_0)$ resp.$none$, contradicting the linearization property.

Constructing $\textsc{MutexSkeleton}$ move sequence (5)

- $ag(m_0) \neq ag(m_{2i})$. Proof: Assume $ag(m_0) = ag(m_{2i})$.
  - by ind hypo $m_0 \leq m_{2i}$ or $m_{2i} < m_0$.
    - Case $m_0 \leq m_{2i}$: this implies $m_0 \leq m_{2i} < m_0$ (since $m_0 \in U$), contradicting the order relation $<$.
    - Case $m_{2i} < m_0$: since $ag(m_0) = ag(m_{2i}) = ag(m_{2i+1}) = a$, by the sequentiality of agents and because a makes no move bw $m_{2i} < m_0$ and m_{2i+1} it follows that $m_0 \geq m_{2i+1}$, contradicting that as an element of U, m_0 is not ordered wrt $_{2i+1}$.
- $I_{m_{2i+1}} \cup \{m_0\}$ is a finite initial segment of M.
  - Proof. Let $m' < m_o$. By ind hypo $m' \leq m_{2i}$ or $m_{2i} < m'$.
  - Case 1. $m' \leq m_{2i}$. This implies $m' \leq m_{2i+1}$, i.e. $m' \in I_{m_{2i+1}}$.
  - Case 2. $m_{2i} < m'$. $m' < m_0$ implies $m' \notin U$ (by minimality of m_0), i.e. $m' \not\leq m$ or $m_{2i+1} < m'$ or $m' \leq m_{2i+1}$. $m' < m_0 \leq m$ implies $m' \leq m$, $m_{2i+1} < m' < m_0$ orders m_0 wrt m_{2i+1}. Thus $m' \in I_{m_{2i+1}}$.

Constructing $\textsc{MutexSkeleton}$ move sequence (6)

Induction step: definition of m_{2i+2} (with hidden choice).

Case 1: After m_{2i+1} there is no more move in M. Then $m \leq m_{2i+1}$ for every $m \in M$ (by ind hypo).

Case 2: Else. Choose a minimal later move $m_{2i+2} > m_{2i+1}$.

- Then there is no $m_{2i+1} < m' < m_{2i+2}$ (by minimality of m_{2i+2}) so that m_{2i+2} is a $\textsc{Grab}$ move (since m_{2i+1} is a $\textsc{Release}$ move).

To show: Every $m \in M$ is ordered wrt ($\leq$ or $\geq$) m_{2i+2}.

- By ind hypo $m \leq m_{2i+1}$ or $m_{2i+1} < m$.
- Case 1: $m \leq m_{2i+1}$. Then $m \leq m_{2i+2}$ (because $m_{2i+1} < m_{2i+2}$).
- Case 2: $m_{2i+1} < m$. We derive a contradiction from the assumption that m is not ordered wrt m_{2i+2}.

Assumption: the set U of wrt m_{2i+2} unordered moves bw m_{2i+1} and m

$$U = \{m' \in M \mid m_{2i+1} < m' \leq m \textbf{ and } m' \text{ is not ordered wrt } m_{2i+2}\}$$

is not empty. Then it has a minimal element $m_0 \in U$.

Constructing $\textsc{MutexSkeleton}$ move sequence (7)

We show that m_{2i+2} and m_0 have different agents and that $I_{m_{2i+1}}$ can be extended to a finite initial segment $I_{m_{2i+1}} \cup \{m_0, m_{2i+2}\}$ of M.

- Then two linearizations ending by m_{2i+2} **seq** m_0 resp. m_0 **seq** m_{2i+2} can be defined with different result $ag(m_{2i+2})$ resp. $ag(m_0)$, contradicting the linearization property.

Since there is no $m_{2i+1} < m' < m_{2i+2}$, for the *initial segment property* it suffices to show that $m' < m_0$ implies $m' \leq m_{2i+1}$.

- Let $m' < m_0$. By ind hypo $m' \leq m_{2i+1}$ or $m_{2i+1} < m'$. But $m_{2i+1} < m'$ implies a contradiction, so that $m' \leq m_{2i+1}$.
  - Assume $m_{2i+1} < m'$. It would imply (1) $m_{2i+2} \not\leq m'$
    - $m_{2i+2} \leq m' < m_0$ implies $m_0 \geq m_{2i+2}$, contradicting $m_0 \in U$

    and (2) $m' \not< m_{2i+2}$ (since there is no $m_{2i+1} < m' < m_{2i+2}$), so that $m' \in U$ (since $m' < m_0 \leq m$ by $m_0 \in U$), by $m' < m_0$ contradicting the minimality of m_0.

$ag(m_0) = ag(m_{2i+2})$ would imply that $m_0 \in U$ is ordered wrt m_{2i+2}.

Invitation to join future work

Test the Concurrent ASM Thesis

- trying to model, analyse and stepwise refine in a provably correct manner asynchronous distributed algorithms/systems in terms of concurrent ASMs

the way it has been achieved successfully for sequential and synchronous systems using single-agent ASMs.

References on ASMs

- E. Börger and K.-D. Schewe, *Concurrent Abstract State Machines*
  - Acta Informatica 53 (5), 469-492 (2016)
- E. Börger and K.-D. Schewe, *A Behavioural Theory of Recursive Algorithms*
  - 2019, submitted for publication

- E. Börger and A. Raschke: Modeling Companion for Software Practitioners.
  - Springer-Verlag 2018 `http://modelingbook.informatik.uni-ulm.de`
- E. Börger and R. Stärk, Abstract State Machines. A Method for High-Level System Design and Analysis.
  - Springer-Verlag 2003.

Further References

- Y.Gurevich: Evolving algebras 1993: Lipari Guide.
  - in: E.Börger (Ed.): Specification and Validation Methods. Oxford University Press 1995, 9–36
    - Here the notion of distributed partial order ASM run appears for the first time.
- L. Lamport: How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs.
  - IEEE Trans. Computers 28 (9) 1979,690-691
- L. Lamport: Time, Clocks, and the Ordering of Events in a Distributed System.
  - Communications of the ACM 21 (7) 1978, 558-565

Copyright Notice

It is permitted to (re-) use these slides under the CC-BY-NC-SA licence

https://creativecommons.org/licenses/by-nc-sa/4.0/

i.e. in particular under the condition that

- the original authors are mentioned
- modified slides are made available under the same licence
- the (re-) use is not commercial