Lezione 27 MSC
Multi-CCS (1/2)

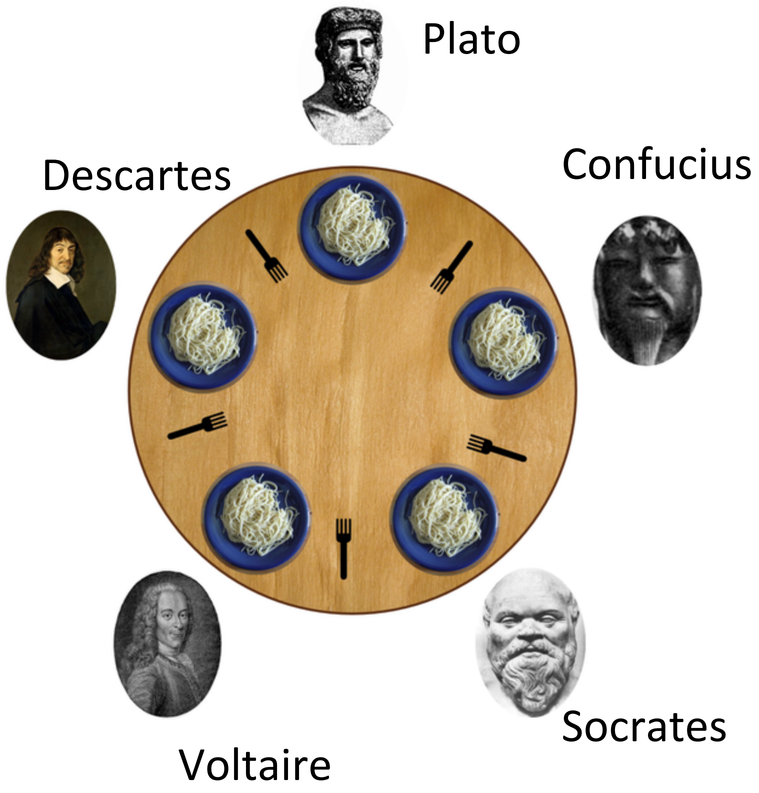
Roberto Gorrieri

Lack of expressiveness of CCS

- CCS is a Turing-complete formalism, i.e., it has the ability to compute all the Turing-computable functions. Therefore, one may think that it is able to solve any kind of problems.
- Unfortunately this is not the case: Turing- completeness is not enough to ensure the solvability of all the problems in concurrency theory.
- For instance, it is well-known that a classic solution to the famous dining philosophers problem that assumes atomicity in the acquisition of the forks (or, equivalently, that requires a three-way synchronization among one philosopher and the two forks), cannot be given in CCS.

The dining philosophers problem
(Dijkstra – Hoare)

- Five philosophers sit at a round table, with a bowl of spaghetti in the middle; each philosopher is equipped with a private plate; there are only five forks, each one placed between two adjacent neighbors.
- Philosophers can think and eat; in order to eat spaghetti, a philosopher has to acquire both forks that he shares with his neighbors
- Eating can happen in ***mutual exclusion*** w.r.t. the neighbors, so that at most two philosophers can eat at the same time.
- After a philosopher finishes eating, he has to put down both forks, so that they become available to his neighbors.
- The various kinds of failures these philosophers may experience are analogous to the difficulties that arise in real computer programming when multiple programs need exclusive access to shared resources.



Requirements

- The problem is to conceive a suitable algorithm that satisfies at least one of the following properties:
- *Deadlock-freeness*: no reachable state is a deadlock; this ensures that all the computations can be extended *ad infinitum*;
- *weak non-starvation*: not only all the computations can be extended *ad infinitum*, but also for any never-ending computation there is at least one philosopher that eats infinitely often; this means that not all hungry philosophers starve.
- *strong non-starvation*: as above, with the additional constraint that for any computation each time a philosopher wants to eat, he will eat eventually; this means that no hungry philosopher will starve.

- We are interested to a solution to this problem that satisfies the following constraints:
  - *fully distributed*: there is no central memory to which all the philosophers may have access, nor a global scheduler that coordinates the activities of the philosophers;
  - *symmetric*: all philosophers are identical.

First tentative solution in CCS

For simplicity's sake, we consider the case with two philosophers only

The forks can be described as

$$F_i \overset{def}{=} \overline{up_i}.\overline{dn_i}.F_i \quad \text{for } i = 0, 1$$

where the complementary action up_i (pick fork$_i$ up) and dn_i (put fork$_i$ down) are to be performed by the philosopher willing to use that fork.
The two philosophers can be described as

$$P_i \overset{def}{=} think.P_i + up_i.up_{i+1}.eat.dn_i.dn_{i+1}.P_i \quad \text{for } i = 0, 1$$

where $i + 1$ is computed modulo 2 (i.e., up_2 *mod* 2 is up_0). A philosopher can think or can begin the procedure for the acquisition of both forks, starting form the one with his same index (we may assume it is the one on his right); when he has got both forks, then he can eat, and when he has finished eating, he has to put down both forks in the same order in which they have been grabbed.

What's wrong? Deadlock!

- The whole system is $DP \overset{def}{=} (\nu L)(((P_0 \mid P_1) \mid F_0) \mid F_1)$
  where *L* = {*up*0,*up*1,*dn*0,*dn*1}.
- Clearly this naıve solution would cause a deadlock exactly when the two philosophers take the fork at their right at the same time and are waiting for the fork at their left. (See the next slide for illustration of the associated lts.)
- The state *DPd* is a deadlock:

$$
\begin{aligned}
DP_d &\overset{def}{=} (\nu L)(((P_0' \mid P_1') \mid F_0') \mid F_1') \\
P_i' &\overset{def}{=} up_{i+1}.eat.dn_i.dn_{i+1}.P_i \qquad \text{for } i = 0, 1 \\
F_i' &\overset{def}{=} \overline{dn_i}.F_i \qquad \text{for } i = 0, 1
\end{aligned}
$$

The lts for DP (with deadlock)

$$\begin{tikzcd}
 & & DP_d & & \\
DP_5 \arrow[loop above, "think"] \arrow[d, bend right, "eat"'] & DP_4 \arrow[loop above, "think"] \arrow[l, "\tau"'] \arrow[ur, bend left, "\tau"] & DP \arrow[loop above, "think"] \arrow[l, "\tau"'] \arrow[r, "\tau"] & DP_0 \arrow[loop above, "think"] \arrow[r, "\tau"] \arrow[ul, bend right, "\tau"'] & DP_1 \arrow[loop above, "think"] \arrow[d, bend left, "eat"] \\
DP_6 \arrow[loop below, "think"] \arrow[r, "\tau"] & DP_7 \arrow[loop below, "think"] \arrow[ur, "\tau"'] & & DP_3 \arrow[loop below, "think"] \arrow[ul, "\tau"] & DP_2 \arrow[loop below, "think"] \arrow[l, "\tau"']
\end{tikzcd}$$

Second tentative solution in CCS

A well-known solution to this problem is to break the symmetry by inverting the order of acquisition of the forks for the last philosopher. In our restricted case with two philosophers only, we have that

$$P_0'' \overset{def}{=} think.P_0'' + up_0.up_1.eat.dn_0.dn_1.P_0''$$

$$P_1'' \overset{def}{=} think.P_1'' + up_0.up_1.eat.dn_1.dn_0.P_1''$$

and the whole system is $DP' \overset{def}{=} (\nu L)(((P_0'' \,|\, P_1'') \,|\, F_0) \,|\, F_1)$

This solution works correctly (i.e., it satisfies deadlock-freeness – see next slide for its associated lts), but it is not compliant to the specification that requires that all philosophers are defined in the same way: the solution is not symmetric!

The LTS associated to DP’ (deadlock-free asymmetric solution)

$$\begin{tikzcd}
DP'_5 \arrow[loop above, "think"] \arrow[d, bend right, "eat"'] & DP'_4 \arrow[loop above, "think"] \arrow[l, "\tau"'] & DP' \arrow[loop above, "think"] \arrow[l, "\tau"'] \arrow[r, "\tau"] & DP'_0 \arrow[loop above, "think"] \arrow[r, "\tau"] & DP'_1 \arrow[loop above, "think"] \arrow[d, bend left, "eat"] \\
DP'_6 \arrow[loop below, "think"] \arrow[r, "\tau"'] & DP'_7 \arrow[loop below, "think"] \arrow[ur, "\tau"'] & & DP'_3 \arrow[loop below, "think"] \arrow[ul, "\tau"] & DP'_2 \arrow[loop below, "think"] \arrow[l, "\tau"]
\end{tikzcd}$$

Fig. 6.2 The deadlock-free asymmetric solution of the two dining philosophers problem.

Third attempt

A simple, well-known solution is to force atomicity on the acquisition of the two forks so that either both are taken or none. This requirement can be satisfied approximately in CCS as follows:

$$P_i''' \overset{def}{=} think.P_i''' + up_i.(dn_i.P_i''' + up_{i+1}.eat.dn_i.dn_{i+1}.P_i''') \quad \text{for } i = 0, 1$$

where, in case the second fork is unavailable, the philosopher may put down the first fork and return to its initial state. However, the new system

$$DP'' \overset{def}{=} (\nu L)(((P_0''' \mid P_1''') \mid F_0) \mid F_1)$$

even if deadlock-free, may now diverge: the two philosophers may be engaged in a never-ending divergent computation because the long operation of acquisition of the two forks may always fail.

Third solution with divergence

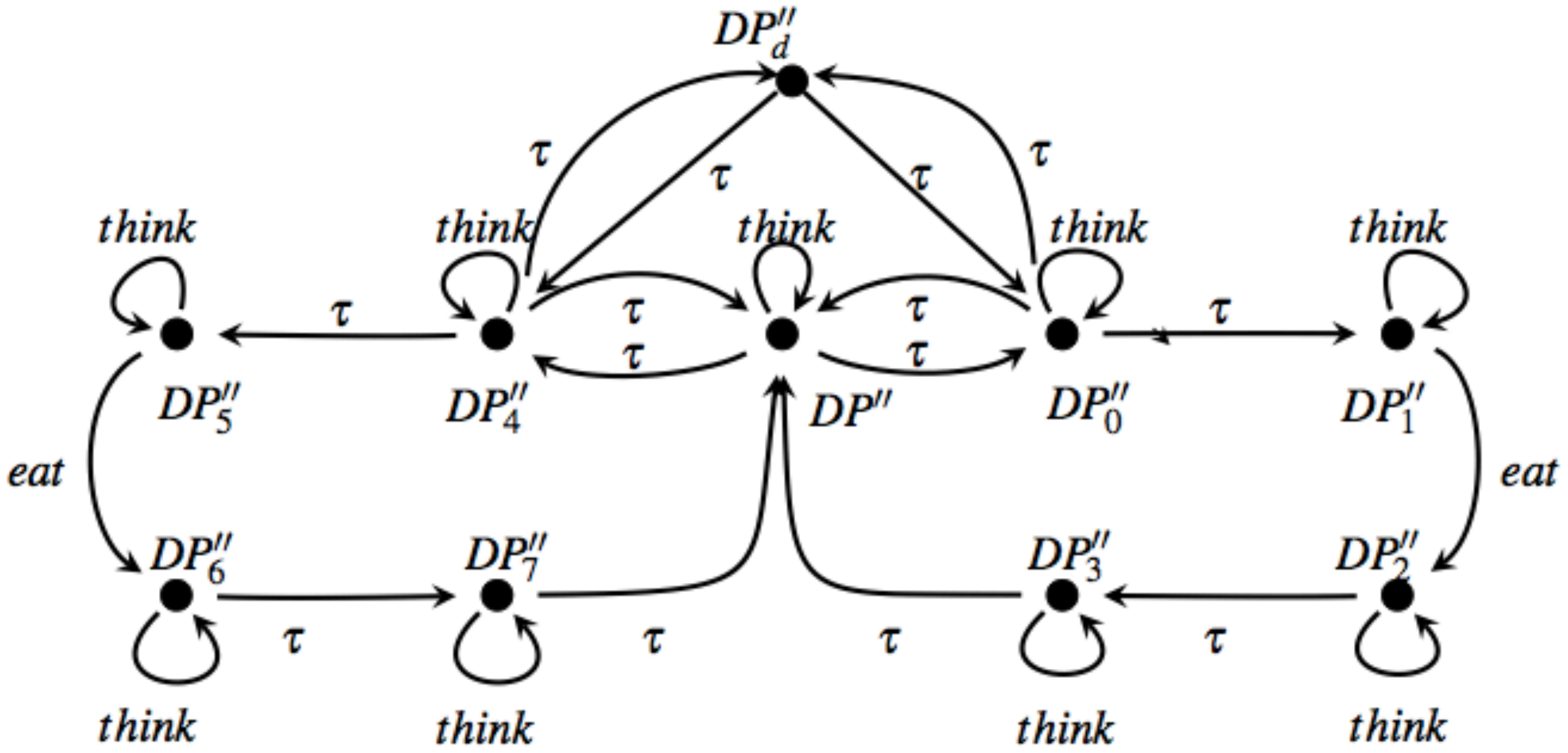


Fig. 6.3 The symmetric solution of the two dining philosophers problem – with divergence.

Is DP’’ a sensible solution?

- It is not difficult to prove that the deadlock-free, asymmetric solution *DP*’ and the divergent, symmetric solution *DP*’’are weakly bisimilar, so that, to some extent, we could consider *DP*’’ a reasonable solution as well.
- This equality makes sense under an assumption of fairness: if a philosopher has the possibility of acquiring its second fork infinitely often, then he will eventually grab it. Indeed, any never-ending internal computation must pass through state DP_0'' or DP_4'' infinitely often so that, by fairness, we can assume that the other fork will be eventually grabbed.
- Nonetheless, divergence is a possible behavior and it is unrealistic in this setting to ignore it.
- A livelock is possible when the two philosophers work somehow synchronously: both acquire the fork on their right at the same time and then, since the other fork is unavailable, both put down their fork at the same time, reaching the initial state and so possibly repeating this cycle forever. The never-ending computation that alternates between states *DP*’’ and DP_d'' is such that both philosophers never have the possibility of grabbing the other fork, so that fairness cannot be invoked in this case.

No satisfactory solution in CCS

- Unfortunately, a deadlock-free, divergence-free solution that implements correctly the atomic acquisition of the two forks cannot be programmed in CCS because it lacks any construct for atomicity that would also enable a multi-party synchronization between one philosopher and the two forks.
- Indeed, on the one hand, a symmetric, fully distributed, deterministic, deadlock-free and divergence-free solution to the dining philosophers problem can be programmed in a language with multi-party synchronization, such as CSP (see next slide)
- On the other hand, Lehmann and Rabin demonstrated that a symmetric, fully distributed, deterministic, deadlock-free (and divergence-free) solution does not exist in a language with only binary synchronization such as CCS.
- Hence, if we want to solve this problem, we have to extend the capabilities of CCS.

Solution in CSP (1)

The five philosophers can be described by the constants P_i as follows:

$$P_i \overset{def}{=} think.P_i + up_{(i,i+1)}.eat.dn_{(i,i+1)}.P_i \quad \text{for } i = 0, \ldots, 4$$

where index i + 1 is computed modulo 5 and the actions are indexed by a pair of numbers in the range {0,...,4}: action *up*(*i*,*i*+1) denotes the philosopher (atomic) action of grabbing the fork of index *i* together with the fork of index *i* + 1.

Process *Phils* can be defined as follows:

$$Phils \overset{def}{=} P_0 \parallel_\emptyset P_1 \parallel_\emptyset P_2 \parallel_\emptyset P_3 \parallel_\emptyset P_4$$

where no synchronization can take place among the philosophers.

Solution in CSP (2)

The five forks can be described by the constants F_i as follows:

$$F_i \stackrel{def}{=} up_{(i,i+1)}.dn_{(i,i+1)}.F_i + up_{(i-1,i)}.dn_{(i-1,i)}.F_i \quad \text{for } i = 0, \ldots, 4$$

where indexes *i* + 1 and *i* -1 are computed modulo 5. Action *up*(*i*,*i*+1) denotes availability of the fork of index *i* to be grabbed, together with the fork of index *i* + 1, by philosopher of index *i*; analogously, action *up*(*i*-1,*i*) denotes the availability of the fork of index *i* to be grabbed, together with the fork of index *i*-1, by philosopher of index *i*-1. This means that forks *Fi* and *Fi*+1 shares two similar summands: *up*(*i*,*i*+1).*dn*(*i*,*i*+1).*Fi* and *up*(*i*,*i*+1).*dn*(*i*,*i*+1).*Fi*+1, respectively, onto which they must synchronize.

$$Forks \stackrel{def}{=} F_0 \parallel_A F_1 \parallel_A F_2 \parallel_A F_3 \parallel_A F_4$$

- where *A* = {*up*(*i*,*i*+1),*dn*(*i*,*i*+1) *i* = 0,...4}. Therefore, e.g., fork *F*0 must synchronize either with fork *F*1 (on actions *up*(0,1),*dn*(0,1)) or with fork *F*4 (on actions *up*(4,0),*dn*(4,0)).

Solution in CSP (3)

The whole system composing the forks and the philosophers is

$$DP_{CSP} \overset{def}{=} (\iota A)(Phils \parallel_A Forks)$$

where the hiding operator internalizes all the ternary synchronizations occurring among each philosopher and his two forks.

This solution is fully-distributed, symmetric, deadlock-free and divergence-free.

Exercise: Draw the lts associated to DP_{CSP} and show that it is deadlock-free and divergence-free.

Extending CCS: Multi-CCS

- One additional operator: Strong prefixing: $\underline{\alpha}.p$ where α is the first action of a transaction that continues with *p* (provided that *p* can complete the transaction). Hence, transitions are labeled by sequences of observable actions!
- Side effect: Multi-party synchronization: obtained as an atomic transaction of binary CCS-like synchronizations.
- Expressive power considerably increased: correct solution of the dining philosophers, as well as many other features we will see in the following.

Strong Prefixing

- One additional operator: Strong prefixing: $\underline{\alpha}.p$ where α is the first action of a transaction that continues with *p* (provided that *p* can complete the transaction).

$$\text{(S-Pref)}\ \frac{p \xrightarrow{\sigma} p'}{\underline{\alpha}.p \xrightarrow{\alpha \diamond \sigma} p'} \quad \text{where} \quad \alpha \diamond \sigma = \begin{cases} \alpha & \text{if } \sigma = \tau, \\ \alpha\sigma & \text{otherwise,} \end{cases}$$

$$\text{(S-Pref)}\ \cfrac{\text{(Sum}_1\text{)}\ \cfrac{\text{(Pref)}\ \cfrac{}{b.\mathbf{0} \xrightarrow{b} \mathbf{0}}}{b.\mathbf{0} + c.\mathbf{0} \xrightarrow{b} \mathbf{0}}}{\underline{a}.(b.\mathbf{0} + c.\mathbf{0}) \xrightarrow{ab} \mathbf{0}} \qquad \text{(S-Pref)}\ \cfrac{\text{(Sum}_2\text{)}\ \cfrac{\text{(Pref)}\ \cfrac{}{c.\mathbf{0} \xrightarrow{c} \mathbf{0}}}{b.\mathbf{0} + c.\mathbf{0} \xrightarrow{c} \mathbf{0}}}{\underline{a}.(b.\mathbf{0} + c.\mathbf{0}) \xrightarrow{ac} \mathbf{0}}$$

Philosopher with atomic acquisition of the two forks

$$P_i \overset{def}{=} think.P_i + \underline{up_i}.up_{i+1}.eat.\underline{dn_i}.dn_{i+1}.P_i \quad \text{for } i = 0,1$$

$$\begin{tikzcd}
 & P_i \arrow[loop left, "think"] \arrow[dr, bend left, "up_i up_{i+1}"] & \\
\underline{dn_i}.dn_{i+1}.P_i \arrow[ur, bend left, "dn_i dn_{i+1}"] & & eat.\underline{dn_i}.dn_{i+1}.P_i \arrow[ll, bend left, "eat"]
\end{tikzcd}$$

Is atomicity ensured?

The execution of a sequence of actions is atomic if such execution is:

- ***All-or-nothing***: the sequence is either executed completely, or not at all; this implies that the process is not observable during the execution of such a sequence, but only before and after that.
- ***Non-interruptible***: no other process can interrupt its execution; this implies that the atomic sequence is never interleaved with others.
- Of course, strong prefixing ensures the all-or-nothing property: by rule (S-Pref), $\underline{a}.b.0$ can only perform a transition to 0 labeled by *ab* and its execution is either completed or not done at all; the semantic model does not represent the intermediate state of the execution where action *a* has been performed but *b* not yet. Moreover, the situation when the atomic execution may fail in the middle and needs recovery is not modeled at all.
- And what about non-interruptibility?

Is atomicity ensured? (2)

What happens when we put a process $\underline{\alpha}.p$ in parallel with another process?

$$\begin{tikzcd}
 & \underline{a}.b.\mathbf{0} \mid c.\mathbf{0} \arrow[dl, "ab"'] \arrow[dr, "c"] & \\
\mathbf{0} \mid c.\mathbf{0} \arrow[dr, "c"'] & & \underline{a}.b.\mathbf{0} \mid \mathbf{0} \arrow[dl, "ab"] \\
 & \mathbf{0} \mid \mathbf{0} &
\end{tikzcd}$$

$$\begin{tikzcd}
 & a.b.\mathbf{0} \mid c.\mathbf{0} \arrow[dl, "a"'] \arrow[dr, "c"] & \\
b.\mathbf{0} \mid c.\mathbf{0} \arrow[d, "b"'] \arrow[drr, "c"] & & a.b.\mathbf{0} \mid \mathbf{0} \arrow[d, "a"] \\
\mathbf{0} \mid c.\mathbf{0} \arrow[dr, "c"'] & & b.\mathbf{0} \mid \mathbf{0} \arrow[dl, "b"] \\
 & \mathbf{0} \mid \mathbf{0} &
\end{tikzcd}$$

hence non-interruptibility is ensured, indeed.

Multi-party synch in Multi-CCS

- Generalization of rule (Com), as now transitions are labeled on sequences of actions

$$\text{(S-Com)}\ \frac{p \xrightarrow{\sigma_1} p' \qquad q \xrightarrow{\sigma_2} q'}{p \,|\, q \xrightarrow{\sigma} p' \,|\, q'}\ \mathit{Sync}(\sigma_1, \sigma_2, \sigma)$$

side-condition on the possible synchronizability of sequences σ1 and σ2, whose result may be σ

- We require that Sync(σ1, σ2, σ) holds if at least one of the two sequences is a single action, say σ1 = α , to be synchronized with the first occurrence of its complementary co-action ‘α within the sequence σ2. Note that it is not possible to synchronize two sequences.

Synchronization relation

$$\frac{}{Sync(\alpha,\overline{\alpha},\tau)} \qquad \frac{\sigma \neq \varepsilon}{Sync(\alpha\sigma,\overline{\alpha},\sigma)} \qquad \frac{\sigma \neq \varepsilon}{Sync(\overline{\alpha},\alpha\sigma,\sigma)}$$

$$\frac{Sync(\sigma,\overline{\alpha},\tau)}{Sync(\beta\sigma,\overline{\alpha},\beta)} \qquad \frac{Sync(\overline{\alpha},\sigma,\tau)}{Sync(\overline{\alpha},\beta\sigma,\beta)} \qquad \frac{Sync(\sigma,\overline{\alpha},\sigma_1)}{Sync(\beta\sigma,\overline{\alpha},\beta\sigma_1)} \qquad \frac{Sync(\overline{\alpha},\sigma,\sigma_1)}{Sync(\overline{\alpha},\beta\sigma,\beta\sigma_1)}$$

Table 6.1 Synchronization relation *Sync*, where $\beta \neq \alpha$

$$\frac{bac \neq \varepsilon}{Sync(abac,\overline{a},bac)} \qquad \cfrac{\cfrac{\cfrac{\cfrac{}{Sync(c,\overline{c},\tau)}}{Sync(ac,\overline{c},a)}}{Sync(bac,\overline{c},ba)}}{Sync(abac,\overline{c},aba)}$$

Note that $Sync(abac,\overline{a},abc)$ is not derivable because the synchronization may take place with the first occurrence of a only, as shown above; indeed, the four rules of the second row of Table 6.1 are applicable only if $\beta \neq \alpha$.

This example also shows that *Sync* is a (partial) function of its first two arguments, as the result of the synchronization of a sequence with an action, if defined, is unique. □

Dining philosophers in Multi-CCS

$$P_i \overset{def}{=} think.P_i + \underline{up_i}.up_{i+1}.eat.\underline{dn_i}.dn_{i+1}.P_i \quad \text{for } i = 0, 1$$

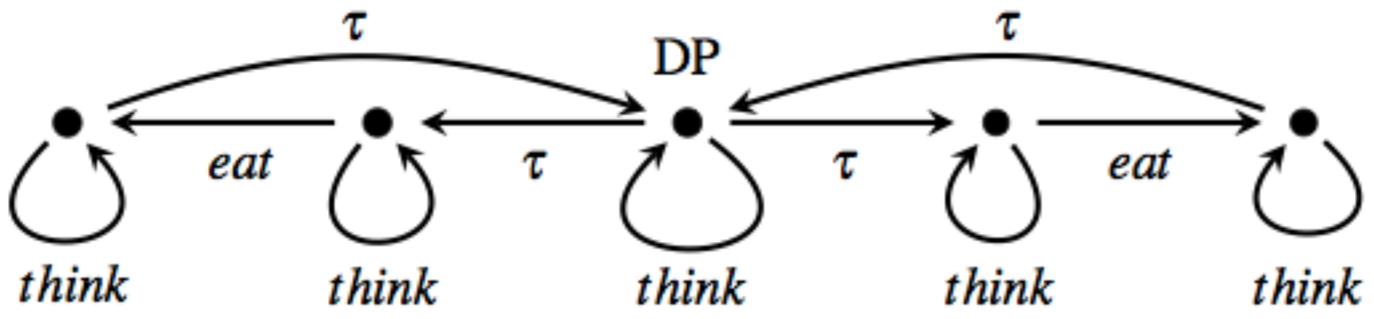
$$F_i \overset{def}{=} \overline{up_i}.\overline{dn_i}.F_i \quad \text{for } i = 0, 1 \qquad DP \overset{def}{=} (\nu L)(((P_0 \,|\, P_1) \,|\, F_0) \,|\, F_1)$$

$$P'_i = eat.\underline{dn_i}.dn_{i+1}.P_i \text{ and } F'_i = \overline{dn_i}.F_i$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{up_1.P'_0 \xrightarrow{up_1} P'_0}}{\underline{up_0}.up_1.P'_0 \xrightarrow{up_0 up_1} P'_0}}{think.P_0 + \underline{up_0}.up_1.P'_0 \xrightarrow{up_0 up_1} P'_0}}{P_0 \xrightarrow{up_0 up_1} P'_0}}{P_0 \,|\, P_1 \xrightarrow{up_0 up_1} P'_0 \,|\, P_1} \quad \dfrac{\dfrac{}{\overline{up_0}.F'_0 \xrightarrow{\overline{up_0}} F'_0}}{F_0 \xrightarrow{\overline{up_0}} F'_0}}{(P_0 \,|\, P_1) \,|\, F_0 \xrightarrow{up_1} (P'_0 \,|\, P_1) \,|\, F'_0} \quad \dfrac{\dfrac{}{\overline{up_1}.F'_1 \xrightarrow{\overline{up_1}} F'_1}}{F_1 \xrightarrow{\overline{up_1}} F'_1}}{((P_0 \,|\, P_1) \,|\, F_0) \,|\, F_1 \xrightarrow{\tau} ((P'_0 \,|\, P_1) \,|\, F'_0) \,|\, F'_1}}{(\nu L)(((P_0 \,|\, P_1) \,|\, F_0) \,|\, F_1) \xrightarrow{\tau} (\nu L)(((P'_0 \,|\, P_1) \,|\, F'_0) \,|\, F'_1)}}{DP \xrightarrow{\tau} (\nu L)(((P'_0 \,|\, P_1) \,|\, F'_0) \,|\, F'_1)}$$

Dining philosophers in Multi-CCS (2)

The solution is correct: fully-distributed, symmetric deadlock-free and divergence-free



Is DP a fully distributed solution?

- It may debatable if this solution, based on multi-party synchronization, is to be considered fully distributed.
- On the one hand, the Multi-CCS specification *DP* is not using any shared memory or global coordinator, so that we can say that the *specification DP* is distributed, at this abstract level of description.
- On the other hand, a truly distributed, deterministic *implementation* of the multi-party synchronization mechanism seems not to exist.
- The same argument can be used about the CSP solution.

Is DP a correct solution?

DP does not ensure *weak non-starvation* because there are never-ending computations in which no philosopher eats infinitely often: e.g., one philosopher, after having eaten, never releases the forks and the other thinks forever. In order to overcome this problem, one can define a more realistic version of the philosopher which alternates between the two phases of thinking and eating. The *alternating* philosopher AP_i is defined as follows:

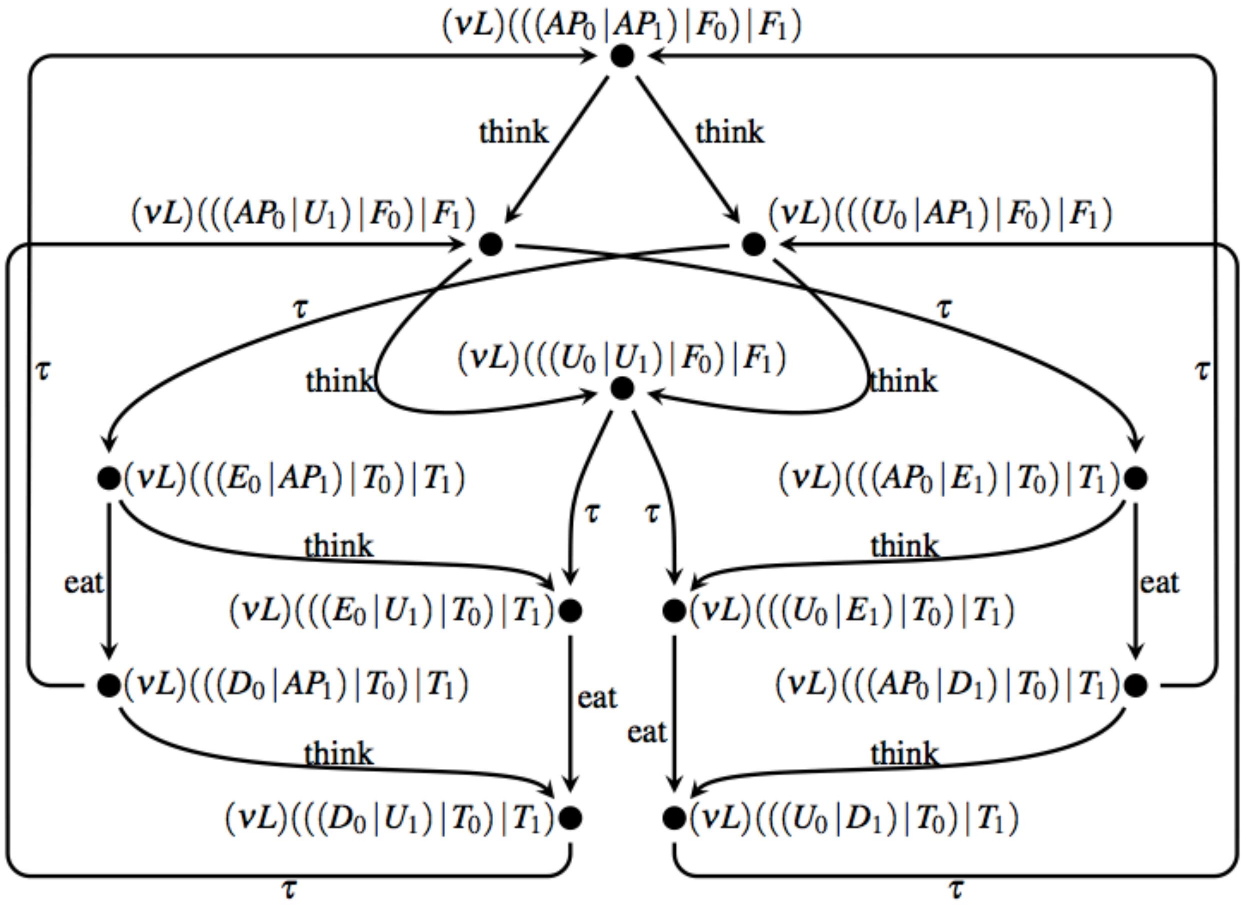
$$AP_i \overset{def}{=} think.U_i \qquad U_i \overset{def}{=} \underline{up_i}.up_{i+1}.E_i$$
$$E_i \overset{def}{=} eat.D_i \qquad D_i \overset{def}{=} \underline{dn_i}.dn_{i+1}.AP_i$$

where he first thinks, then acquires the forks, then eats, then releases the forks in a cyclic, never-ending behavior. Contrary to what happens for *Pi*, progress is not always ensured: if the forks are unavailable, the alternating philosopher is stuck. The whole system *ADP* can be defined as:

$$ADP \overset{def}{=} (\nu L)(((AP_0 \,|\, AP_1) \,|\, F_0) \,|\, F_1)$$

ADP satisfies weak non-starvation

ADP satisfies the weak non-starvation property: in any never-ending computation at least one philosopher eats infinitely often.



$(\nu L)(((AP_0 \mid AP_1) \mid F_0) \mid F_1)$
think
think
$(\nu L)(((AP_0 \mid U_1) \mid F_0) \mid F_1)$
$(\nu L)(((U_0 \mid AP_1) \mid F_0) \mid F_1)$
τ
τ
τ
τ
think
$(\nu L)(((U_0 \mid U_1) \mid F_0) \mid F_1)$
think
$(\nu L)(((E_0 \mid AP_1) \mid T_0) \mid T_1)$
$(\nu L)(((AP_0 \mid E_1) \mid T_0) \mid T_1)$
τ
τ
think
think
eat
$(\nu L)(((E_0 \mid U_1) \mid T_0) \mid T_1)$
$(\nu L)(((U_0 \mid E_1) \mid T_0) \mid T_1)$
eat
$(\nu L)(((D_0 \mid AP_1) \mid T_0) \mid T_1)$
eat
eat
$(\nu L)(((AP_0 \mid D_1) \mid T_0) \mid T_1)$
think
think
$(\nu L)(((D_0 \mid U_1) \mid T_0) \mid T_1)$
$(\nu L)(((U_0 \mid D_1) \mid T_0) \mid T_1)$
τ
τ

Courteous dining philosophers

- Strong non-starvation: for any computation each time a philosopher wants to eat, he will eat eventually, that is no hungry philosopher will starve.
- How to ensure strong non-starvation? By using auxiliary variables shared by neighbour philosophers expressing:
  - Desire to eat: $S(i,i+1)$, lets Pi to inform $Pi+1$ of his desire to eat (values *on*/*off*), and vice versa, variable $S(i+1,i)$ informs Pi that $Pi+1$ wishes (or not) to eat.
  - Who has eaten last: $K(i,i+1)$ shows which of the two has eaten last (values $i/i+1/$*neutral*, the last being the dummy initial value).

Courteous dining philosophers (2)

When a philosopher *Pi* gets hungry, he first declares his desire to eat to his two neighbors by setting variables *S*(*i*,*i*+1) and *S*(*i*,*i*-1) to *on*; then he tries to perform a multiway synchronization with the two forks, provided that the following conditions are satisfied:

- *S*(*i*+1,*i*) is set to *off* or *K*(*i*,*i*+1) is set to *i* + 1 or *neutral*; this means that the philosopher *Pi*+1 is either not willing to eat or has been served last (or neither has been served yet);
- similarly, *S*(*i*-1,*i*) is set to *off* or *K*(*i*-1,*i*) is set to *i*-1 or *neutral*; this means that the philosopher *Pi*-1 is either not willing to eat or has been served last (or neither has been served yet).

After eating, philosopher *Pi* puts down the forks and updates the variables he shares with his neighbors in the obvious way (no longer hungry and last to eat).

CDP in (value-passing) Multi-CCS

- For simplicity sake, two philosophers only. This solution is slightly simpler than the general parametric one for *n*≥3. The whole system is:

$$CDP \overset{def}{=} (\nu L)(P_0 \,|\, S_{(0,1)}(off) \,|\, K_{(0,1)}(neutral) \,|\, S_{(1,0)}(off) \,|\, P_1 \,|\, F_0 \,|\, F_1)$$

- For (*i*, *j*) = (0,1),(1,0), the signal variable *S*(*i*, *j*)(*x*) is defined as:

$$S_{(i,j)}(x) \overset{def}{=} \overline{r_{(i,j)}}(x).S_{(i,j)}(x) + w_{(i,j)}(on).S_{(i,j)}(on) + w_{(i,j)}(off).S_{(i,j)}(off)$$

- The variable *K*(0,1) is defined as follows:

$$K_{(0,1)}(x) \overset{def}{=} \overline{l_{(0,1)}}(x).K_{(0,1)}(x) + v_{(0,1)}(0).K_{(0,1)}(0) + v_{(0,1)}(1).K_{(0,1)}(1)$$

CDP in (value-passing) Multi-CCS (2)

- *Pi* first thinks and then declares his intention to eat by setting *S*(*i*,*i*+1) to *on* and, at the same time, by performing the observable action $will_i$.
- Now process *Pi''* tries to perform a long transaction, starting with the atomic acquisition of the two forks and then ending with a safety check: *Qi* checks that either *S*(*i*+1,*i*) is set to *off*, or *K*(*i*,*i*+1) is set to *neutral* or to *i* + 1; if this four-way synchronization is successful, then *Pi''* reaches state *Qi'*, i.e., the philosopher can eat now, reaching state Pi'''.

$$
\begin{aligned}
P_i &\overset{def}{=} think_i.P_i' \\
P_i' &\overset{def}{=} \underline{\overline{w_{i,i+1}}}(on).will_i.P_i'' \\
P_i'' &\overset{def}{=} \underline{up_i}.\underline{up_{i+1}}.Q_i \\
Q_i &\overset{def}{=} r_{(i+1,i)}(off).Q_i' + l_{(i,i+1)}(neutral).Q_i' + l_{(i,i+1)}(i+1).Q_i' \\
Q_i' &\overset{def}{=} eat_i.P_i''' \\
P_i''' &\overset{def}{=} \underline{\overline{w_{(i,i+1)}}}(off).\underline{\overline{v_{(i,i+1)}}}(i).\underline{dn_i}.dn_{i+1}.P_i
\end{aligned}
$$

$$\begin{aligned}
CDP \xrightarrow{think_0} \xrightarrow{think_1} \; & (\nu L)(P'_0 \,|\, S_{(0,1)}(\mathit{off}) \,|\, K_{(0,1)}(\mathit{neutral}) \,|\, S_{(1,0)}(\mathit{off}) \,|\, P'_1 \,|\, F_0 \,|\, F_1) \\
\xrightarrow{will_0} \xrightarrow{will_1} \; & (\nu L)(P''_0 \,|\, S_{(0,1)}(\mathit{on}) \,|\, K_{(0,1)}(\mathit{neutral}) \,|\, S_{(1,0)}(\mathit{on}) \,|\, P''_1 \,|\, F_0 \,|\, F_1) \\
\xrightarrow{\tau} \; & (\nu L)(Q'_0 \,|\, S_{(0,1)}(\mathit{on}) \,|\, K_{(0,1)}(\mathit{neutral}) \,|\, S_{(1,0)}(\mathit{on}) \,|\, P''_1 \,|\, T_0 \,|\, T_1) \\
\xrightarrow{eat_0} \; & (\nu L)(P'''_0 \,|\, S_{(0,1)}(\mathit{on}) \,|\, K_{(0,1)}(\mathit{neutral}) \,|\, S_{(1,0)}(\mathit{on}) \,|\, P''_1 \,|\, T_0 \,|\, T_1) \\
\xrightarrow{\tau} \; & (\nu L)(P_0 \,|\, S_{(0,1)}(\mathit{off}) \,|\, K_{(0,1)}(0) \,|\, S_{(1,0)}(\mathit{on}) \,|\, P''_1 \,|\, F_0 \,|\, F_1) \\
\xrightarrow{think_0} \xrightarrow{will_0} \; & (\nu L)(P''_0 \,|\, S_{(0,1)}(\mathit{on}) \,|\, K_{(0,1)}(0) \,|\, S_{(1,0)}(\mathit{on}) \,|\, P''_1 \,|\, F_0 \,|\, F_1)
\end{aligned}$$

but now the reached state is such that only P''_1 can perform the fork acquisition transaction; P''_0 cannot because $S_{(1,0)}$ is set to *on* and $K_{(0,1)}$ records that P''_0 was served last. Then

$$\begin{aligned}
& (\nu L)(P''_0 \,|\, S_{(0,1)}(\mathit{on}) \,|\, K_{(0,1)}(0) \,|\, S_{(1,0)}(\mathit{on}) \,|\, P''_1 \,|\, F_0 \,|\, F_1) \\
& \qquad \xrightarrow{\tau} (\nu L)(P''_0 \,|\, S_{(0,1)}(\mathit{on}) \,|\, K_{(0,1)}(0) \,|\, S_{(1,0)}(\mathit{on}) \,|\, Q'_1 \,|\, T_0 \,|\, T_1) \\
& \qquad \xrightarrow{eat_1} (\nu L)(P''_0 \,|\, S_{(0,1)}(\mathit{on}) \,|\, K_{(0,1)}(0) \,|\, S_{(1,0)}(\mathit{on}) \,|\, P'''_1 \,|\, T_0 \,|\, T_1) \\
& \qquad \xrightarrow{\tau} (\nu L)(P''_0 \,|\, S_{(0,1)}(\mathit{on}) \,|\, K_{(0,1)}(1) \,|\, S_{(1,0)}(\mathit{off}) \,|\, P_1 \,|\, F_0 \,|\, F_1)
\end{aligned}$$

and the reached state will allow P''_0 to get the forks, even in case P_1 will desire to eat, because variable $K_{(0,1)}$ records that P''_1 was served last.