Concurrent Abstract State Machines

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

- 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)?
Is there a notion of concurrent ASM runs s.t.

- it allows the practitioner to \textit{faithfully model} concurrent real-life systems
  - for a \textit{rigorous analysis} (‘ground model’ concern)
  - for a \textit{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 \textit{most general concept of state/change} of sequential ASMs
  - integrating shared data into the control-flow view

- it preserves the \textit{parallel computation model} for single-agent ASMs
  - relegating behaviourally irrelevant sequentialization to refinements
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 $a = \textit{self}$ 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 $\Sigma$ (global view) of the signatures $\Sigma_a$ (local view) of programs $pgm_a$ of $a \in Agent$. We also write $(a, pgm_a)$ for $pgm_a$ and $ag(P) = a$ if $P = pgm_a$. 
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:

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

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: IF, ID, EX, MEM, WB

---

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

Copyright Egon Börger, released under Creative Commons BY-NC-SA International Licence 4.0
A *SyncAsm* parallelizing DLX pipelining stages

A refinement relying upon compiler assumptions to avoid conflicts

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

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

Verified stepwise refinement see AsmBook Ch.3.3

Copyright Egon Börger, released under Creative Commons BY-NC-SA International Licence 4.0
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.

\[
\text{InterleavingAsm}(a, \text{pgm}_a)_{a \in \text{Agent}} = \text{choose } a \in \text{Agent} \text{ do } \text{pgm}_a
\]

Characteristics of \text{InterleavingAsm}-runs \( R \):

- \( R \) linearly orders the actions of the (‘concurrently’ running) agents
- \( R \) treats each step of \( \text{pgm}_a \) as an atomic action
- \( R \) appears to each \( a \) as single-agent ASM run \( \text{View}_a(R) \)
  – 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 \( \text{env}(a) \) can make a move to update monitored or shared locations.

---

\(^2\) in the AsmBook and the ModelingCompanion
Single-agent $\text{View}_a(R)$ of $\text{INTERLEAVING ASM}$ runs $R$

- let $R$ be a run of $\text{INTERLEAVING ASM}((a, pgm_a)_{a \in \text{Agent}})$,
- 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 $\text{env}(a)$ be the set $\text{Agent} \setminus \{a\}$ of other agents.

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

Therefore in every $\text{View}_a(R)$-step

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

the effect of the $n$-th move of the $\text{env}(a)$ is defined by

$$S'_{n,a} \rightarrow e_1 \ldots \rightarrow e_k \quad S_{n+1,a}$$
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.
Let \textbf{RacyWrite} consist of agents $a_i \ (i = 1, 2)$ with rule
\[
\text{if } \text{mode}_{a_i} = \text{start} \text{ then }
\text{if } f \neq i \text{ then } f := i \\
\text{mode}_{a_i} := \text{stop}
\]

Consider the 1-step runs of $a_i$, started in an initial state of $a_i$ where $\text{mode}_{a_i} = \text{start}$ 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 \textbf{RacyWrite}-components with two witnesses (runs with sequential move executions) yielding different results:

$m_1 \text{ seq } m_2$ yielding $f = 2$ and $m_2 \text{ seq } m_1$ yielding $f = 1$

\textbf{NB.} \textit{There is no partial order \textbf{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

\[
x := 1 \mid y := 1 \mid \text{Read}(x) \text{ seq } \text{Read}(y) \mid \text{Read}(y) \text{ seq } \text{Read}(x)
\]

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 $<$:

\[
x := 1 < (a_3, \text{Read}(x)) \quad -- a_3 \text{ should read } x = 1, \text{ initially } x = 0
\]

\[
< (a_3, \text{Read}(y)) \quad -- \text{by sequential order of } a_3\text{-moves}
\]

\[
< y := 1 \quad -- \text{since } a_3 \text{ should read } y = 0
\]

\[
< (a_4, \text{Read}(y)) \quad -- \text{since } a_4 \text{ should read } y = 1 \text{ and initially } y = 0
\]

\[
< (a_4, \text{Read}(x)) \quad -- \text{by sequential order of } a_4\text{-moves}
\]

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

Extending sequential consistency to ASM runs

- read/write operations $<\text{op},\text{val}>$ 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 $S_{n,a}$

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

implies $S'_{n,a} \rightarrow e_1 \ldots \rightarrow e_k \quad S_{n+1,a}$
Gurevich’s partial order ASM runs: the definition

Let $M$ be a multi-agent ASM. A partial order $M$-run (called also distributed ASM run or po-run of $M$) 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 $\sigma$ 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.
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.
Consider **INDEPENDENT WRITE** with two agents $a, b$, rule $f(\text{self}) := 1$ 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 $\sigma$ illustrated by:

\[
\begin{align*}
  f(a) &= f(b) = 0 \\
  \emptyset \\
  \leftarrow & \quad \rightarrow \\
  f(a) &= 1 \quad f(b) = 0 \quad \{m_a\} \quad \{m_b\} \quad f(a) = 0 \quad f(b) = 1 \\
  \downarrow & \quad \downarrow \\
  \{m_a, m_b\} \\
  f(a) &= f(b) = 1
\end{align*}
\]
NB. In **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 $S_{k+1} = \text{the result of performing move } m_k \text{ in } S_k$.

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 $\sigma(I_{k+1})$.  

Copyright Egon Börger, released under Creative Commons BY-NC-SA International Licence 4.0
Proof of the unique result property

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

- **Base step** \( \text{size}(I) = 0 \): \( \sigma(I_0) = S_0 = \sigma'(I_0) \) by assumption.
- **Induction step** \( \text{size}(I) = k + 1 \): 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
    
    \[
    \sigma(I) = \text{result}(m, \sigma(I \setminus \{m\})) = \text{result}(m, \sigma(I^*)) \\
    = \text{result}(m, \sigma'(I^*)) = \sigma'(I \setminus \{m\}) = \sigma'(I)
    \]
**RacyWrite** has no not-single-agent partial order ASM runs

\[ \text{RacyWrite} = \]

\[
\text{if mode}_{a_i} = \text{start} \text{ then}
\]

\[
\text{if } f \neq i \text{ then } f := i
\]

\[
\text{mode}_{a_i} := \text{stop}
\]

-- two agents \( a_i \) with \( i = 1, 2 \)

-- NB. \( f \) is a *shared* fct

There is no partial order ASM run of **RacyWrite**, started in \( \text{mode}_{a_i} = \text{start} \), 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 \( \sigma \) satisfies \( f = i \) in \( \sigma(\{m_i\}) \).

– \( m_1 \text{ seq } m_2 \) and \( m_2 \text{ seq } m_1 \) 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.

Copyright Egon Börger, released under Creative Commons BY-NC-SA International Licence 4.0
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

\[ C = (asm_a)_{a \in Agent} \]

such that

- each component \( asm_a \) is an instance \( \text{amb } a \text{ in } P \) of an ASM rule \( P \) from a **finite program base** \( B \) of

  - (possibly non-deterministic) **sequential** ASMs
  
  - **programs which may create new agents** (potentially infinitely many)

    - via rules \( \text{let } a = \text{new } (Agent) \text{ in } r \)

- 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 $G$ with $Transition$ rules)
- global state:
  - in classical Petri nets a marking of Places of $G$ by (possibly coloured) tokens, i.e. a dynamic fct marking : Place $\rightarrow$ Token
  - in predicate/transition nets a 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:

  $\text{PetriBehavior}(G, Transition) =$

  \[ \text{choose } T \subseteq Transition \text{ if } \text{Enabled}(T) \text{ then } \text{Fire}(T) \]

  - 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 $Transition$ has only finitely many subsets so that $\text{PetriBehavior}$ is in fact a non-deterministic sequential ASM.
What it means to FIRE an Enabled single Petri net transition

The (atomic!) actions to FIRE(t) a single Enabled t ∈ 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 FIRE(t) yields a set of updates:
    - ((marking, p), marking(p) − m_{t,p}) for each p ∈ PrePlace(t)
    - ((marking, q), marking(q) + n_{t,q}) for each q ∈ PostPlace(t)
- if Enabled(t), a conjunction of conditions m_{t,p} ≤ marking(p)

- in predicate/transition nets a generalization
  - of +/− on tokens to union/difference operations on sets
  - of ≤ in 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 \textbf{Fire} an \textit{Enabled} set of Petri net transition

\textit{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 \( t \)

\[ \text{forall } p \in \text{PrePlace}(t) \; m_{t,p} \leq \text{marking}(p) \]

is sharpened to a \textit{cumulative enabling condition} which provides simultaneously all resources required by any transition in \( T \).

- Let \( T_p = \{ t \in T \mid p \in \text{PrePlace}(t) \} \) (transitions with pre-place \( p \)).

Then the cumulative enabling condition becomes

- in traditional Petri nets \( \sum_{t \in T_p} m_{t,p} \leq \text{marking}(p) \)
- in predicate/transition nets generalized in terms of \( \bigcup \) and \( \subseteq \)

\textbf{Fire}(T) cumulatively applies the partial updates computed by \textbf{Fire}(t) for each \( t \in T \).
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 *Enabledness* reads every involved location \((\text{marking}, p)\)
  - \(\text{Enabled}(t)\) reads \((\text{marking}, p)\) for each pre-place \(p\) of \(t\)
  - \(\text{Enabled}(T)\) reads for each \(t \in T\) each \((\text{marking}, p)\) 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)
Consider Petri nets as multi-agent systems \((t_a)_{a \in \text{Agent}}\).

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 \(\text{Enabled}(T)\) resp. \(\text{Fire}(T)\), defined in terms of ‘concurrent access to variables’—i.e. simultaneous atomic reads resp. atomic read & write actions for shared *Places*
- 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

\[
\text{choose } T \subseteq \text{Transition} \quad \text{if } \text{Enabled}(T) \quad \text{then } \text{FIRE}(T)
\]

- 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 ($|T| = 1$).
  - 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 $\text{Enabled}(T)$

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 $C = (a_i, asm_i)_{i \in I}$ of nd-seq components whose concurrent runs are definable by partial-order runs one can construct an nd-seq ASM $S_C$ such that the concurrent runs of $C$ and the runs of $S_C$ 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
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 $\mathcal{A}_n$ 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 interfere 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 \( A \) be a multi-agent ASM \((a, \text{pgm}(a))_{a \in A}\). A concurrent ASM run of \( A \) 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 \( \text{loc} \) between
    - the state \( S_{ja} \) where agent \( a \) interacts with other agents to read this \( \text{loc} \) 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 \( S_{ja} \)
  - 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 $\text{ConcurStep}(\text{pgm}(a))$ 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 $\text{pgm}(a)$ 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&SaveGlobalData, LocalWriteStep, WriteBack

Asynchronously, in different states (read: at clock ticks of $a$)
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 $\text{ConcurStep}(\text{pgm}(a))$.

Expressed equationally:

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

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

- computing the updates in $S_n$ either by $\text{pgm}(a)$ or by one of the simulation components
  - asynchronously at the agent’s clock tick
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 ConcurSteps of $pgm(a)$, it is in one of three (resp. two) modes:

- in $mode(a) = \text{interactive}$, $a$ 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 $\text{SaveGlobalData}$ locally, switching to local emulation

- in $mode(a) = \text{localEmulation}$, $a$ computes a local copy of $U_a$, using the saved global data, & switches to $\text{WriteBack}$ to interaction locs

- in $mode(a) = \text{writeBack}$, $a$ will $\text{WriteBack}$ to those (globally visible) interaction locations whose values it has updated locally (by executing an assignment $f_a(s) := t$ in $mode = \text{localEmulation}$).

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

- other formalizations are possible which hide local updates
Local simulation components of ConcurStep$(a)$

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

LocalWriteStep$(pgm) =$

\[ pgm[f/f_a] \quad \text{-- replace each interaction fct } f \text{ by a local copy } f_a \]

add in the modified program in parallel to each $f_a(s) := t$

\[ \text{Record}(\text{updData}(f_a, s, t), \text{GlobalUpd}) \]

where $a = ag(pgm)$

SaveGlobalData$(pgm) =$ forall $f \in Monitored \cup Shared \quad f_a := f$

WriteBack$(pgm) =$

forall updData$(f_a, s, t) = ((f_a, args), val) \in \text{GlobalUpd}$

\[ f(args) := val \]

\[ \text{GlobalUpd} := \emptyset \quad \text{-- GlobalUpd assumed to be initially empty} \]
Two-step version of \texttt{ConcurStep}(pgm)

\begin{verbatim}
choose \ P \in \{\texttt{Read\&WriteStep}(pgm), \texttt{ReadStep}(pgm)\} \ do \ P

if \ mode = \texttt{localEmulation} \ then
\texttt{LocalEmulation}(pgm)

\texttt{LocalEmulation}(pgm) =

\texttt{Read\&WriteStep}(pgm) =

\texttt{ReadStep}(pgm) =

\texttt{SaveGlobalData}(pgm)

\texttt{LocalEmulation}(pgm) =

\texttt{LocalWriteStep}(pgm) \ seq \ \texttt{WriteBack}(pgm)

\end{verbatim}

NB. Turbo ASM operator \texttt{seq} guarantees atomic 1-step execution.
Multi-step version of \texttt{ConcurStep}(\textit{pgm})

NB. One may refine \texttt{Record}(\textit{updData}(f_a, s, t), \textit{GlobalUpd}) (or \texttt{WriteBack}) to handle also local update overwritings
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 altogether they yield an inconsistent update set, another subset of agents is tried out, until all possible combinations have 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 \((a, \text{alg}(a))_{a \in A}\) 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 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

\[ \mathcal{P} = (P_i)_{i \in I} \]

- for example in a distributed mutual exclusion protocol that a
  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 **MutexLamport** algorithm with a corresponding **MutexSkeleton** ASM.
Idea: *parameterize component actions by* a **logical time parameter**—a local $\text{stepCount}_{P_i}$ 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 $\text{Send}/\text{Receive}(msg)$ are linked to **locally reflect** the following stipulation:
  - $\text{Send}(msg)$ by $P_i$ to $P_j$ HappensBefore $\text{Receive}(msg)$ by $P_j$

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

- $m$ HappensBefore $m'$ implies

$$\text{stepCount}_{\text{ag}(m)}(m) < \text{stepCount}_{\text{ag}(m')}(m')$$

The ClockCondition can then be used to make HappensBefore locally observable.
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 $\text{MutexLamport}$ with 5 actions:
    - $\text{RequestResource}$ and $\text{ReleaseResource}$
    - $\text{FetchMsgs}$ (transfer from *mailbox* to local memory)
    - $\text{ReceiveRequestMsg}$ and $\text{ReceiveReleaseMsg}$.

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

- every $\text{RequestResource}$ 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 $\text{ReleaseResource}$ 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 request 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

\( m \text{ HappensBefore } m' \Rightarrow \text{stepCount}_{ag}(m)(m) < \text{stepCount}_{ag}(m')(m') \)

- for \( P_i \)-moves \( m, m' \): use \textsc{Increment}(\text{stepCount}_{P_i}) at every \( P_i \)-step
- for corresponding \text{Send/Receive}(msg) moves \( m, m' \):
  - associate every \( msg \in \text{Msg} \) with a ‘logical’ \( \text{sendTime}(msg) \)
    - namely \( \text{stepCount}_{ag}(m)(m) \) when move \( m = \text{Send}(msg) \) is made
  - for each \( msg \) found in \( \text{mailbox}_{receiver} \)
    - namely by a \( receiver \)'s \text{FETCHMSGs} move \( m \)
      \[ \text{Adjust} \: \text{stepCount}_{receiver}(m) \text{ to a value } > \text{sendTime}(msg) \]
    - define the proper \text{Receive}(msg) moves as \text{REQUESTMsg} or \text{RELEASEMsg} step for a \( msg \) \text{FETCHMSGs} has transferred to local memory, \text{Adjusting} \text{stepCount}_{receiver} \]

This makes \( \text{Send}(msg) \) \text{HappensBefore} \text{Receive}(msg) \) locally observable in \text{Receive[Request/Release]Msg} steps by
\[ \text{stepCount}_{sender}(\text{Send}(msg)) < \text{stepCount}_{receiver}(\text{Receive}(msg)) \]
How to locally compute $SatisfiedAllReqsCompetingWith(req)$

- 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 $sendTime(req)$ is known locally
  – to order also concurrent req moves (not ordered by $HappensBefore$) use any static linear order $<_P$ every process knows
- NB. Similarly one can linearize the entire $HappensBefore$.

$req$ GoesBefore $req'$ iff

$$sendTime(req) < sendTime(req') \text{ or }$$

$$(sendTime(req) = sendTime(req') \text{ and } sender(req) <_P sender(req'))$$

$SatisfiedAllReqsCompetingWith(req)$ iff

forall $req' \in ReqMsg \setminus \{req\}$ $req$ GoesBefore $req'$
Assumptions on communicating agents

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

For defining $\text{HasSeenAllReqsCompetingWith}(\text{req})$ we exploit that msgs are not lost and furthermore received in sending order.
How to locally compute $\text{HasSeenAllReqsCompetingWith}(\text{req})$

- We say that a request of $P_i$ competes with a req' of another $P_j$ if $\text{req'}$ GoesBefore $\text{req}$, in which case $\text{req'}$ has been made before or at the same time as $\text{req}$ (i.e. $\text{sendTime}(\text{req'}) \leq \text{sendTime}(\text{req})$).
- Therefore, when $P_i$ has a pending $\text{req} \in \text{ReqMsg}$, this req does not compete with any laterReq sent by $P_j$, i.e. timestamped with $\text{sendTime}(\text{laterReq}) > \text{sendTime}(\text{req})$.
- 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 $\text{req} \in \text{ReqMsg}$ and has Received from another $P_j$ any laterMsg
- i.e. with $\text{sendTime}(\text{laterMsg}) > \text{sendTime}(\text{req})$

then in some preceding states $P_i$ must have seen—Received and Inserted into $\text{ReqMsg}$—any req' from $P_j$ its own req competes with.
Defining \textit{ResourceGrantedTo}

The Competitor Knowledge Lemma can be applied if

- after sending a request, \( P_i \) receives from each other \( P_j \) a laterMsg.

To guarantee this, we let each \texttt{ReceiveRequestMsg} move send a timestamped \texttt{acknowledgment} to the sender of the request (see below).

\texttt{HasSeenAllReqsCompetingWith}(\texttt{req}) \texttt{iff}

\[
\text{forall } P \in \mathcal{P} \setminus \{\text{self}\} \text{ there is some } \texttt{msg} \text{ with } \texttt{sender}(\texttt{msg}) = P \text{ and } \texttt{Received}(\texttt{msg}) \text{ and } \texttt{sendTime}(\texttt{msg}) > \texttt{sendTime}(\texttt{req})
\]

\texttt{ResourceGrantedTo}(P) \texttt{iff}

\[
\text{forsome } \texttt{req} \in \texttt{ReqMsg} \text{ with } \texttt{sender}(\texttt{req}) = P \text{ -- an own req}
\]

\texttt{HasSeenAllReqsCompetingWith}(\texttt{req}) \texttt{and}

\texttt{SatisfiedAllReqsCompetingWith}(\texttt{req})
Recap **MutexLamport** signature

- \( \text{Msg} = \) initially empty set of messages created by protocol actions
  - msg content comprehends sender, sendTime and type, formally:
    - \( \text{sendTime} : \text{Msg} \to \mathbb{NAT} \)
    - \( \text{type} : \text{Msg} \to \{ \text{request}, \text{release}, \text{ack} \} \)
    - \( \text{sender} : \text{Msg} \to \{ P_i \mid i \in I \} \)

- Controlled functions (instantiated for each \( P_i \) by \( \text{self} = P_i \))
  - \( \text{stepCount} : \mathbb{NAT} \) -- step counter, assumed to be initially \( \geq 0 \)
  - \( \text{mailbox} \subseteq \text{Msg} \) -- assumed to be initially empty
  - \( \text{ReqMsg} \subseteq \{ m \in \text{Msg} \mid \text{type}(m) = \text{request} \} \) -- initially empty

- Auxiliary sets used by **FETCHMSG** to internally store mailbox content
  - \( \text{Received} \subseteq \text{Msg} \) -- assumed to be initially empty
  - \( \text{MsgForProperReceive} \subseteq \text{Msg} \) -- assumed to be initially empty

- static orders \( \prec_P \) of components \( P_i \) and \( \prec \) of \( \mathbb{NAT} \)ural numbers and static set \( \mathcal{P} \).
Definition of \textsc{FetchMsgs}

\begin{align*}
\textbf{forall } m \in \text{mailbox} \\
\text{\textsc{Insert}}(m, \text{Received}) & \quad \text{-- transfer to local memory} \\
\text{\textsc{Adjust}}(\text{stepCount}, \text{mailbox}) & \quad \text{-- to become } > \text{sendTime}(m) \text{ for every } m \in \text{mailbox} \\
\text{if type}(m) \neq \text{ack} \text{ then } \text{\textsc{Insert}}(m, \text{MsgForProperReceive}) \\
\text{\textsc{Delete}}(m, \text{mailbox}) & \quad \text{-- usual meaning of \textsc{Insert}/\textsc{Delete}}
\end{align*}

\textbf{where}

\begin{align*}
\text{\textsc{Adjust}}(\text{stepCount}, \text{mailbox}) &= \\
\text{stepCount} &:= 1 + \max(\text{stepCount}, \max\text{SendTime}(\text{mailbox})) \\
\max\text{SendTime}(\text{mailbox}) &= \\
\max(\{\text{sendTime}(m') + 1 \mid m' \in \text{mailbox}\})
\end{align*}

This rule guarantees the ClockCondition for \textit{Send}/\textit{Receive}(msg) moves.
Defining `RequestResource`

\[\text{RequestResource} = \]

\[
\quad \text{let } req = \text{new } (Msg) \\
\quad \text{DecorateAsReq}(req) \\
\quad \forall P \in \mathcal{P} \setminus \{\text{self}\} \quad \text{SEND}(req, \text{to } P) \\
\quad \text{INSERT}(req, \text{ReqMsg}) \\
\quad \text{stepCount} := \text{stepCount} + 1
\]

\[\text{where}
\]

\[
\quad \text{DecorateAsReq}(m) = \\
\quad \quad \text{type}(m) := \text{request} \\
\quad \quad \text{sender}(m) := \text{self} \\
\quad \quad \text{sendTime}(m) := \text{stepCount}
\]

\[
\quad \text{SEND}(m, \text{to } P) \text{ triggers } \text{INSERT}(m, \text{mailbox}_P) \text{ to eventually happen.}
\]
let \( \text{msg} = \text{new} \ (\text{Msg}) \)

\text{DecorateAsRelease}(\text{msg})

\text{forall} \ P \in \mathcal{P} \setminus \{\text{self}\} \ \text{SEND}(\text{msg}, \text{to} \ P)

\text{DELETEOWNREQS}(\text{ReqMsg})

\text{stepCount} := \text{stepCount} + 1

where

\text{DecorateAsRelease}(m) =

\text{type}(m) := \text{release}

\text{sender}(m) := \text{self}

\text{sendTime}(m) := \text{stepCount}

\text{DELETEOWNREQS}(\text{ReqMsg}) =

\text{forall} \ \text{req} \in \text{ReqMsg} \ \text{with} \ \text{sender}(\text{req}) = \text{self}

\text{DELETE}(m, \text{ReqMsg})
\textbf{Defining \texttt{ReceiveRequestMsg}}

\texttt{ReceiveRequestMsg} = 
\begin{align*}
\text{forall } \, &msg \in \text{MsgForProperReceive with type}(msg) = \text{request} \\
\text{INSERT}(msg, \text{ReqMsg}) \\
\text{ACKNOWLEDGE}(msg) \\
\text{stepCount} &:= \text{stepCount} + 1 \\
\text{DELETE}(msg, \text{MsgForProperReceive})
\end{align*}

where
\begin{align*}
\text{ACKNOWLEDGE}(msg) &= \\
\text{let } m &= \text{new} \, (\text{Msg}) \\
\text{DECORATEASACK}(m) \\
\text{SEND}(m, \text{to sender}(msg))
\end{align*}
Defining **ReceiveReleaseMsg**

\[
\text{ReceiveReleaseMsg} = \\
\text{forall } \text{msg} \in \text{MsgForProperReceive with type(msg) = release} \\
\text{DeleteSenderReqs}(\text{ReqMsg}) \\
\text{stepCount} \:= \text{stepCount} + 1 \\
\text{Delete}(\text{msg}, \text{MsgForProperReceive})
\]

**where**

\[
\text{DeleteSenderReqs}(\text{ReqMsg}) = \\
\text{forall } \text{req} \in \text{ReqMsg with sender(req) = sender(msg)} \\
\text{Delete}(\text{req}, \text{ReqMsg})
\]
We call a process $P$ equipped with \texttt{MutexLamport} if $P$ executes a sequential $asm(P)$ which

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

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

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

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

\textbf{NB.} Different linearizations $GoesBefore$, applying different $\prec_{\mathcal{P}}$ 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 $\text{sendTime}(\text{ackMsg}) > \text{sendTime}(\text{req})$, so that $P_i$ in state $S$ HasSeenAllReqsCompetingWith(req).
- By Competitor Knowledge Lemma in some preceding states, $P_i$ has Inserted into ReqMsg every $\text{req}'$ its own req competes with.
- If in $S$ there is no such $\text{req}' \in \text{ReqMsg}$, then by its definition $\text{ResourceGrantedTo}(P_i)$ is true in $S$.
- By induction, when $\text{ResourceGrantedTo}(\text{sender(}\text{req}'\text{)})$, by assumption $\text{sender(}\text{req}'\text{)}$ will eventually RELEASERESOURCE so that eventually $\text{ResourceGrantedTo}(P_i)$ becomes true (via the rule RECEIVERELEASEMsg).
To show: *Whenever the resource has been granted to* $P_i$, $P_i$ *must* \texttt{ReleaseResource} *before it can be granted to another* $P_j$.

- Assume $\text{ResourceGrantedTo}(P_i)$ and $\text{ResourceGrantedTo}(P_j)$ in $S$.
- Then by definition both processes have an own $req_i \in ReqMsg_{P_i}$ resp. $req_j \in ReqMsg_{P_j}$ that $\text{GoesBefore}$ any other $req'$ in their $ReqMsg$.
- Since $\text{GoesBefore}$ is a total order of request messages, either $req_i \text{ GoesBefore req}_j$ or $req_j \text{ GoesBefore req}_i$.
- Case 1. $req_i \text{ GoesBefore req}_j$. 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 $\text{ReleaseResource}$, $req_i \in ReqMsg_{P_j}$ still holds in $S$. Therefore $req_j \text{ GoesBefore req}_i$ (because $\text{SatisfiedAllReqsCompetingWith}(req_j)$), a contradiction.
- Case 2. Symmetric in $i, j$. 
To show: \textit{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 \( req_i \) \textit{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 \texttt{ReceiveReleaseMsg} step of \( P_j \), which must have been triggered by a \texttt{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
if owner = self then owner := none
```

-- Grab move
-- 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.
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 MutexSkeleton ASM.  

- Observation: independent (not-ordered) $\text{Grab}$ 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 $\text{Grab}$ move $m$ must come in pair with a $\text{Release}$ move $m'$ of the same agent
  - as guaranteed already by the rule guards
  – but furthermore each other such $\text{Grab}/\text{Release}$ 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 MutexSkeleton moves.

---

3 We follow the analysis Robert St¨ark presented in his lectures at ETH Z¨urich in 2004.
In the following

- let $\mathcal{M}$ be any multi-agent ASM where each agent has the rule $\text{MutexSkeleton}$
  – possibly also other rules which however do not affect the values of the $\textit{owner}$ location, though they may depend on them
- let $(\mathcal{M}, \leq)$ be a partial order of $\text{MutexSkeleton}$-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 $\textit{owner}$—the global behavior requirement is about
- consider runs where initially $\textit{owner} = \textit{none}$
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 \textsc{Grab}/\textsc{Release} moves of a same agent \(ag(m_{2i})\)
- between moves of a \textsc{Grab}/\textsc{Release} pair and between a \textsc{Release} and the next \textsc{Grab} there is no other move in \(M\)
- all moves \(m \in M\) are ordered wrt \(m_i\), i.e. \(m \leq m_i \lor m_i < m\)
- **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 $\sigma$-states from 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:
  $$owner = \begin{cases} 
  ag(m_{last(I)}) & \text{if } last(I) \text{ is even} \\
  none & \text{else}
  \end{cases}$$
- 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 $m = m_{\text{last}}(I)$

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

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

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

- The total order condition \( m \leq m_{\text{last}(I)} \) or \( m_{\text{last}(I)} < m \) implies that \( m \) comes after \( m_{\text{last}(I)} \) (i.e. \( m_{\text{last}(I)} < m \))
  - because \( m \neq m_{\text{last}(I)} \) and because \( m \) is maximal in \( I \).

- Case 1. \( \text{last}(I) \) is even. Let \( a = ag(m_{\text{last}(I)}) \).

Then owner = \( a \) holds in \( \sigma(I) \) and in \( \sigma(J) \) (by definition because \( \text{last}(I) = \text{last}(J) \) are even) and \( m_{\text{last}(I)}, m_{\text{last}(J)} \) are \text{Grab} moves.

Furthermore \( ag(m) \neq a \).

- If \( ag(m) = a \), by the last move condition move \( m > m_{\text{last}(I)} \) is made by a different \( ag(m) \neq ag(m_{\text{last}(I)}) = a \), a contradiction.

Therefore rule \text{MutexSkeleton} applied by \( ag(m) \) to \( \sigma(J) \) yields an empty update set (read: does not change the state), resulting in \( \sigma(I) \).
Case 2. \( \text{last}(I) \) is odd.

– Then \( \text{owner} = \text{none} \) holds in \( \sigma(I) \) so that \( m_{\text{last}}(I) \) is a \texttt{RELEASE}.
– Due to \( m_{\text{last}}(I) < m \), \( m_{\text{last}}(I) \) is not the last move in the sequence so that the next move \( m_{\text{last}}(I)+1 \), a \texttt{GRAB} move, is in the sequence.
– The total order condition implies
  \[
  m < m_{\text{last}}(I)+1 \quad \text{or} \quad m \leq m_{\text{last}}(I)+1
  \]

• \( m < m_{\text{last}}(I)+1 \) implies \( m_{\text{last}}(I) < m < m_{\text{last}}(I)+1 \), contradicting the property that there is no move between a \texttt{RELEASE} and the next \texttt{GRAB} move.

• \( m_{\text{last}}(I)+1 \leq m \in I \) implies (by downward closure) that \( m_{\text{last}}(I)+1 \in I \), contradicting the definition of the \texttt{last} function.

Therefore Case 2 is not possible.
**Constructing 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:
  - \( m_1 \nleq m_0 \) since \( m_1 \leq m_0 \) implies \( m_1 = m_0 \) by minimality of \( m_0 \).
  - \( m_0 \nleq m_1 \) since \( m_0 < m_1 \) implies \( m_0 \nleq m \) (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 \( m_0 \text{ seq } m_1, m_1 \text{ seq } m_0 \) 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 \texttt{MutexSkeleton} move sequence (2)

Case: $i=0$ (Cont’d).

\textit{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 \texttt{MutexSkeleton}, $m_0$ must be a \texttt{GRAB} move and it must be the first \texttt{GRAB} move in the given po-run.
  - Wlog we can consider the case that also $m$ is a \texttt{GRAB} move (otherwise consider the corresponding preceding \texttt{GRAB} move).
  - Since $m \not< m_0$ (by minimality of $m_0$) and by the guards of \texttt{MutexSkeleton}, move $m$ can only happen after $ag(m_0)$ did \texttt{RELEASE} the resource it did \texttt{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 $\{m \in M \mid m_{2i} < m \text{ and } ag(m) = a\}$.

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 $\beta$ be a linearization of $\{m \in M \mid m \not\leq m_{2i} \text{ and } m < m_{2i+1}\}$.

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 $\beta$ does not update owner = $a$ and $m_{2i+1}$ sets it to none.
To show: Every \( m \in M \) is ordered wrt \((\leq \text{ or } \geq)\) \( m_{2i+1} \).

- 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 \( m \)

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

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 \( m_{2i+1} \text{ seq } m_0 \) resp. \( m_0 \text{ seq } m_{2i+1} \) can be defined with different result \( ag(m_0) \) resp. none, contradicting the linearization property.
Constructing **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 $m' < m_0$. 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. $m' \notin m$ 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 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 GRAB move (since $m_{2i+1}$ is a 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 $m$

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

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

Copyright Egon Börger, released under Creative Commons BY-NC-SA International Licence 4.0
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 $m_{2i+2} \text{ seq } m_0$ resp. $m_0 \text{ seq } m_{2i+2}$ 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


  – Springer-Verlag 2018
  http://modelingbook.informatik.uni-ulm.de

Further References

  - 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
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