Egon Börger (Pisa)

Concurrency, Interleaving and Mutual Exclusion

A critical analysis of interleaving used for Mutex algorithms

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

The observation

- The *interleaving model of computation* is widely used for the design and analysis of distributed algorithms.
  - see for example Nancy A. Lynch: Distributed Algorithms (1996).
- This includes mutual exclusion (Mutex) algorithms, see op.cit. Ch. 10.2, 10.4.
- An alternative computation model to describe and analyse runs of distributed systems is that of *concurrent ASMs*.
  - see E. Börger and K.-D. Schewe: Concurrent Abstract State Machines (Acta Informatica, 2016).
- An analysis of Mutex algorithms in terms of concurrent ASM runs reveals that

  using interleaving for Mutex algorithms may be begging the question.

Mutual Exclusion problem (Lynch 10.2, 10.4)

Problem: *allocate one nonshareable resource for exclusive use among ≥ 2 distributed processes*

- avoiding any form of central control
- using shared variables for information exchange bw processes

Schematic idea: each process

- from its remainder region (R)
- moves into a trying region (T)
- to gain exclusive access to the critical region (C)
- when the resource is not needed any more, in its exit region (E) executes an exit protocol
- to return to its remainder region (R)

i.e. the desired sequence of protocol phases for each process is

$$R \longrightarrow T \longrightarrow C \longrightarrow E \longrightarrow R$$

Typical Mutual Exclusion requirements

Mutual Exclusion: It never happens that two processes are simultaneously in the critical section.

Lockout-Freedom:

- Assuming that each process always returns the resource, every process that reaches the *trying* region *eventually* will enter the *critical* region.
  - Every process that reaches its *exit* region *eventually* will reenter its *remainder* region.

Fairness: The first-come-first-served principle holds (for a reasonable notion of coming-first which has to be defined)

The following **Progress** property follows from Lockout-Freedom (but no vice versa):

- If some p is in T and nobody in C, then eventually some p will enter C.
  - If some p is in E then eventually some p will enter R.

Peterson’s Mutex algorithm (Peterson 1981, Lynch 10.5)

- There are 2 processes, say $Process = \{p_1, p_2\}$.
- We define a concurrent ASM

  $$2\textsc{MutexPetersonAsm} = (p, 2\textsc{MutexPeterson}_p)_{p \in Process}$$

  where each process p executes its instance $2\textsc{MutexPeterson}_p$ of the algorithm $2\textsc{MutexPeterson}$ defined below.

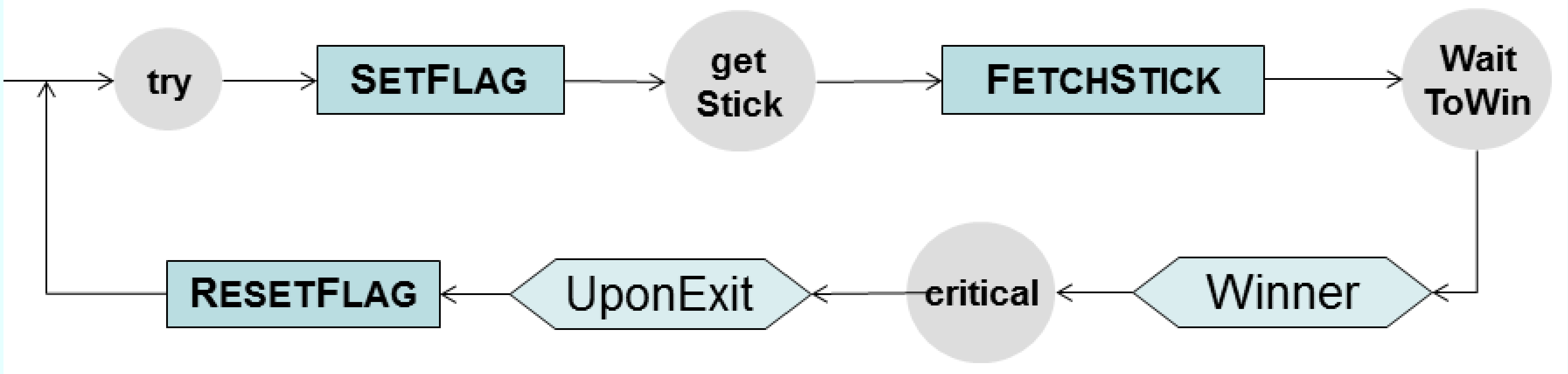
Algorithmic idea: each $p \in Process$, to compete for the resource:

- first indicates its interest by setting a $flag_p$, a variable (0-ary location) the other process can read
- then will FetchStick to become $stickHolder$, a variable (0-ary location) shared by both processes, initially $stickHolder \in Process$
- in possession of which it must $WaitToWin$ until, by *becoming a* $Winner$, it can enter the critical section
- whereafter $UponExit$ it resets its $flag_p$ to return to its remainder section

$\textsc{2MutexPeterson}$ control flow and data model

$\textsc{SetFlag} = (\mathit{flag}_{\textbf{self}} := 1) \qquad \textsc{ReSetFlag} = (\mathit{flag}_{\textbf{self}} := 0)$

$\mathit{flag}_p \in \{0, 1\}$ (initially $\mathit{flag}_p = 0$), writable by p (output location), readable (monitored) by $\mathit{theOtherProcess}$



$\textsc{FetchStick} =$ **if** $($**not** $\mathit{HasStick}($**self**$))$ **then** $\mathit{stickHolder} :=$ **self**

where $\mathit{HasStick}(p)$ **iff** $\mathit{stickHolder} = p$

NB. In each state only one process $\mathit{HasStick}$ and only one of them —$\mathit{theOtherProcess}$—can fetch it, by updating $\mathit{stickHolder}$ to it**self**.[1]

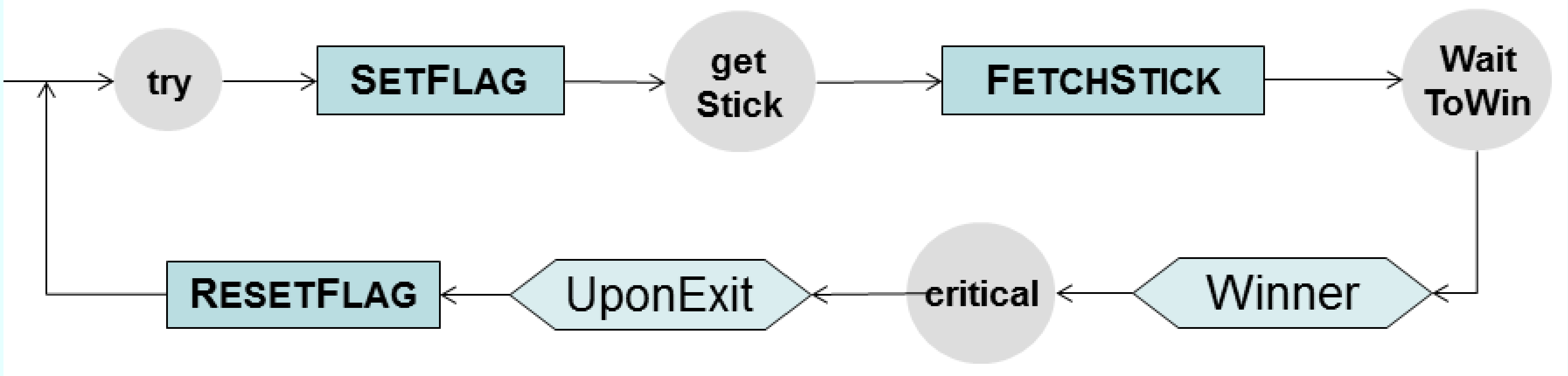
---

[1] Figure © 2016 Springer-Verlag Germany, reused with permission.

2MUTEXPETERSON *Winner* strategy

Winner **iff**

$NobodyElseInterested$ **or** $MeantimeSomebodyElseFetchedStick$



$NobodyElseInterested$ **iff** $flag_{theOtherProcess(\textbf{self})} = 0$

$MeantimeSomebodyElseFetchedStick$ **iff**

$stickHolder = theOtherProcess(\textbf{self})$

where $theOtherProcess(p_i) = p_j$ **with** $i \neq j \quad (i, j \in \{1, 2\})$

Proving 2MutexPetersonAsm properties

Assume that concurrent runs of 2MutexPetersonAsm start in an initial state and that each time a process can perform a step it eventually will execute a step. Then the following properties hold.

Proposition 1. 2MutexPetersonAsm satisfies the Mutual Exclusion requirement: never more than one process is in its $critical$ phase.

Proposition 2. 2MutexPetersonAsm satisfies the Lockout-Freedom requirement.

NB. Fairness holds only wrt which process is the first to enter mode $WaitToWin$.

Proof: induction on concurrent 2MutexPeterson runs.

A petitio principii concerning FETCHSTICK and $stickHolder$

- In Lynch 10.5 FETCHSTICK is defined (with different naming) simply as $stickHolder :=$ **self**, without guard '**if not** $HasStick$'.
  - It is the interleaving assumption for asynchronous runs of distributed algorithms which guarantees consistency, i.e. that at each moment at most one process makes a step, e.g. to update $stickHolder$.

This interleaving begs the question. It implies Mutex directly:

```
if $FreeCS$ then $csHolder :=$ self
  where $FreeCS$ iff $csHolder = \mathbf{undef}$
```

- In 2MUTEXPETERSON, the definition of FETCHSTICK is consistent for concurrent ASM runs, *without making the interleaving assumption*
  - because there are only two processes p_1, p_2 and because $stickHolder$ in each state has one of them as value. Therefore, in each step of a concurrent run of 2MUTEXPETERSONASM, at most one process can fetch the stick, namely by updating $stickHolder$ to it**self**.

Generalizing Peterson's Mutex algorithm for $n > 2$

NB. The generalization in Lynch op.cit. illustrates the petitio principii even more clearly. We show this by formulating in terms of ASM the generalization presented in Lynch op.cit.

- There are $n \geq 2$ processes, say $Process = \{p_1, \ldots, p_n\}$.
- We define a concurrent ASM
  $$\textsc{MutexPetersonAsm} = (p, \textsc{MutexPeterson}_p)_{p \in Process}$$
  where each process p executes its instance $\textsc{MutexPeterson}_p$ of $\textsc{MutexPeterson}$.

Algorithmic idea: every p must *compete* for the resource successively *at each level*, from 1 to $n-1$,

- in a competition arranged such that at each level, one process looses
  - so that at each level k, at most $n-k$ processes can win, therefore at most one process at level $n-1$

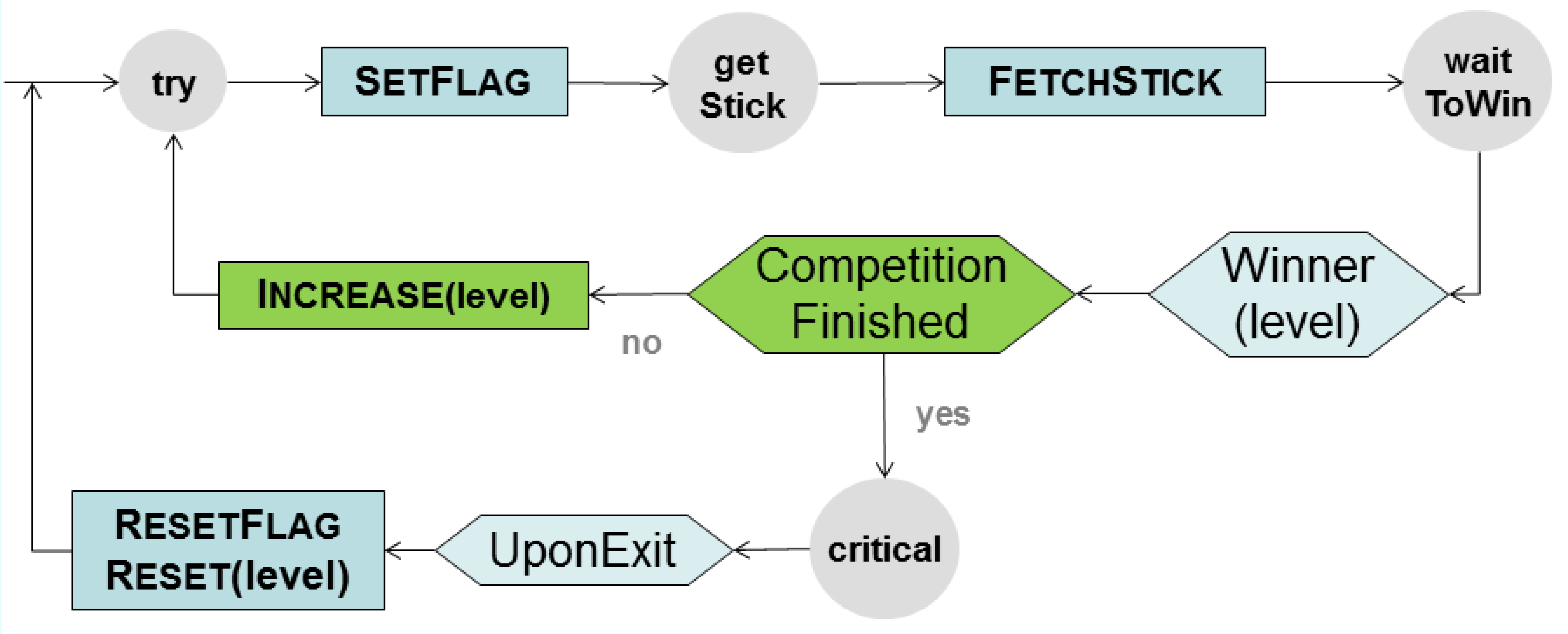
This means that $\textsc{MutexPeterson}_n$ is defined as an iterative version of $2\textsc{MutexPetersonAsm}$.

The generalized (iterated) competition at each $level$

To compete at the current $level_p \in \{1, \ldots, n-1\}$ (initially $level_p = 1$):

- the $flag_p$ location (initially $flag_p = 0$) is *set to $level$*, as interest indicator for this $level$, and can be read by *all* other processes
- the $stickHolder_{level} \in \{p_1, \ldots, p_n\}$ is parameterized by $level$ and shared by *all* processes
- the *$Winner$ strategy is parameterized* by $level$ and permits to $\textsc{Increase}(level)$
- *until $CompetitionFinished$* at $level_p = n - 1$, so that p can enter the critical section
- whereafter $UponExit$ it resets its $flag_p$ (to 0) and its $level_p$ (to 1) to return to its remainder section

$\textsc{MutexPeterson}_n$ control flow model



$\textsc{SetFlag} = (\mathit{flag}_{\textsf{self}} := \mathit{level}_{\textsf{self}}) \quad \textsc{Reset}(\mathit{level}) = (\mathit{level}_{\textsf{self}} := 1)$

$\mathit{CompetitionFinished}$ **iff** $\mathit{level}_{\textsf{self}} = n - 1$

$\textsc{Increase}(\mathit{level}) = (\mathit{level}_{\textsf{self}} := \mathit{level}_{\textsf{self}} + 1)$[2]

---

[2] Figure © 2016 Springer-Verlag Germany, reused with permission.

level-**parameterization of** *Winner* **strategy**

$Winner(level) = NobodyElseInterested(level_{\textbf{self}})$
or $MeantimeSomebodyElseFetchedStick(level_{\textbf{self}})$

$NobodyElseInterested(level)$ **iff forall** $p \neq$ **self** $flag_p < level$

-- generalizing $flag_{theOtherProcess(\textbf{self})} = 0$ (for $n = 2$)

$MeantimeSomebodyElseFetchedStick(level)$ **iff**
$stickHolder_{level} \neq$ **self**

-- generalizing $stickHolder = theOtherProcess(\textbf{self})$ (for $n = 2$)

Multiple-writer resolution is a Mutex resolution

- How to guarantee that in each step, at most one process p can (write $stickHolder_{level}$ to) become the $stickHolder_{level}$?

In Lynch op.cit. such a ‘multiple-writer resolution’ is hidden in the interleaving assumption, so that there $\textsc{FetchStick}$ is generalized by simply parameterizing it to $stickHolder_{level(\textbf{self})} := \textbf{self}$.

But this is a *petitio principii*, made visible by a $select$ion function—a form of external control!—which chooses *one* new $stickHolder$:

$$\textsc{FetchStick} = \textbf{if } chosenStickHolder_{level(\textbf{self})} = \textbf{self}$$

$$\quad \textbf{then } stickHolder_{level(\textbf{self})} := \textbf{self}$$

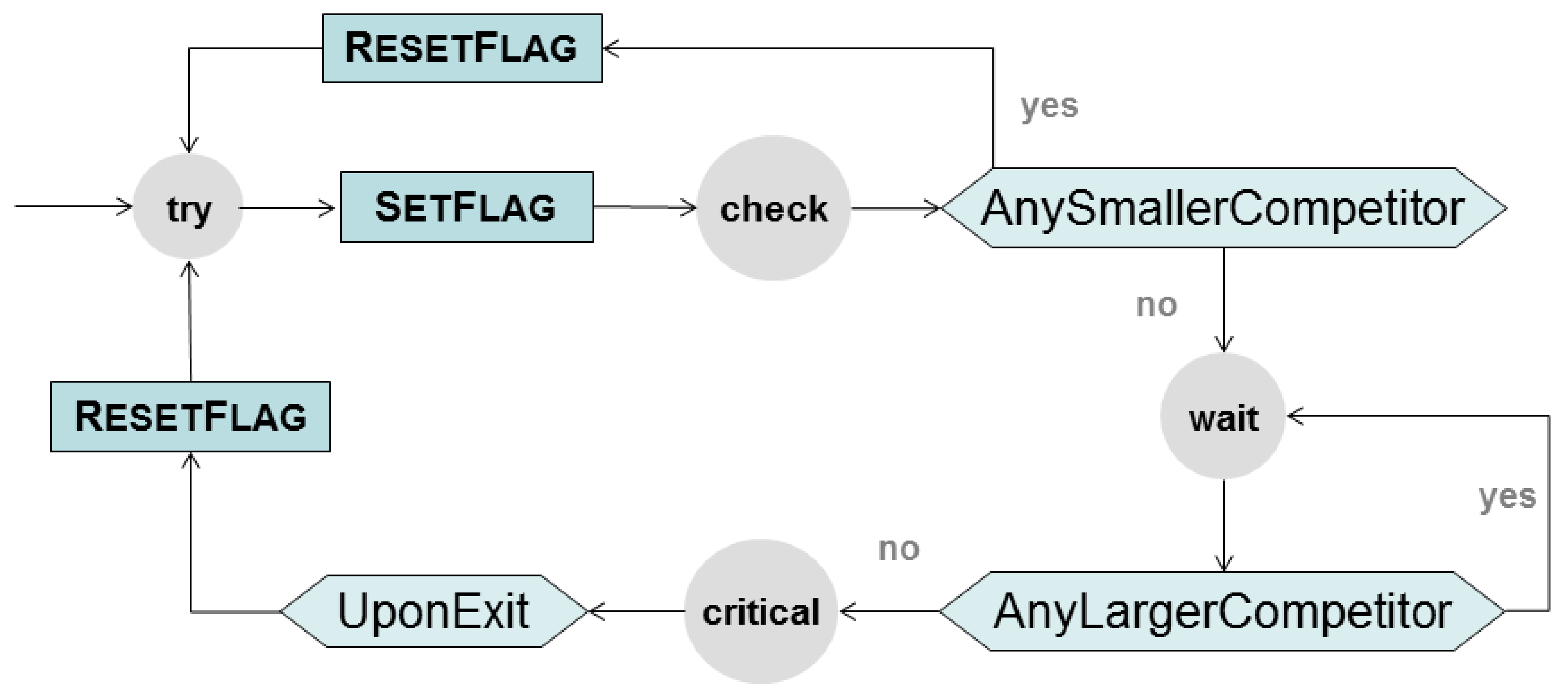
$$\textbf{where let } Cand = \{p \mid flag_p = level \textbf{ and } mode_p = getStick \\ \textbf{and } stickHolder_{level} \neq p\}$$

$$chosenStickHolder_{level} = \begin{cases} \textbf{undef} & \textbf{if } Cand = \emptyset \\ select(Cand) & \textbf{else} \end{cases}$$

MutexBurns with single-writer multiple-reader locs

A fresh competitor (who just set $flag := 1$) withdraws in case there is *AnySmaller Competitor* (for some $q < p \;\; flag_q = 1$).

A competitor who entered with no smaller competitor has to $wait$ as long as there is *AnyLargerCompetitor* (for some $q > p \;\; flag_q = 1$)



Analyzing MUTEXBURNS

Boolean-valued *shared flags are single-writer multiple-reader locations*.
No interleaving assumption is needed to show the following properties:

- Mutual Exclusion holds because
  - only the currently largest waiting competitor, say p, can enter the critical section CS, it remains competitor until exiting
    - no other (smaller) waiting competitor can enter the critical section
    - no larger process q can reach the wait phase because right after entering it must withdraw due to $p < q$
- Progress holds because if nobody is in the critical section CS:
  - a waiting competitor (the largest one) can enter CS
  - any competitor with no smaller competitor can enter phase $wait$
- No particular Fairness property is guaranteed.

NB. For a Mutex algorithm with single-writer multiple-reader locs which also satisfies Lockout-Freedom see Lamport's Bakery algorithm.

Exclusive Allocation of Multiple Resources

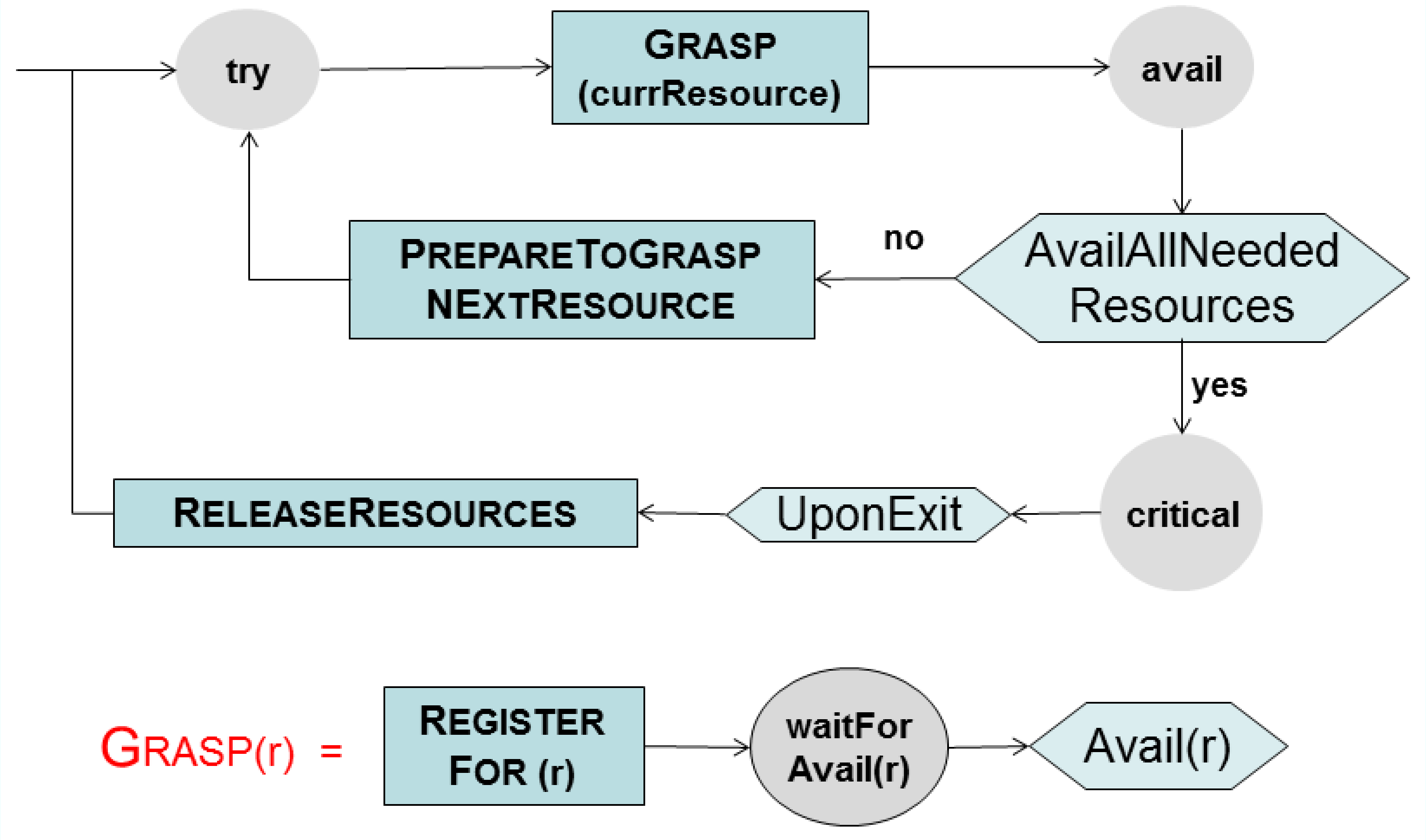
NB. Atomic simultaneous grasping of all resources is inadequate.

Algorithmic idea (Lynch 11.3):

- for each resource, competitors must register in a *FIFO* $queue(res)$
  - *shared by all processes* which may request the resource

  such that $Avail(res)_p$ iff $p = head(queue(res))$
  - Assumption: 'simultaneous insertion' of requests into a queue are sequentialized in an arbitrary manner (a queue plugin assumption)
- processes p must compete for the $neededResources$ one-by-one following a total resource order $<$ (*hierarchical resource allocation*) s.t.
  - $neededResources_p = res(1, p), \ldots, res(m_p, p)$
    - where $m_p = resQty_p$, $res(i, p) < res(i + 1, p)$
  - iterate $\textsc{Grasp}(currResource)$ over $neededResources$
    - where $currResource_p = res(currResNo_p, p)$ with local iterator variable $currResNo_p \in \{1, \ldots, resQty_p\}$ (initially $currResNo_p = 1$).

HIERARCHICALRESALLOC **algorithm**



NB. Generalizes the Dining Philosopher algorithm with 2 forks.

$\textsc{HierarchicalResAlloc}$ **predicates and actions**

$currResource = res(currResNo_{\textsf{self}}, \textsf{self})$

$\textsc{RegisterFor}(r) = \textsc{Insert}(\textsf{self}, queue(r))$

$Avail(r)$ iff $head(queue(r)) = \textsf{self}$

$AvailAllNeededResources = (currResNo_{\textsf{self}} = resQty_{\textsf{self}})$

$\textsc{PrepareToGraspNextResource} =$

$\quad currResNo_{\textsf{self}} := currResNo_{\textsf{self}} + 1$

$\textsc{ReleaseResources} =$

$\quad$ **forall** $r \in neededResources_{\textsf{self}}$ $\textsc{Dequeue}(\textsf{self}, queue(r))$

$\quad \textsc{ReInitialize}(currResNo)$ -- $currResNo_{\textsf{self}} := 1$

Analyzing $\textsc{HierarchicalResAlloc}$

Mutual Exclusion holds

- in fact, if p is in mode $critical$, for each needed resource p is the head of the $queue(res)$. Thus no other process can have $\textsc{Grasp}$ed any such resource until p has $\textsc{ReleaseResources}$ $UponExit$.

Lockout-Freedom holds

- since in each state of a run, the process p which holds (meaning $head(queue(r)) = p$) the largest resource r can make a move
  - in $mode = critical$, by run assumption p will eventually $\textsc{ReleaseResources}$
  - in $mode = try$, p can $\textsc{RegisterFor}(currResource)$, holding $r < currResource$
  - in $mode = waitForAvail(currResource)$, $Avail_p(currResource)$ must be true (because otherwise some other process would hold $currResource > r$) so that p can either move to $mode = critical$ or $\textsc{PrepareToGraspNextResource}$

References

- Nancy A. Lynch, *Distributed Algorithms*.
  – Morgan Kaufmann, San Franciso 1996
  See also material for course 6.852 at MIT, Fall 2013
- E. Börger and K.-D. Schewe: *Concurrent Abstract State Machines*. Acta Informatica 53 (2016), 469-492
- E. Börger: Modeling Distributed Algorithms by Abstract State Machines Compared to Petri Nets
  – Springer LNCS 9675 (2016) 3-34
- E. Börger and A. Raschke: Modeling Companion for Software Practitioners. Springer 2018
  `http://modelingbook.informatik.uni-ulm.de`

Copyright Notice