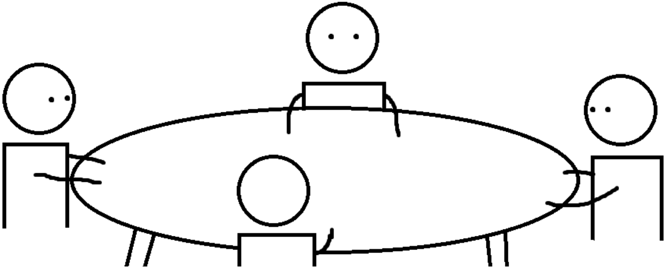
Avoiding deadlocks in lock-sharing systems

Corto Mascle

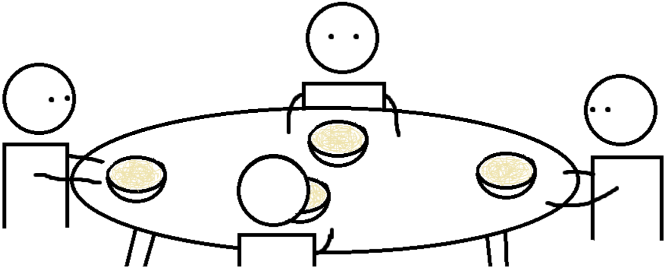
Joint work with Hugo Gimbert, Anca Muscholl and Igor Walukiewicz

April 12th 2022

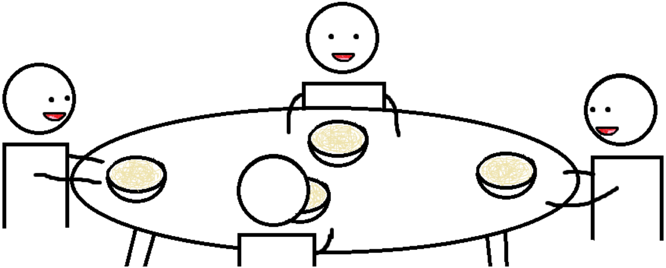
Dining philosophers



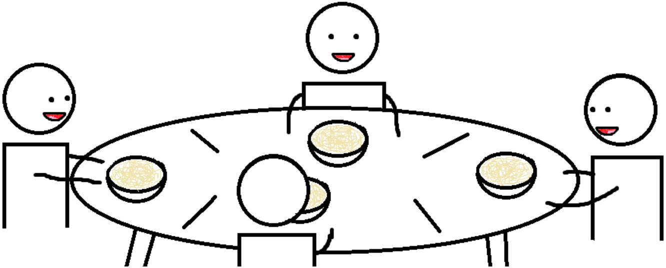
Dining philosophers



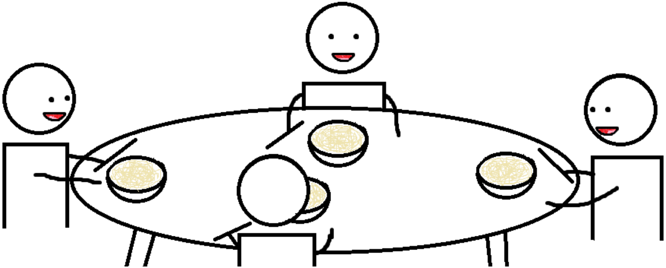
Dining philosophers



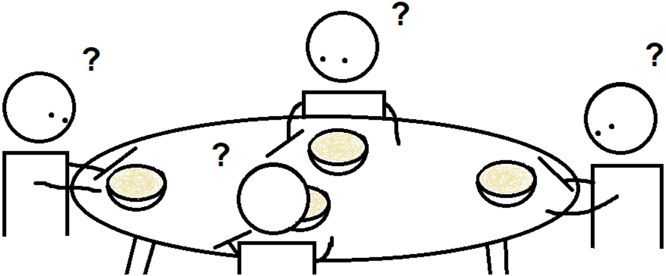
Dining philosophers



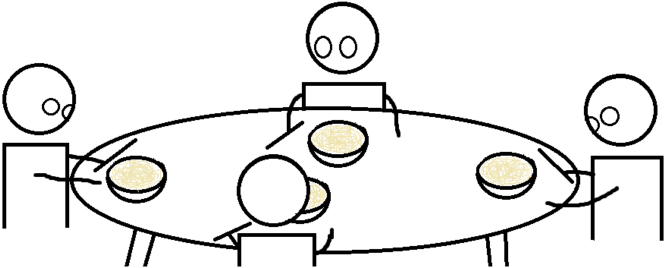
Dining philosophers



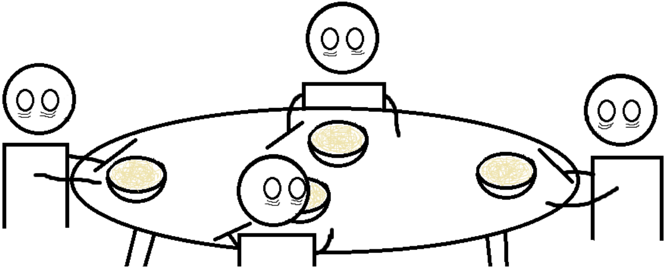
Dining philosophers



Dining philosophers



Dining philosophers



Which model?

Many models exist for distributed synthesis:

- $\rightarrow$ Zielonka automata
- $\rightarrow$ Petri nets
- $\rightarrow$ Processes with shared variables

Which model?

Many models exist for distributed synthesis:

- $\rightarrow$ Zielonka automata
- $\rightarrow$ Petri nets
- $\rightarrow$ Processes with shared variables

We would like a simple model that allows very little communication between processes.

Lock-sharing systems

Lock-sharing system

Let $Proc$ be a set of processes and T a set of locks.
A lock-sharing system (LSS) is a family of finite transition systems $\mathcal{A}_P = (S_P, \Sigma_P, \delta_P, init_P)$, one for each process P.

Transitions include operations on tokens :
$\delta_P : S_P \times \Sigma_P \to Op_T \times S_P$ with $Op_T = \{acq_t, rel_t \mid t \in T\} \cup \{nop\}$.

Lock-sharing systems

Lock-sharing system

Let $Proc$ be a set of processes and T a set of locks.
A lock-sharing system (LSS) is a family of finite transition systems $\mathcal{A}_P = (S_P, \Sigma_P, \delta_P, init_P)$, one for each process P.

Transitions include operations on tokens :
$\delta_P : S_P \times \Sigma_P \to Op_T \times S_P$ with $Op_T = \{acq_t, rel_t \mid t \in T\} \cup \{nop\}$.

Intuitive semantics: a process holds some of the locks and cannot acquire a lock that is not free or release a lock it does not hold.

Lock-sharing systems

Lock-sharing system

Let $Proc$ be a set of processes and T a set of locks.
A lock-sharing system (LSS) is a family of finite transition systems $\mathcal{A}_P = (S_P, \Sigma_P, \delta_P, init_P)$, one for each process P.

Transitions include operations on tokens :
$\delta_P : S_P \times \Sigma_P \to Op_T \times S_P$ with $Op_T = \{acq_t, rel_t \mid t \in T\} \cup \{nop\}$.

Intuitive semantics: a process holds some of the locks and cannot acquire a lock that is not free or release a lock it does not hold.

We split each set of actions into *controllable* and *uncontrollable* ones $\Sigma_P = \Sigma_P^c \sqcup \Sigma_P^u$.

Strategies

Strategy

A control strategy is a family $(\sigma_P)_{P \in Proc}$ of local strategies, with $\sigma_P : \Sigma_P^* \to 2^{\Sigma_P}$ such that $\Sigma_P^u \subseteq \sigma_P(u)$ for all u.
A local σ-run u_P of P is such that for all prefix va of u_P, $a \in \sigma_P(v)$.
A σ-run is a run whose projection on any process P is a local σ-run.

Strategies

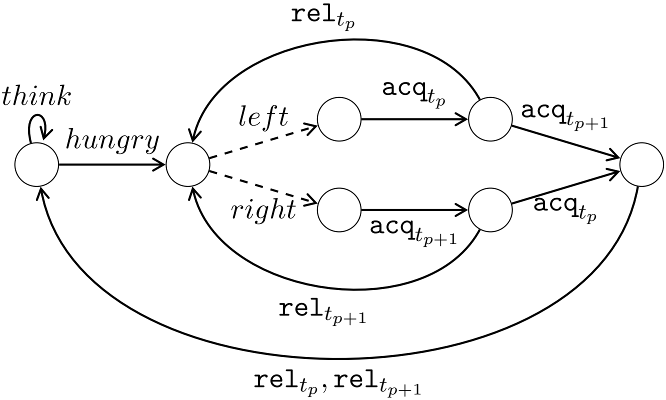
Strategy

A control strategy is a family $(\sigma_P)_{P \in Proc}$ of local strategies, with $\sigma_P : \Sigma_P^* \to 2^{\Sigma_P}$ such that $\Sigma_P^u \subseteq \sigma_P(u)$ for all u.
A local σ-run u_P of P is such that for all prefix va of u_P, $a \in \sigma_P(v)$.
A σ-run is a run whose projection on any process P is a local σ-run.

Deadlock

A σ-run u reaches a deadlock if it cannot be extended into a longer σ-run ua.

Example



The strategy σ such that σ_P always selects $left$ and σ_Q $right$ for some processes P, Q avoids deadlocks.

The problem

Deadlock avoidance problem

Input: A set of processes $Proc$, a set of tokens T and an LSS $(\mathcal{A}_P)_{P \in Proc}$
Output: Does there exist a strategy σ such that no σ-run reaches a deadlock?

The same problem can be formulated with *partial deadlocks*, in which we give a subset of processes and look for a strategy that avoids blocking those.

Undecidability

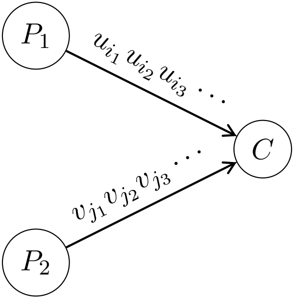
Theorem

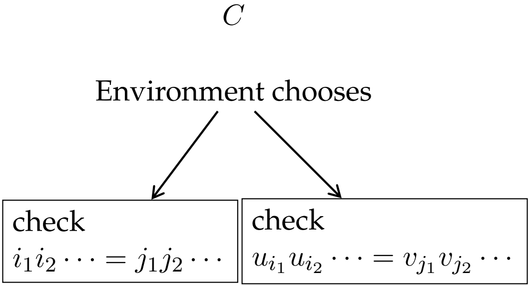
The deadlock avoidance problem is undecidable, even with 3 processes and 4 tokens in total.

$\rightarrow$ Processes can share information by interleaving lock acquisitions!

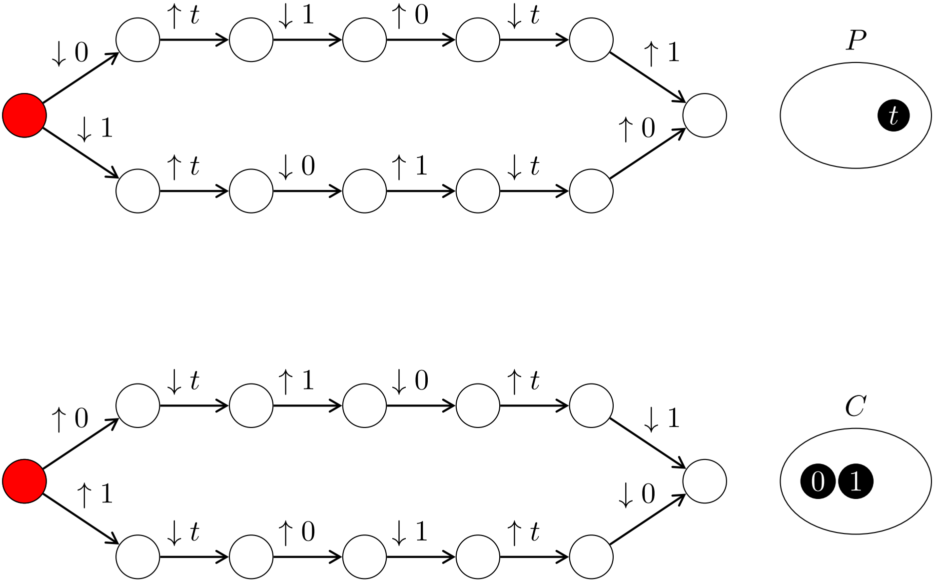
Proof scheme: PCP encoding

Let $(u_1, v_1), \dots, (u_n, v_n)$ be a PCP instance.

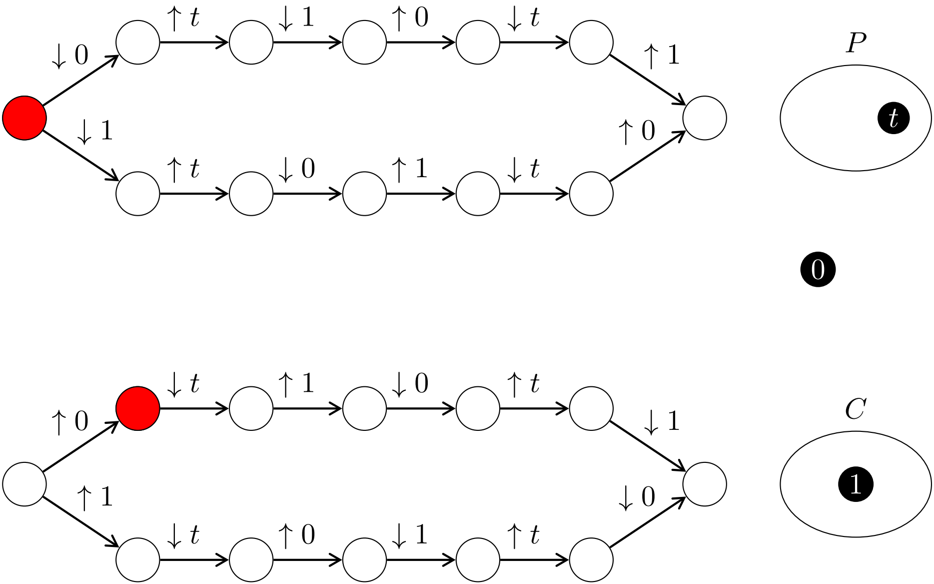




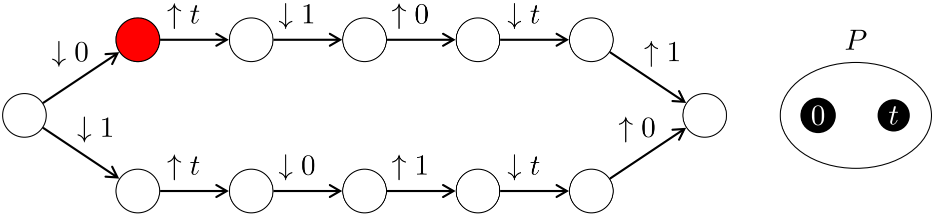
Proof scheme: Passing information

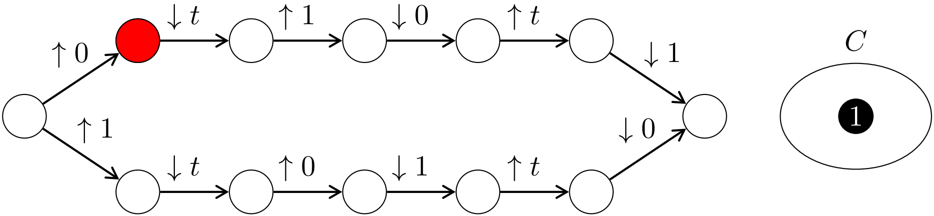


Proof scheme: Passing information

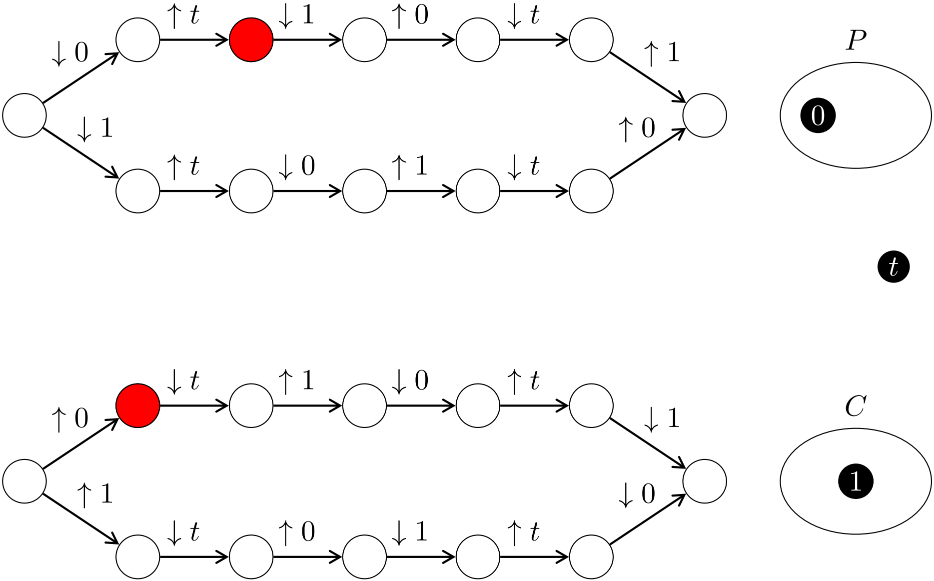


Proof scheme: Passing information

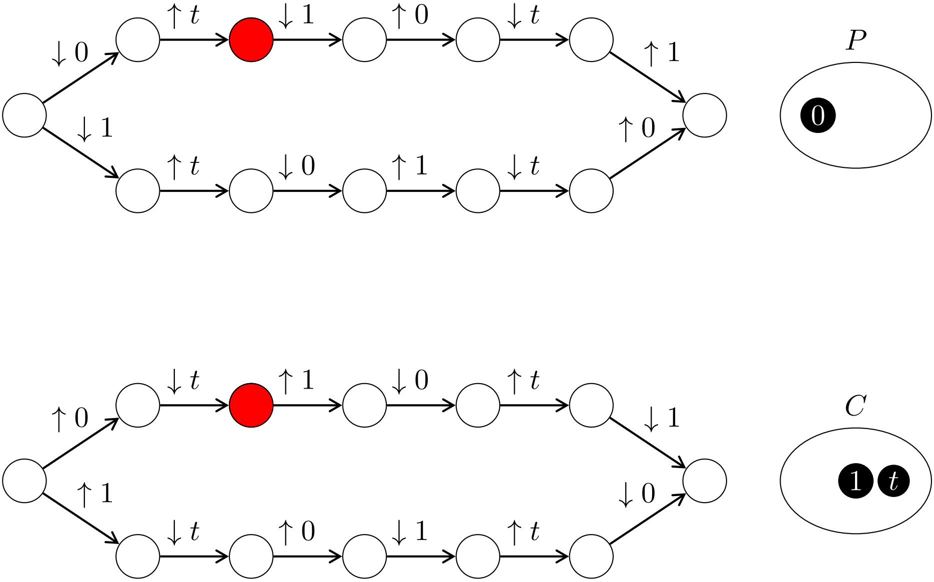




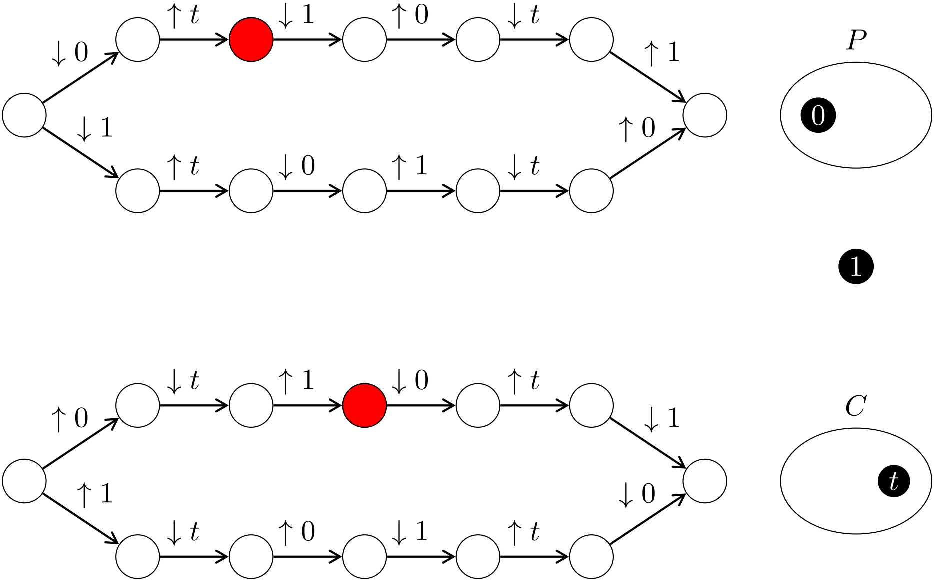
Proof scheme: Passing information



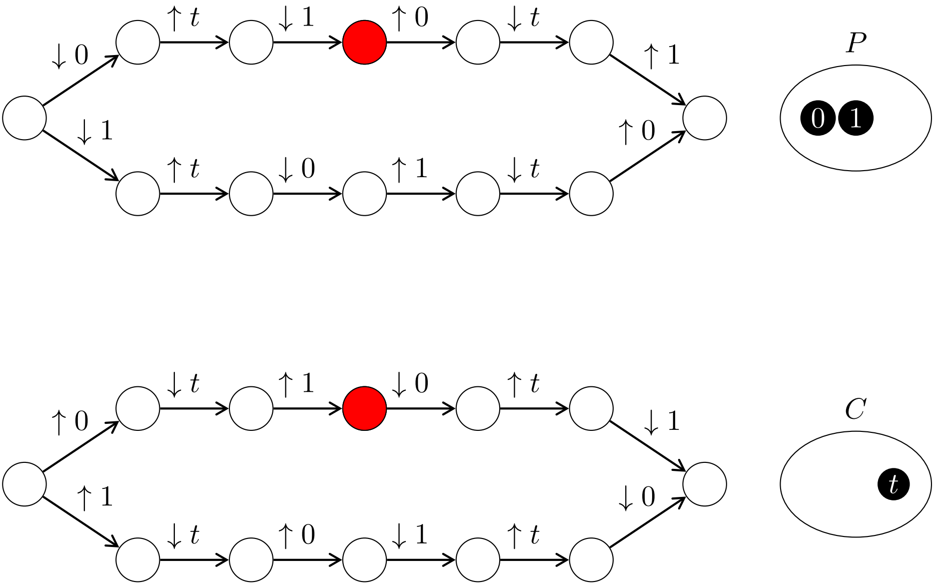
Proof scheme: Passing information



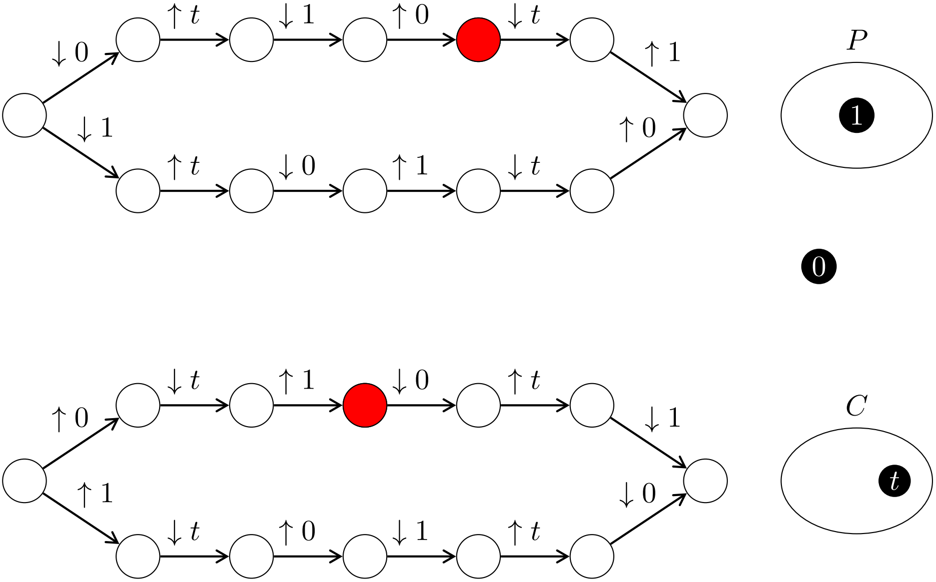
Proof scheme: Passing information



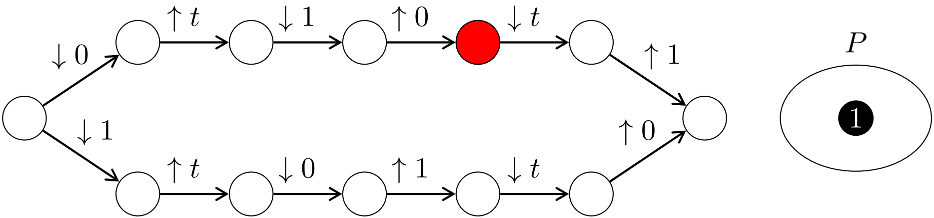
Proof scheme: Passing information

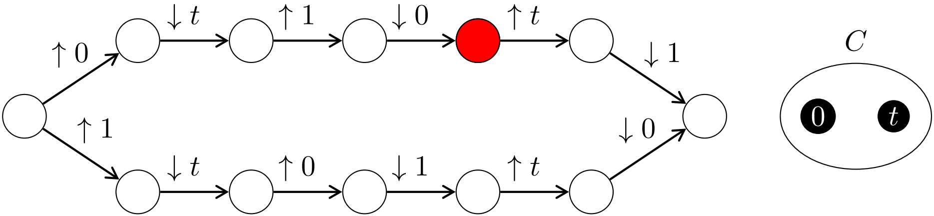


Proof scheme: Passing information

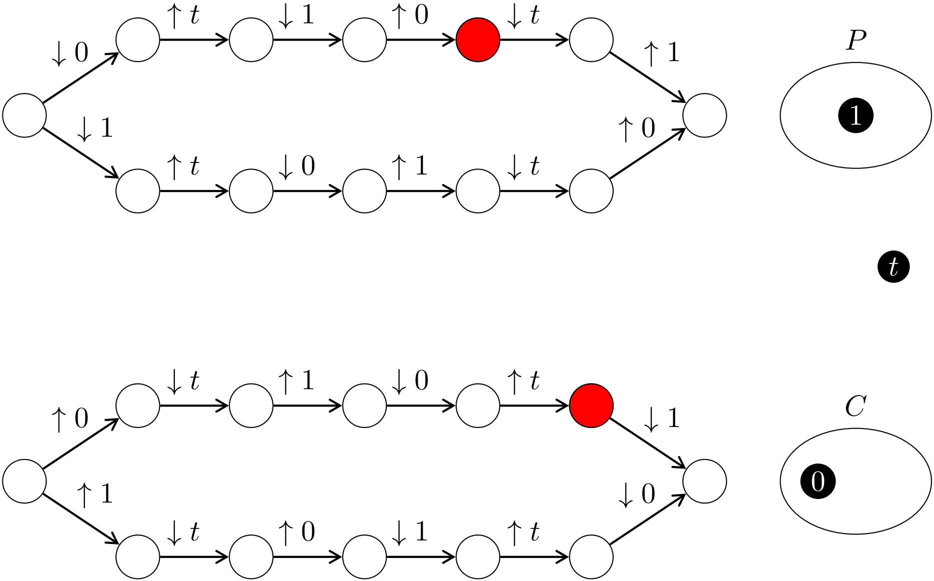


Proof scheme: Passing information

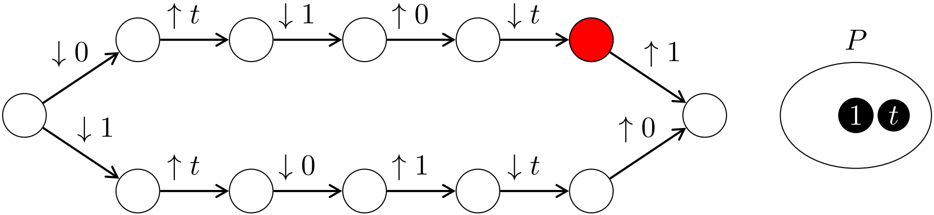


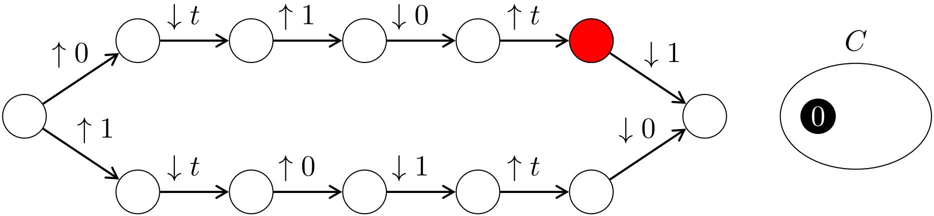


Proof scheme: Passing information

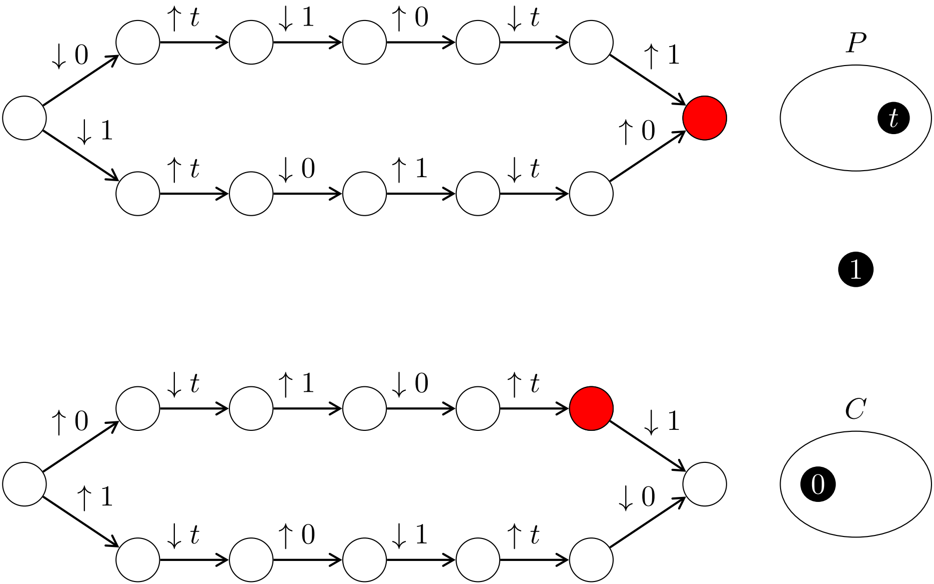


Proof scheme: Passing information

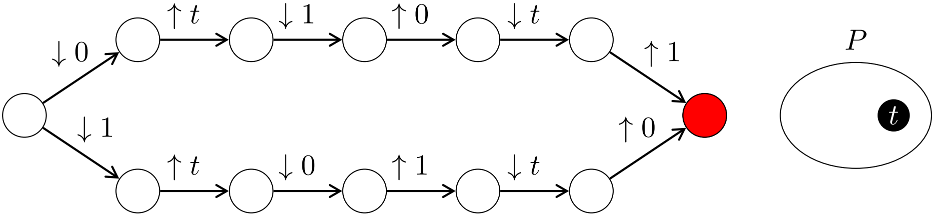


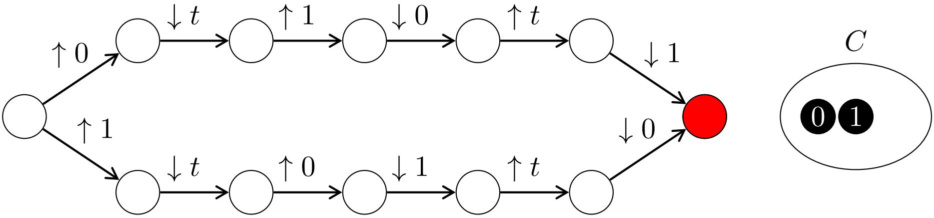


Proof scheme: Passing information



Proof scheme: Passing information





Two locks per process

2LSS

A 2LSS is an LSS in which the transitions of each process contain operations on at most two different locks.

This prevents processes from communicating an unbounded amount of information.

Patterns

A local run u_P of a process P in a 2LSS satisfies one of the following things:

- $\rightarrow$ P holds no lock at the end
- $\rightarrow$ P holds both of its locks at the end
- $\rightarrow$ P holds only t_1 at the end, and its last operation on locks is acq_{t_1}
- $\rightarrow$ P holds only t_1 at the end, and its last operation on locks is rel_{t_2}

The key property of 2LSS is that we can abstract local runs into those patterns to solve the problem.

Scheduling

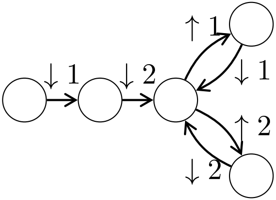
Given patterns $(pat_P)_{P \in Proc}$, we want to check that local runs with those patterns can be scheduled into a global run.

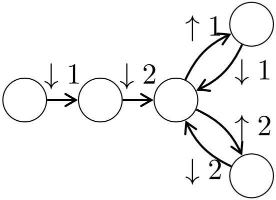
We check that a lock is not held by two different processes at the end of their local run, but this is not sufficient.

Scheduling

Given patterns $(pat_P)_{P \in Proc}$, we want to check that local runs with those patterns can be scheduled into a global run.

We check that a lock is not held by two different processes at the end of their local run, but this is not sufficient.

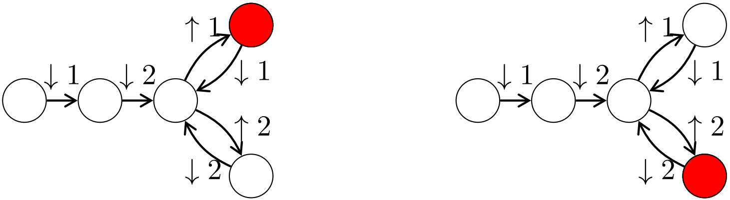




Scheduling

Given patterns $(pat_P)_{P \in Proc}$, we want to check that local runs with those patterns can be scheduled into a global run.

We check that a lock is not held by two different processes at the end of their local run, but this is not sufficient.



This is a deadlock but not a reachable one!

Scheduling

We split each local run into two parts, cutting at the last position at which the process holds no lock $u_p = v_p w_p$.
All v_p can be executed sequentially at the beginning of the run.

- $\rightarrow$ P holds no lock at the end of u_P

- $\rightarrow$ P holds both of its locks at the end

- $\rightarrow$ P holds only t_1 at the end, and its last operation on locks is acq_{t_1}

- $\rightarrow$ P holds only t_1 at the end, and its last operation on locks is rel_{t_2}

Scheduling

We split each local run into two parts, cutting at the last position at which the process holds no lock $u_p = v_p w_p$.
All v_p can be executed sequentially at the beginning of the run.

- $\rightarrow$ P holds no lock at the end of u_P
- $\rightarrow$ *Then w_P is empty.*
- $\rightarrow$ P holds both of its locks at the end

- $\rightarrow$ P holds only t_1 at the end, and its last operation on locks is acq_{t_1}

- $\rightarrow$ P holds only t_1 at the end, and its last operation on locks is rel_{t_2}

Scheduling

We split each local run into two parts, cutting at the last position at which the process holds no lock $u_p = v_p w_p$.
All v_p can be executed sequentially at the beginning of the run.

- $\rightarrow$ P holds no lock at the end of u_P
- $\rightarrow$ *Then w_P is empty.*
- $\rightarrow$ P holds both of its locks at the end
- $\rightarrow$ *We can run w_P **after** all the other runs.*
- $\rightarrow$ P holds only t_1 at the end, and its last operation on locks is acq_{t_1}

- $\rightarrow$ P holds only t_1 at the end, and its last operation on locks is rel_{t_2}

Scheduling

We split each local run into two parts, cutting at the last position at which the process holds no lock $u_p = v_p w_p$.
All v_p can be executed sequentially at the beginning of the run.

- $\rightarrow$ P holds no lock at the end of u_P
- $\rightarrow$ *Then w_P is empty.*
- $\rightarrow$ P holds both of its locks at the end
- $\rightarrow$ *We can run w_P **after** all the other runs.*
- $\rightarrow$ P holds only t_1 at the end, and its last operation on locks is acq_{t_1}
- $\rightarrow$ *We can run w_P **after** all the other runs as well.*
- $\rightarrow$ P holds only t_1 at the end, and its last operation on locks is rel_{t_2}

Scheduling

We split each local run into two parts, cutting at the last position at which the process holds no lock $u_p = v_p w_p$.
All v_p can be executed sequentially at the beginning of the run.

- $\rightarrow$ P holds no lock at the end of u_P
- $\rightarrow$ *Then w_P is empty.*
- $\rightarrow$ P holds both of its locks at the end
- $\rightarrow$ *We can run w_P **after** all the other runs.*
- $\rightarrow$ P holds only t_1 at the end, and its last operation on locks is acq_{t_1}
- $\rightarrow$ *We can run w_P **after** all the other runs as well.*
- $\rightarrow$ P holds only t_1 at the end, and its last operation on locks is rel_{t_2}
- $\rightarrow$ *We need to be more careful.*

Scheduling

Say a local run u_P has P take locks t_1, t_2, then release t_2 (but never t_1).

In a global run u, this means that the last operation on t_1 in u is before the last operation on t_2.

A local run u_Q which takes t_1 and t_2, then releases t_1 cannot be interleaved with u_P to form a global run.

Lemma

A family of local runs $(u_P)_{P \in Proc}$ can be scheduled into a global run iff

- No token is held by two different processes at the end of their local run.
- There is a total order on tokens compatible with the patterns of all u_P.

Abstracting strategies

Abstract strategies

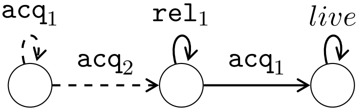
An abstract local strategy for process P is a set of pairs $(pat_P, Block_P)$.

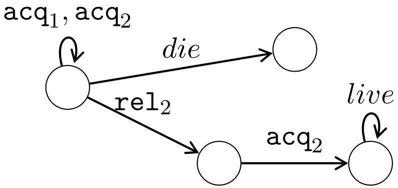
To a local strategy σ_P we associate an abstract one Abs_P such that

$$(pat_P, Block_P) \in Abs_P \\ \Leftrightarrow \\ \exists u_P, \delta_P(\sigma_P(u_P)) \subseteq \{\mathtt{acq}_t \mid t \in Block_P\} \times S_P$$

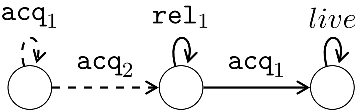
There is a local run respecting σ with pattern pat_P ending in a configuration in which all actions available are acquiring a lock from $Block_P$.

Example

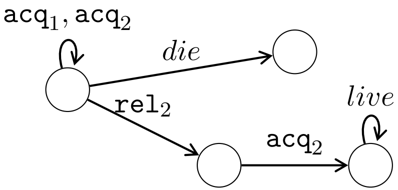




Example



A good strategy here is to first allow the $\mathtt{acq}_1$ loop, then the $\mathtt{acq}_2$ transition.



For the first process this strategy yields the abstract runs $(\varepsilon, \{1\}), (\mathtt{acq}_1, \{2\}), (\mathtt{acq}_{1,2}\mathtt{rel}_1, \{1\})$.

For the second one it yields $(\varepsilon, \{1, 2\}), (\varepsilon, \{2\}), (\mathtt{acq}_{1,2}\mathtt{rel}_2, \{2\})$.

A Σ_2^p algorithm...

We translate the problem into a one-turn two-player game:

A Σ_2^p algorithm...

We translate the problem into a one-turn two-player game:

Step 1: Sys gives for each process P an abstract strategy Abs_P.

A Σ_2^p algorithm...

We translate the problem into a one-turn two-player game:

Step 1: Sys gives for each process P an abstract strategy Abs_P.

Step 2: We check (in PTIME) that there exist local strategies yielding those abstract ones.

A Σ_2^p algorithm...

We translate the problem into a one-turn two-player game:

Step 1: Sys gives for each process P an abstract strategy Abs_P.

Step 2: We check (in PTIME) that there exist local strategies yielding those abstract ones.

Step 3: Env chooses in each Abs_P a pair $(pat_P, Block_P)$.

A Σ_2^p algorithm...

We translate the problem into a one-turn two-player game:

Step 1: Sys gives for each process P an abstract strategy Abs_P.

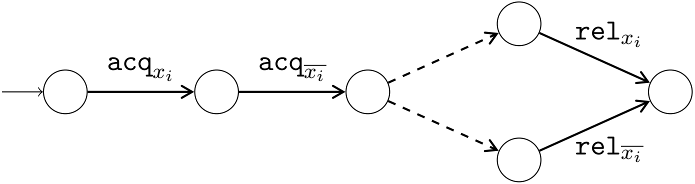
Step 2: We check (in PTIME) that there exist local strategies yielding those abstract ones.

Step 3: Env chooses in each Abs_P a pair $(pat_P, Block_P)$.

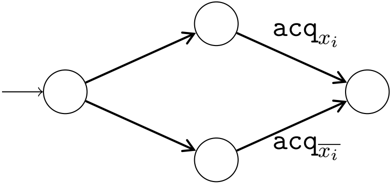
Step 4: We check (in PTIME) that the $(pat_P, Block_P)_{P \in Proc}$ witness a deadlock: the pat_P represent local runs that can be scheduled into a global one, and none of the locks in the $Block_P$ are free at the end.

...and a lower bound

We reduce the ∃∀SAT problem. We encode valuations as sets of non-free tokens.
Sys chooses a value for some variables:



And Env chooses the values of the other ones.



Liveness hypothesis

Locally live strategy

A strategy is *locally live* if all local σ-run u_P can be extended into a longer local σ-run $u_p a$.

Thus the deadlocks can only arise from a bad token distribution.

In terms of abstract strategies, it means that the $Block_P$ component in the abstract runs cannot be empty.

Thus a process that holds a lock in a deadlock must be blocked by another one.

Lock graph and deadlock schemes

Let σ be a strategy. Abstract strategies can be represented as a *lock graph* $G_\sigma = (T, E)$ with locks as vertices.

Lock graph

The lock graph $G_\sigma = (T, E)$ is such that for all $P \in Proc$ of abstract strategy Abs_P:

- if $(\mathtt{acq}_{t_1}, \{t_2\}) \in Abs_P$ then there is a *weak* edge $t_1 \xrightarrow{P} t_2$.
- if $(\mathtt{acq}_{t_1,t_2}\mathtt{rel}_{t_2}, \{t_2\}) \in Abs_P$ and $(\mathtt{acq}_{t_1}, \{t_2\}) \notin Abs_P$ then there is a *strong* edge $t_1 \stackrel{P}{\Longrightarrow} t_2$.

Lock graph and deadlock schemes

We also separate *solid* and *fragile* processes.

Lock graph

A process P is fragile if Abs_P contains $(\varepsilon, Blocks_P)$ for some $Blocks_P$, i.e., if P can be blocked without holding any lock.
P is solid otherwise.

Lock graph and deadlock schemes

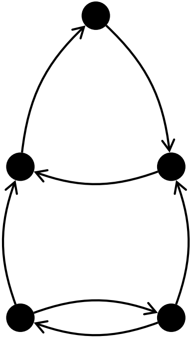
Let σ be a strategy. A deadlock situation can be represented by a *deadlock scheme*.

Deadlock scheme

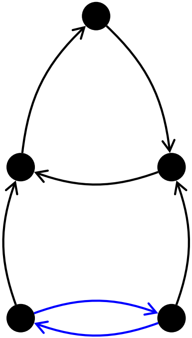
A deadlock scheme is a pair (BT, ds) with $BT \subseteq T$ and $ds : Proc \to E \cup \{\perp\}$ such that:

- If $ds(P) = \perp$ then $ds(P)$ is fragile and Abs_P contains some $(\varepsilon, Blocks_P)$ with $Blocks_P \subseteq BT$.
- If $ds(P)$ is an edge then it is labelled by P and within BT.
- Every lock in BT has exactly one outgoing edge.
- There are no cycles of strong edges in $ds(P)$.

Example: solid v. fragile processes

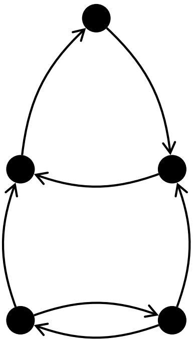


There is no deadlock scheme for this lock graph if every process is solid.

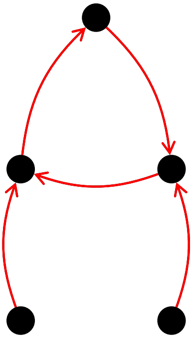


But if the process in blue is fragile, then there is one!

Example: solid v. fragile processes

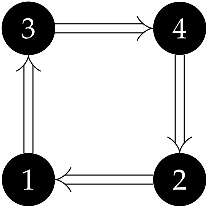


There is no deadlock scheme for this lock graph if every process is solid.

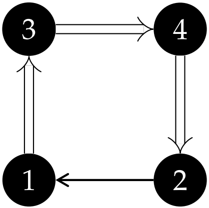


But if the process in blue is fragile, then there is one!

Example: weak v. strong edges



A deadlock scheme cannot have a cycle of strong edges as they represent incompatible runs.



A cycle with a weak edge always represents a set of runs that can be interleaved.

Exclusive case

Exclusive

An LSS is *exclusive* if whenever a process can acquire a lock, its only available actions are acquiring locks.

This means that every $\mathtt{acq}_t$ operation is an opportunity for a deadlock.

Exclusive case

Exclusive

An LSS is *exclusive* if whenever a process can acquire a lock, its only available actions are acquiring locks.

This means that every $\mathtt{acq}_t$ operation is an opportunity for a deadlock.

It further restricts the abstract runs:
An abstract run taking both tokens, releasing 2 and blocking on 2...

$$\xrightarrow{\qquad} \; \xrightarrow{\mathtt{acq}_1} \; \xrightarrow{\mathtt{acq}_2} \; \xrightarrow{\mathtt{rel}_2} \; \xrightarrow{\mathtt{acq}_2}$$

Exclusive case

Exclusive

An LSS is *exclusive* if whenever a process can acquire a lock, its only available actions are acquiring locks.

This means that every $\mathtt{acq}_t$ operation is an opportunity for a deadlock.

It further restricts the abstract runs:
An abstract run taking both tokens, releasing 2 and blocking on 2...
... implies another one taking one token and blocking on the other...

$$\xrightarrow{\qquad} \xrightarrow{\mathtt{acq}_1} \xrightarrow{\mathtt{acq}_2} \xrightarrow{\mathtt{rel}_2} \xrightarrow{\mathtt{acq}_2}$$

Exclusive case

Exclusive

An LSS is *exclusive* if whenever a process can acquire a lock, its only available actions are acquiring locks.

This means that every $\mathtt{acq}_t$ operation is an opportunity for a deadlock.

It further restricts the abstract runs:
An abstract run taking both tokens, releasing 2 and blocking on 2...
... implies another one taking one token and blocking on the other...
... in turn implying one blocking without taking any token.

$$\xrightarrow{\qquad} \xrightarrow{\mathtt{acq}_1} \xrightarrow{\mathtt{acq}_2} \xrightarrow{\mathtt{rel}_2} \xrightarrow{\mathtt{acq}_2}$$

Two key lemmas

Let σ be a locally live strategy for an exclusive LSS.

Lemma

If there is a strong edge $t_1 \to t_2$ in G_σ then there is also a weak one $t_2 \to t_1$.

In particular, every strong cycle can be replaced by a weak one.

Two key lemmas

Let σ be a locally live strategy for an exclusive LSS.

Lemma

If there is a strong edge $t_1 \to t_2$ in G_σ then there is also a weak one $t_2 \to t_1$.

In particular, every strong cycle can be replaced by a weak one.

Lemma

All processes are fragile with respect to σ.

This allows us to "erase" edges, thus the constraint that all locks have **exactly one** outgoing edge becomes **at least one**.

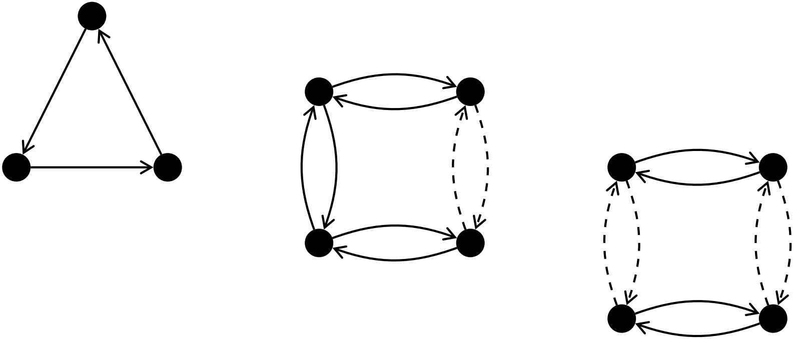
Finding cycles

The problem essentially comes down to a game in which:

- → For each process P using tokens t_1, t_2 Sys chooses a set of edges between t_1 and t_2 (one that can be obtained by a local strategy).
- → Env chooses one of those edges (if the set is non-empty), and wins if all tokens with an incoming edge also have an outgoing one.

Finding cycles

Full edge $\rightarrow$ will appear no matter the local strategy.
Double dashed edge $\rightarrow$ an edge will appear but Sys can choose its orientation.



A best strategy for Sys is to orient edges according to an order on strongly connected components of the graph of unavoidable edges.

Exclusive case

Theorem

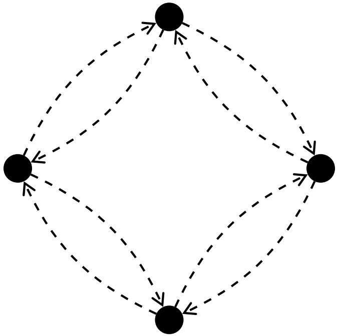
The deadlock avoidance control problem is in PTIME for exclusive LSS with locally live strategies.

$\Rightarrow$ In quadratic time in the number of states per process and linear time in the number of processes.

Theorem

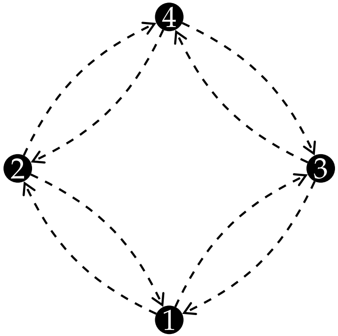
The deadlock avoidance control problem is Σ_2^P-complete for exclusive LSS with general strategies.

Solving the dining philosophers



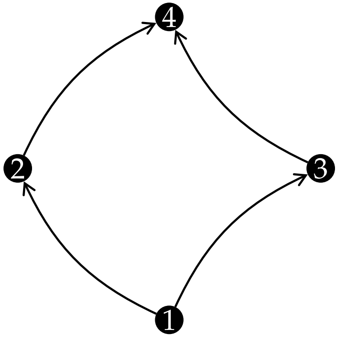
→ Just pick an order on chopsticks and have all philosophers take them accordingly!

Solving the dining philosophers



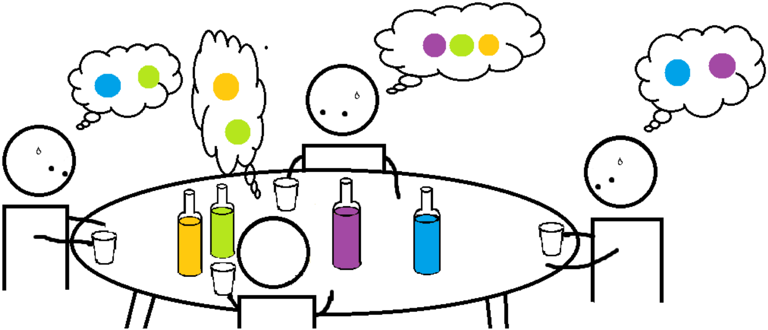
→ Just pick an order on chopsticks and have all philosophers take them accordingly!

Solving the dining philosophers



→ Just pick an order on chopsticks and have all philosophers take them accordingly!

Drinking philosophers



Nested locks

Nested locks condition

An LSS satisfies the nested lock condition if all processes acquire and release locks in a stack-like order, i.e., a process can only release the last lock it acquired.

Nested locks

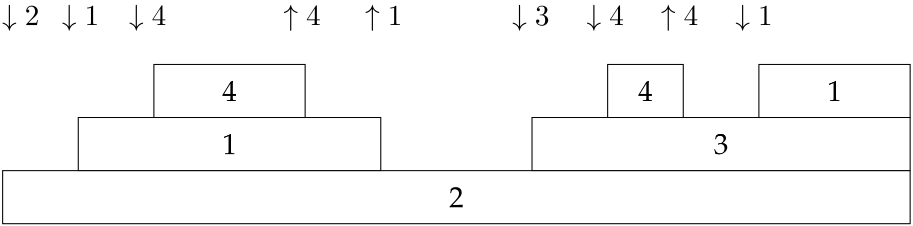
Nested locks condition

An LSS satisfies the nested lock condition if all processes acquire and release locks in a stack-like order, i.e., a process can only release the last lock it acquired.

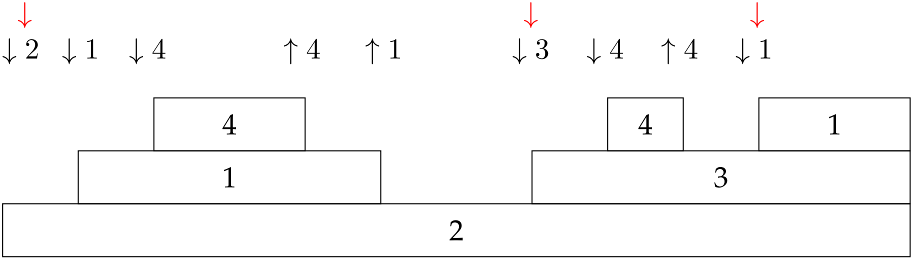
Theorem

The deadlock avoidance control problem is NEXPTIME-complete for LSS respecting the nested lock condition.

Stair decomposition

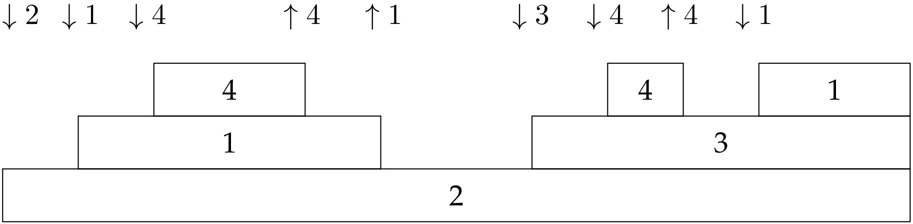


Stair decomposition



We distinguish the `acq` operations that take a lock that is never released

Stair decomposition



We distinguish the `acq` operations that take a lock that is never released as well as the last `rel` operations on the other locks.

The pattern of the run is the word of these ordered operations: $\downarrow 2 \downarrow 3 \uparrow 4 \downarrow 1$.

Scheduling = Finding a common order

Theorem

A family of runs $(u_P)_{P \in Proc}$ can be scheduled into a global run if and only if their patterns can be shuffled into a word w such that:

- No $\texttt{acq}_t$ operation appears more than once.
- For all t, all $\texttt{rel}_t$ operations are before the $\texttt{acq}_t$ one (if it exists).

Like for 2LSS, an abstract run for P is a pair $(pattern, Blocks)$ and an abstract strategy for P is a set of such abstract runs.

Scheduling = Finding a common order

Lemma

Given an abstract strategy for process P, we can check in polynomial time in the size of that abstract strategy (and exponential in the size of the LSS) whether it is the abstraction of a local strategy.

From there we proceed as in the 2LSS case:

→ Sys chooses a set of abstract runs for each process (NEXPTIME).

Scheduling = Finding a common order

Lemma

Given an abstract strategy for process P, we can check in polynomial time in the size of that abstract strategy (and exponential in the size of the LSS) whether it is the abstraction of a local strategy.

From there we proceed as in the 2LSS case:

- $\rightarrow$ Sys chooses a set of abstract runs for each process (NExpTime).
- $\rightarrow$ We check that those can be achieved by local strategies (ExpTime).

Scheduling = Finding a common order

Lemma

Given an abstract strategy for process P, we can check in polynomial time in the size of that abstract strategy (and exponential in the size of the LSS) whether it is the abstraction of a local strategy.

From there we proceed as in the 2LSS case:

- $\rightarrow$ Sys chooses a set of abstract runs for each process (NExpTime).
- $\rightarrow$ We check that those can be achieved by local strategies (ExpTime).
- $\rightarrow$ Env chooses an abstract run in each set (ExpTime).
- $\rightarrow$ We check that those represent local runs that can be scheduled into a global one leading to a deadlock (PTime).

NExpTime-hardness idea

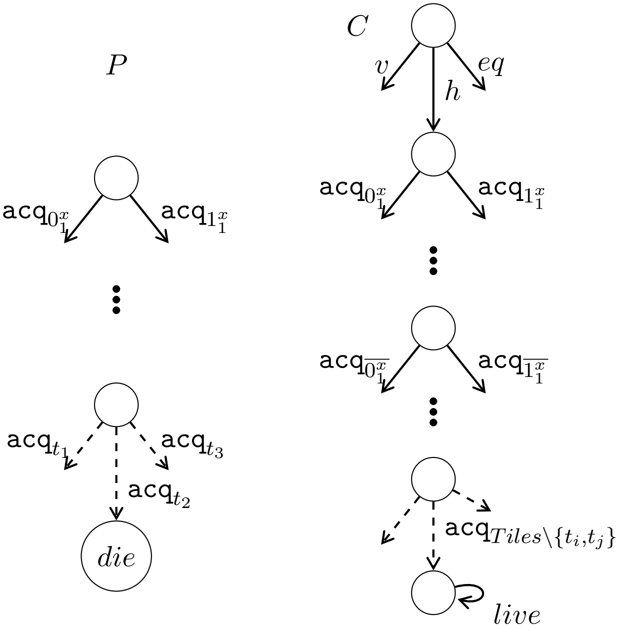
We reduce the problem of tiling a $N \times N$ (N in binary) square with a given set of coloured tiles $Tiles \subseteq Colours^{\{up, down, left, right\}}$.

The colours have to match between all pairs of adjacent tiles.

We use two sets of locks $0_i^x, 1_i^x, 0_i^y, 1_i^y$ and $\overline{0_i^x}, \overline{1_i^x}, \overline{0_i^y}, \overline{1_i^y}$ to encode coordinates in binary.

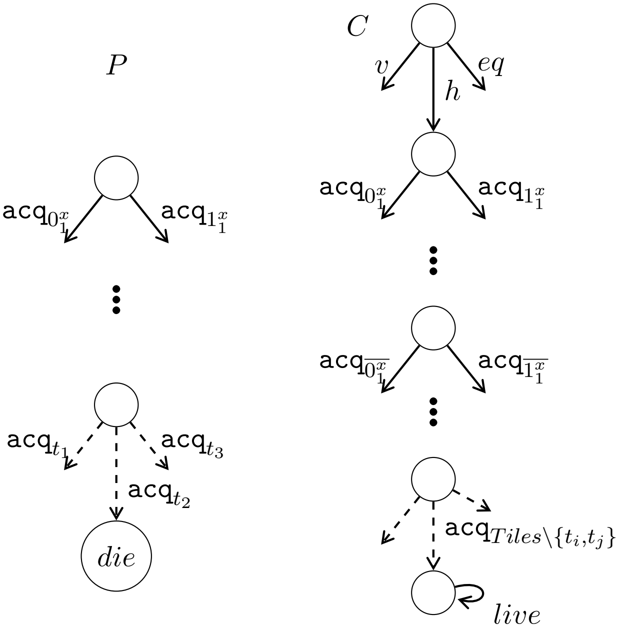
We also use one lock t for each tile $t \in Tiles$.

NEXPTIME-hardness idea



→ Env chooses to check a vertical, horizontal or equality condition.

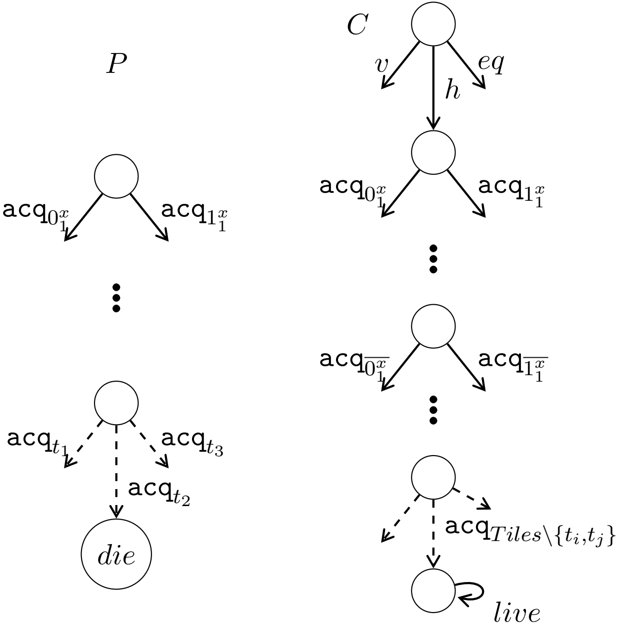
NEXPTIME-hardness idea



→ Env chooses to check a vertical, horizontal or equality condition.

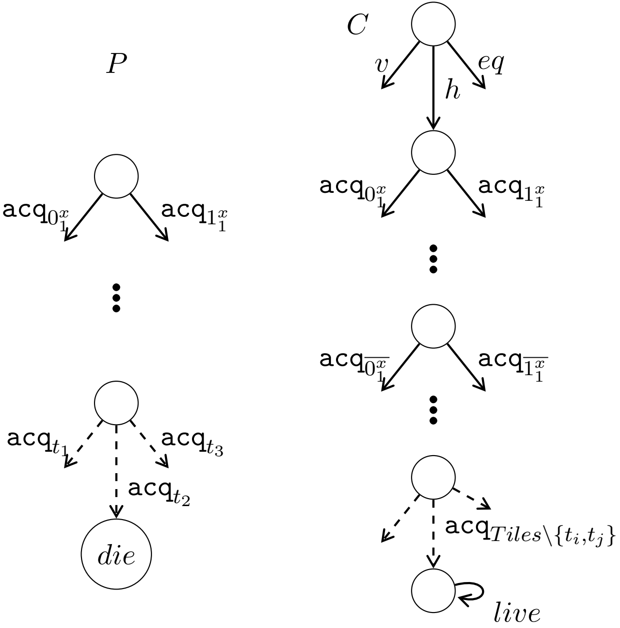
→ It takes locks encoding the coordinates of two adjacent tiles.

NEXPTIME-hardness idea



- → Env chooses to check a vertical, horizontal or equality condition.
- → It takes locks encoding the coordinates of two adjacent tiles.
- → P and $\overline{P}$ get the remaining bits.

NExpTime-hardness idea



- → Env chooses to check a vertical, horizontal or equality condition.
- → It takes locks encoding the coordinates of two adjacent tiles.
- → P and $\overline{P}$ get the remaining bits.
- → They each acquire a tile, while C chooses two tiles to leave and takes all the others.

Practical prospects

Examples of processes with two mutex can be found

→ In the BSD kernel

→ In the C++ documentation

Implementation of the PTIME algorithms.

Open problems

- $\rightarrow$ Decidability for two processes or three locks/process
- $\rightarrow$ PTIME algorithm for 2LSS with locally live strategies
- $\rightarrow$ Other restrictions on lock usage to prevent passing information

Thank you for your attention!