Verification and synthesis of dynamic systems with locks and variables

Corto Mascle
joint work with Anca Muscholl, Igor Walukiewicz

LaBRI, Bordeaux

Verification and *synthesis* of dynamic systems with locks and variables

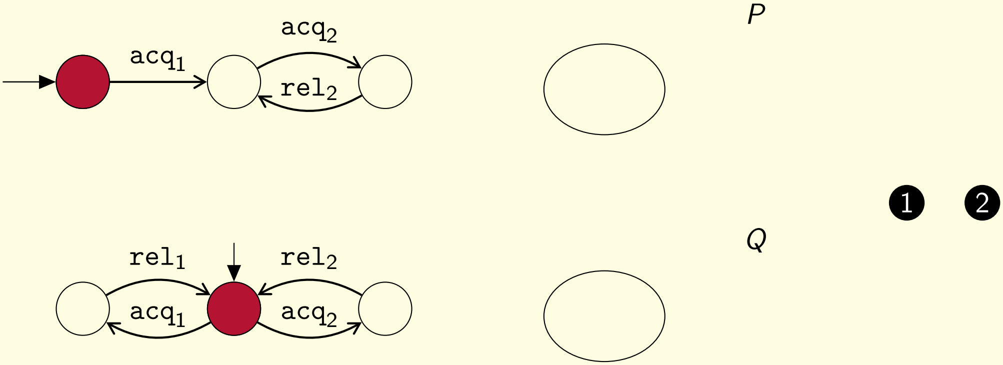
Corto Mascle
joint work with Anca Muscholl, Igor Walukiewicz

LaBRI, Bordeaux

Part I

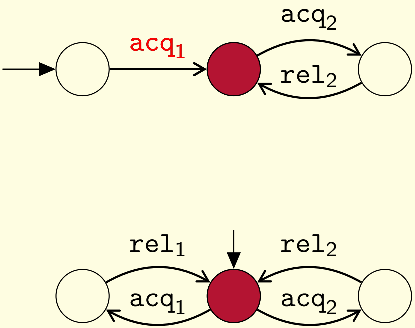
Verification

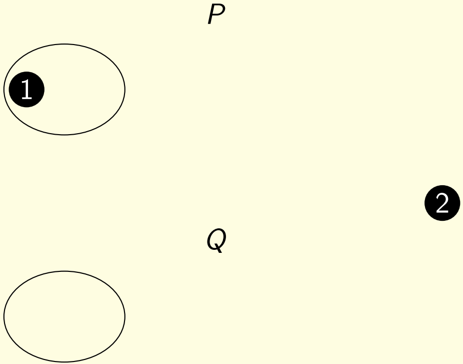
Lock-sharing systems[1]



[1]Kahlon, Ivancic, Gupta CAV 2005

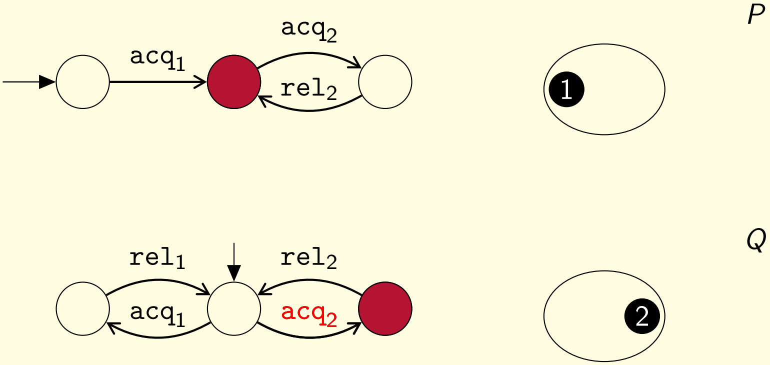
Lock-sharing systems[1]





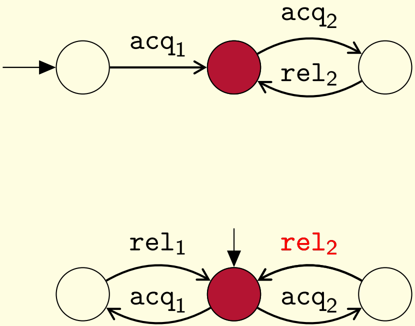
[1]Kahlon, Ivancic, Gupta CAV 2005

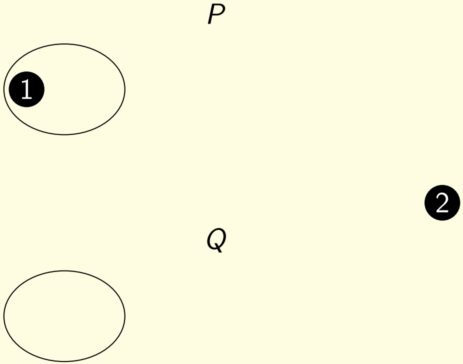
Lock-sharing systems[1]



[1]Kahlon, Ivancic, Gupta CAV 2005

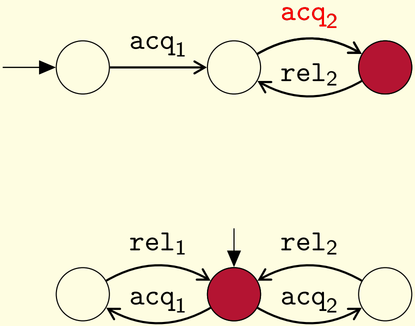
Lock-sharing systems[1]

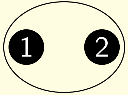




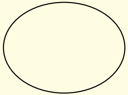
[1]Kahlon, Ivancic, Gupta CAV 2005

Lock-sharing systems[1]





P

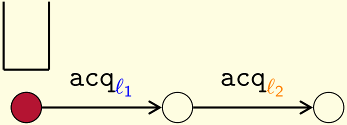


Q

[1]Kahlon, Ivancic, Gupta CAV 2005

Restriction: Nested locking

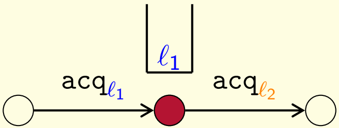
All processes acquire and release locks in a **stack-like order**, i.e., a process can only release the lock it acquired the latest.



This restricts communication between processes.

Restriction: Nested locking

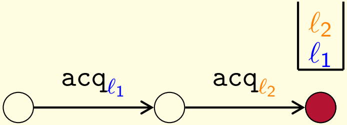
All processes acquire and release locks in a **stack-like order**, i.e., a process can only release the lock it acquired the latest.



This restricts communication between processes.

Restriction: Nested locking

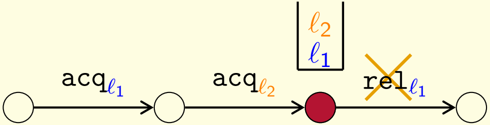
All processes acquire and release locks in a **stack-like order**, i.e., a process can only release the lock it acquired the latest.



This restricts communication between processes.

Restriction: Nested locking

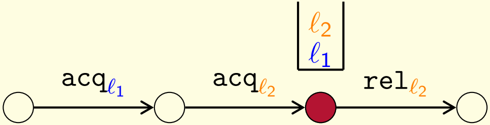
All processes acquire and release locks in a **stack-like order**, i.e., a process can only release the lock it acquired the latest.



This restricts communication between processes.

Restriction: Nested locking

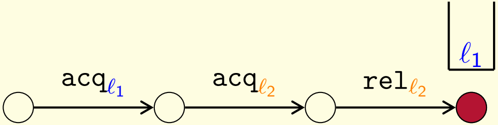
All processes acquire and release locks in a **stack-like order**, i.e., a process can only release the lock it acquired the latest.



This restricts communication between processes.

Restriction: Nested locking

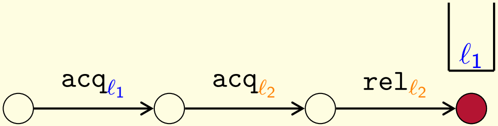
All processes acquire and release locks in a **stack-like order**, i.e., a process can only release the lock it acquired the latest.



This restricts communication between processes.

Restriction: Nested locking

All processes acquire and release locks in a **stack-like order**, i.e., a process can only release the lock it acquired the latest.



We assume nested locking in the rest of the presentation.

This restricts communication between processes.

Dynamic LSS

▷ We want to allow an unbounded number of processes and locks.

Dynamic LSS

- We want to allow an unbounded number of processes and locks.
- A process can spawn other processes

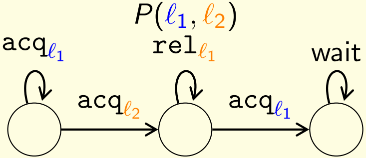
Dynamic LSS

- ▷ We want to allow an unbounded number of processes and locks.
- ▷ A process can spawn other processes
- ▷ A process takes parameters, represented by *lock variables*

$$Proc = \{P(\ell_1, \ell_2), Q(\ell_1, \ell_2, \ell_3), R(), ...\}$$

Dynamic LSS[2]

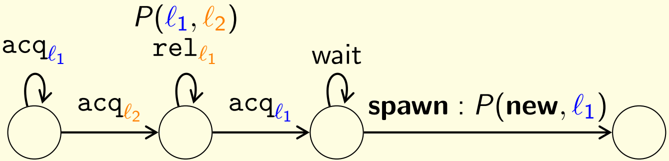
Locks :



[2]Bouajjani, Müller-Olm, Touili, CONCUR 2005 + Kenter's thesis 2022

Dynamic LSS[2]

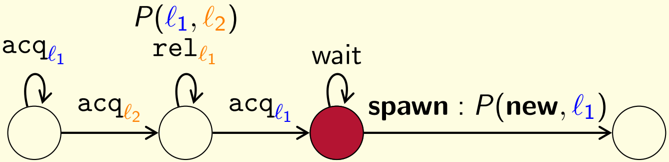
Locks :



[2] Bouajjani, Müller-Olm, Touili, CONCUR 2005 + Kenter's thesis 2022

Dynamic LSS[2]

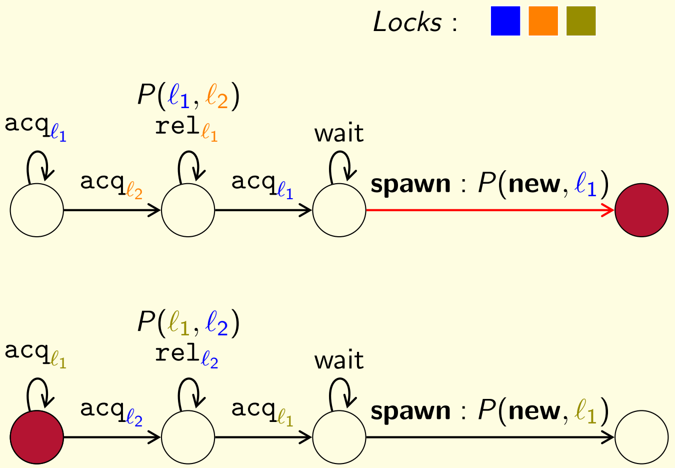
Locks :



[2]Bouajjani, Müller-Olm, Touili, CONCUR 2005 + Kenter's thesis 2022

Dynamic LSS[2]

Locks :

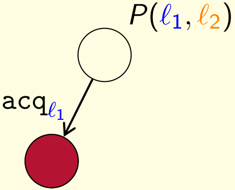


[2] Bouajjani, Müller-Olm, Touili, CONCUR 2005 + Kenter's thesis 2022

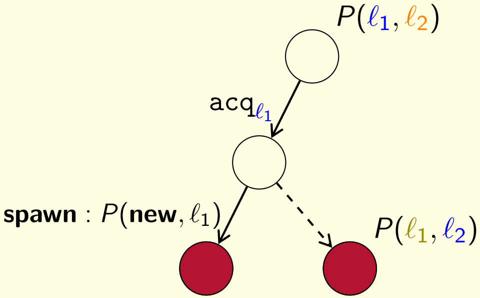
Tree representation

$P(\ell_1, \ell_2)$

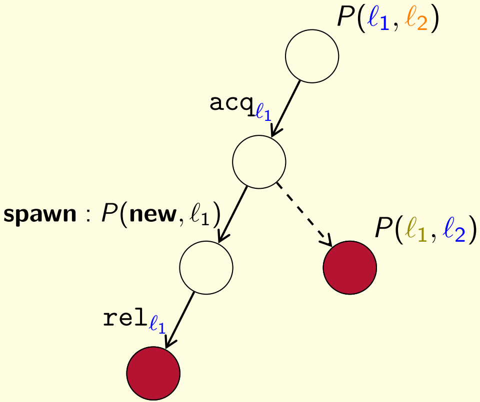
Tree representation



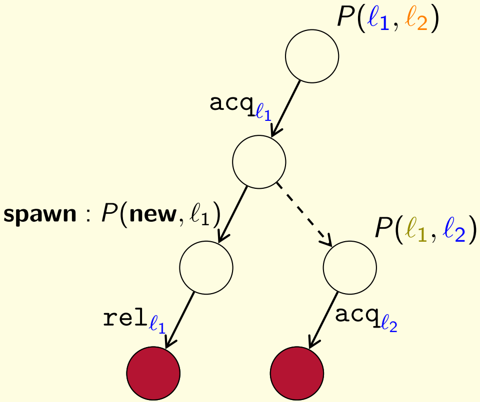
Tree representation



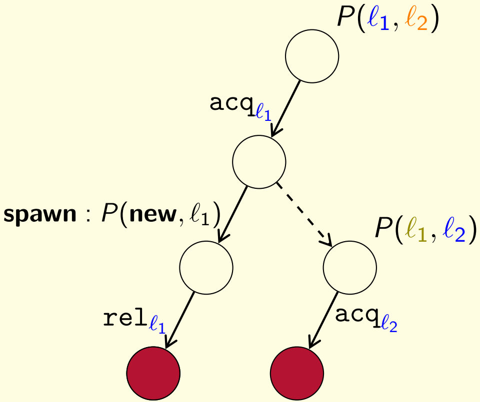
Tree representation



Tree representation

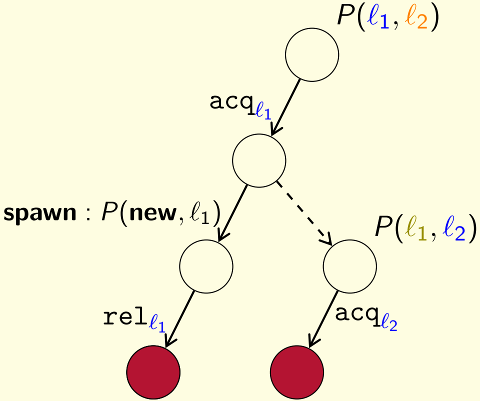


Tree representation



Specifications are ω-regular tree languages.

Tree representation



Specifications are ω-regular tree languages.

"Every process is blocked after some point"

"Finitely many processes are spawned"

"Infinitely many processes reach an error state q_{err}"

Deadlocks

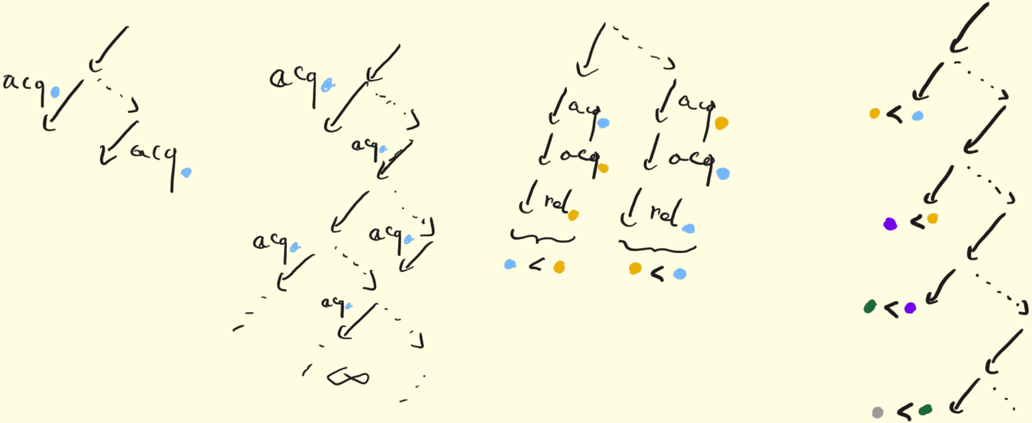
Regular model-checking problem

Input: A DLSS $\mathcal{D}$ and a parity tree automaton $\mathcal{A}$.
Output: Is there a run of $\mathcal{D}$ accepted by $\mathcal{A}$?

Regular model-checking problem

Input: A DLSS $\mathcal{D}$ and a parity tree automaton $\mathcal{A}$.
Output: Is there a run of $\mathcal{D}$ accepted by $\mathcal{A}$?

Problem: characterise trees that represent actual executions.



Lemma

The set of execution trees of a DLSS is recognised by a Büchi tree automaton of exponential size.

Lemma

The set of execution trees of a DLSS is recognised by a Büchi tree automaton of exponential size.

$P(\ell_1, \ell_2, \ell_3, \ell_4)$

For each node we guess a label of the form

- "ℓ_1 is taken and will never be released", "ℓ_2 will be acquired infinitely many times", ...
-

$$\ell_3 \begin{array}{l} < \ell_1 \\ < \ell_4 \end{array}$$

Lemma

The set of execution trees of a DLSS is recognised by a Büchi tree automaton of exponential size.

$P(\ell_1, \ell_2, \ell_3, \ell_4)$

For each node we guess a label of the form

- "ℓ_1 is taken and will never be released", "ℓ_2 will be acquired infinitely many times", ...
- $\ell_3 < \ell_1$, $\ell_3 < \ell_4$

The automaton checks that:

- the labels are consistent
- There exists a well-founded linear ordering on locks in which all local orders embed. (Technical part, also see related work [Demri Quaas, Concur '23])

Theorem [M., Muscholl, Walukiewicz Concur 2023]

Regular model-checking of DLSS is EXPTIME-complete, and
PTIME for fixed number of locks per process and parity index.

Theorem [M., Muscholl, Walukiewicz Concur 2023]

Regular model-checking of DLSS is EXPTIME-complete, and
PTIME for fixed number of locks per process and parity index.

What about pushdown processes?

Right-resetting pushdown tree automata

Right-resetting = the stack is emptied every time we go to a right child.

Lemma

Emptiness is decidable in PTIME for right-resetting parity pushdown tree automata when the parity index is fixed.

Right-resetting pushdown tree automata

Right-resetting = the stack is emptied every time we go to a right child.

Lemma

Emptiness is decidable in PTIME for right-resetting parity pushdown tree automata when the parity index is fixed.

Theorem

Regular model-checking of nested **pushdown** DLSS is EXPTIME-complete,
and PTIME when the parity index and the number of locks per process are fixed.

Right-resetting pushdown tree automata

Right-resetting = the stack is emptied every time we go to a right child.

Lemma

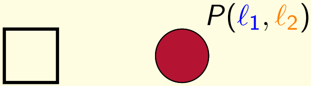
Emptiness is decidable in PTIME for right-resetting parity pushdown tree automata when the parity index is fixed.

Theorem

Regular model-checking of nested **pushdown** DLSS is EXPTIME-complete,
and PTIME when the parity index and the number of locks per process are fixed.

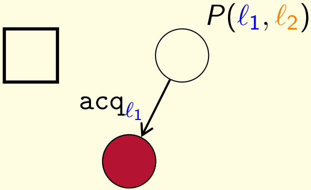
What about shared variables?

DLSS with variables



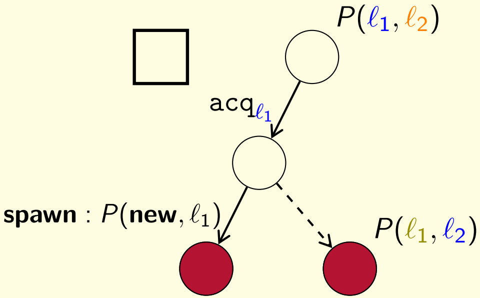
We add a register and operations `wr` and `rd` writing and reading letters from a finite alphabet in the register.

DLSS with variables



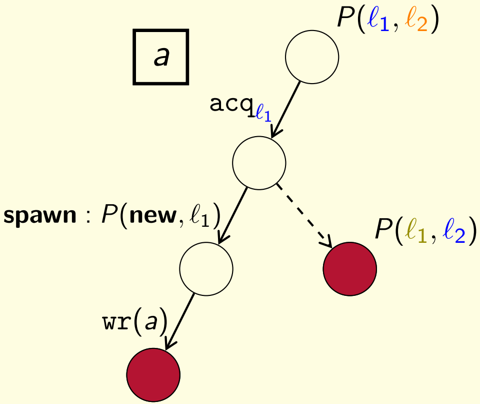
We add a register and operations `wr` and `rd` writing and reading letters from a finite alphabet in the register.

DLSS with variables



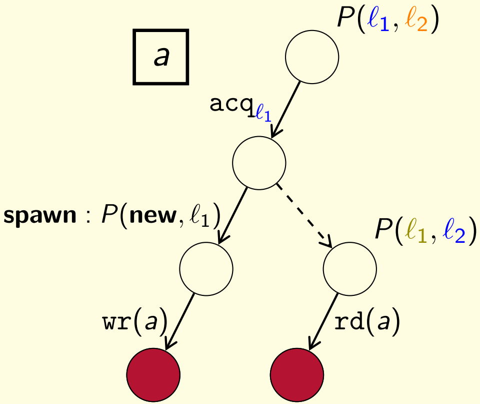
We add a register and operations `wr` and `rd` writing and reading letters from a finite alphabet in the register.

DLSS with variables



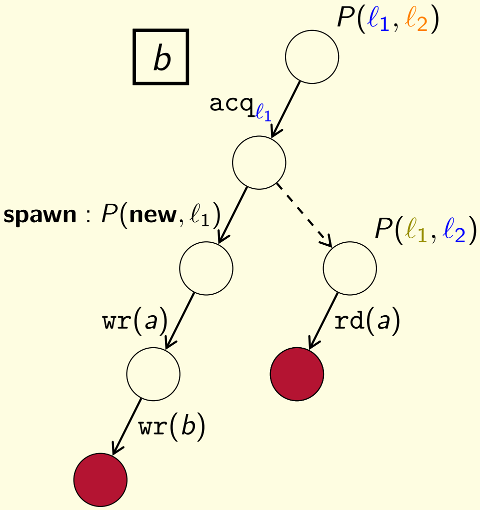
We add a register and operations `wr` and `rd` writing and reading letters from a finite alphabet in the register.

DLSS with variables



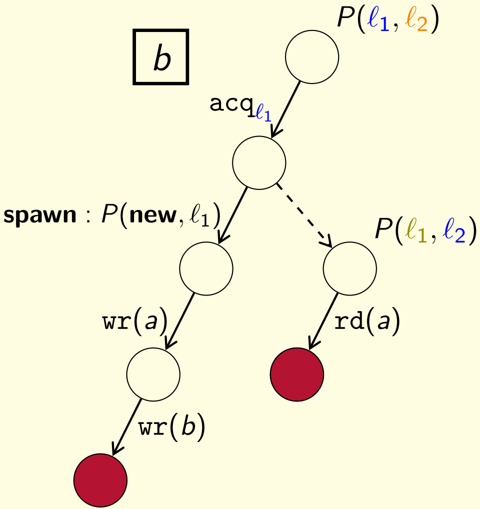
We add a register and operations `wr` and `rd` writing and reading letters from a finite alphabet in the register.

DLSS with variables



We add a register and operations `wr` and `rd` writing and reading letters from a finite alphabet in the register.

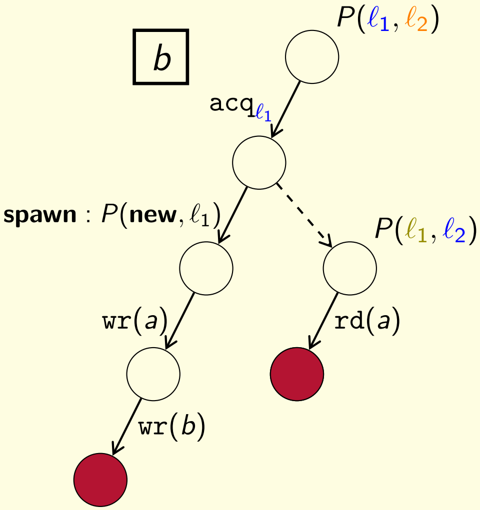
DLSS with variables



We add a register and operations `wr` and `rd` writing and reading letters from a finite alphabet in the register.

Sets of runs are no longer regular.

DLSS with variables



We add a register and operations `wr` and `rd` writing and reading letters from a finite alphabet in the register.

Sets of runs are no longer regular.

Theorem

State reachability is undecidable for DLSS with variables.

Bounded writer reversals

Writer reversal = the process writing in the shared register changes.

[3] Atig, Bouajjani, Kumar, Saivasan FSTTCS 2014

Bounded writer reversals

Writer reversal = the process writing in the shared register changes.

Theorem

State reachability is decidable for DLSSV with bounded writer reversals.

[3] Atig, Bouajjani, Kumar, Saivasan FSTTCS 2014

Bounded writer reversals

Writer reversal = the process writing in the shared register changes.

Theorem

State reachability is decidable for DLSSV with bounded writer reversals.

It is undecidable when the processes are pushdown systems[3].

[3] Atig, Bouajjani, Kumar, Saivasan FSTTCS 2014

Proof sketch

Consider a run with one process writing and others reading.

Proof sketch

Consider a run with one process writing and others reading.

Phase: run section where

- the writer is in the same state and has the same locks at the start and at the end,
- none of the locks used by the writers in the phase are held by another process at the start or the end

Proof sketch

Consider a run with one process writing and others reading.

Phase: run section where

- the writer is in the same state and has the same locks at the start and at the end,
- none of the locks used by the writers in the phase are held by another process at the start or the end

Lemma

Every finite run with a single writer can be cut into $2^{O(|Q|)}$ phases.

Proof sketch

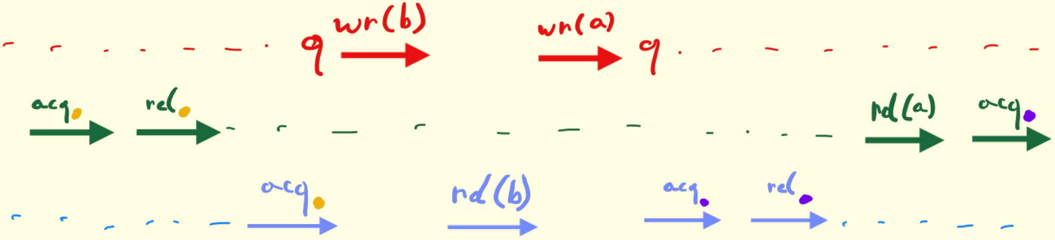
Consider one phase.

Proof sketch

Consider one phase.

Lemma

Every phase can be replaced by a sequence of phases where at most one reader moves.

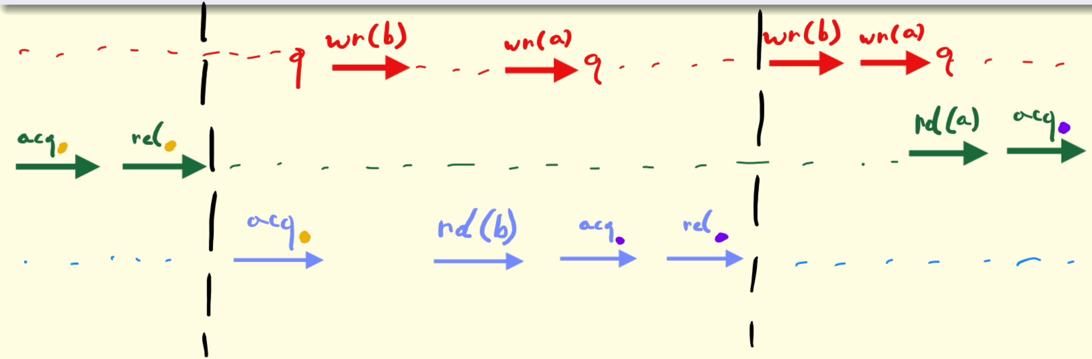


Proof sketch

Consider one phase.

Lemma

Every phase can be replaced by a sequence of phases where at most one reader moves.



Proof sketch

Consider one phase.

Lemma

Every phase can be replaced by a sequence of phases where at most one reader moves.

Construct $\mathcal{A}$ that:

- guesses a partition of the tree in $K2^{O(|Q|)}$ phases, each with a single writer.
- checks lock conditions
- checks compatibility of each reader with the writer

Proof sketch

Consider one phase.

Lemma

Every phase can be replaced by a sequence of phases where at most one reader moves.

Construct $\mathcal{A}$ that:

- guesses a partition of the tree in $K2^{O(|Q|)}$ phases, each with a single writer.
- checks lock conditions
- checks compatibility of each reader with the writer

To sum up

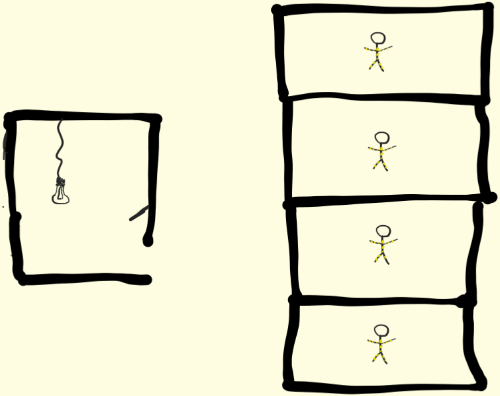
	State reach	ω-regular
Nested LSS	✓	✓
Nested Dynamic LSS	✓	✓
Nested DLSS + var with bounded v.r.	✓	?
Nested DLSS + var	✗	✗

Part II

Controller Synthesis

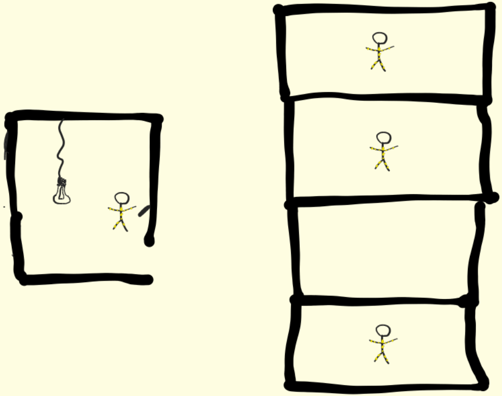
The prisoners and the lightbulb

- 4 prisoners are waiting in their cells



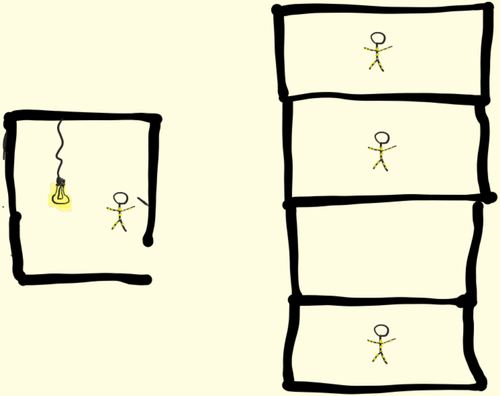
The prisoners and the lightbulb

- 4 prisoners are waiting in their cells
- Everyday, one is picked at random and taken to a room with a lightbulb and a switch, then brought back to the cell.



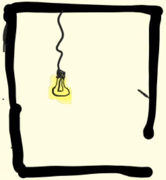
The prisoners and the lightbulb

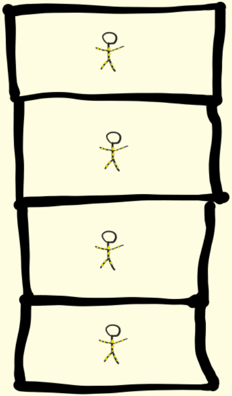
- 4 prisoners are waiting in their cells
- Everyday, one is picked at random and taken to a room with a lightbulb and a switch, then brought back to the cell.



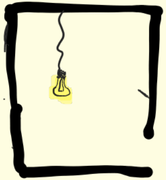
The prisoners and the lightbulb

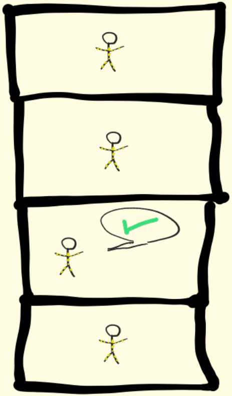
- 4 prisoners are waiting in their cells
- Everyday, one is picked at random and taken to a room with a lightbulb and a switch, then brought back to the cell.





The prisoners and the lightbulb





- 4 prisoners are waiting in their cells
- Everyday, one is picked at random and taken to a room with a lightbulb and a switch, then brought back to the cell.
- At any point a prisoner can claim that all prisoners have been in the cell at least once.
  They win if it is true, otherwise they lose.

Formal problem

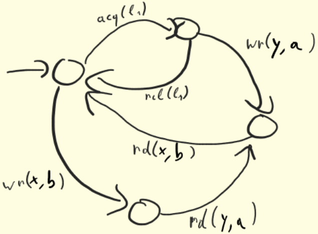
We have:

- A finite set of processes 
- A finite set of variables 
- A finite set of locks 

Each process is a finite-state transition system with operations

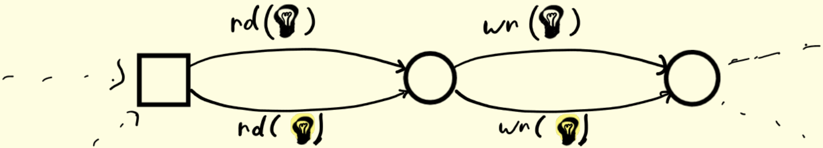
$acq(\ell)$, $rel(\ell)$

$wr(x, a)$, $rd(x, a)$



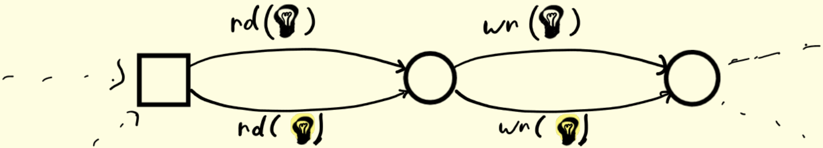
Formal problem

Processes have controllable ○ and uncontrollable □ states.



Formal problem

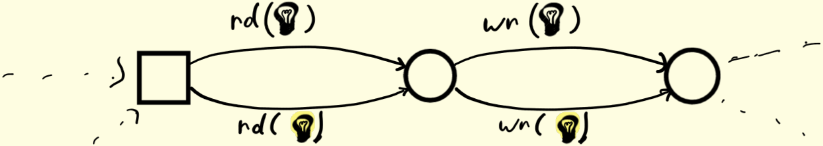
Processes have controllable ○ and uncontrollable □ states.



A **strategy** σ_p for process p is a function choosing the next action of p from controllable states with as input the local run seen so far.

Formal problem

Processes have controllable ○ and uncontrollable □ states.



A **strategy** σ_p for process p is a function choosing the next action of p from controllable states with as input the local run seen so far.

Specifications $=$ boolean combinations of local regular conditions.

- Processes have controllable and uncontrollable states
- Strategies are local, ie, only use the sequence of local actions of the process as input.
- Every copy of each process uses the same strategy

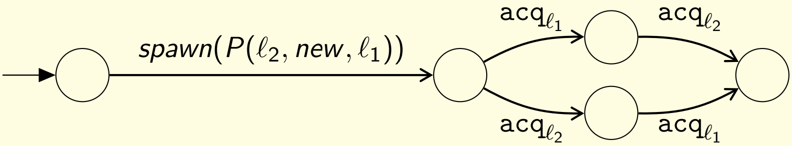
- Processes have controllable and uncontrollable states
- Strategies are local, ie, only use the sequence of local actions of the process as input.
- Every copy of each process uses the same strategy

Problem

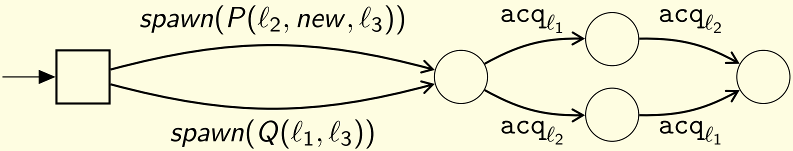
Is there a strategy $\sigma = (\sigma_p)_{p \in Proc}$ ensuring that there is no execution accepted by $\mathcal{A}$?

Example

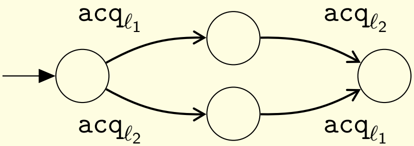
$P_{init}(\ell_1, \ell_2)$:



$P(\ell_1, \ell_2, \ell_3)$:



$Q(\ell_1, \ell_2)$:



With locks only (DLSS)

Problem

Is there a strategy $\sigma = (\sigma_p)_{p \in Proc}$ ensuring that there is no execution accepted by $\mathcal{A}$?

With locks only (DLSS)

Problem

Is there a strategy $\sigma = (\sigma_p)_{p \in Proc}$ ensuring that there is no execution accepted by $\mathcal{A}$?

Lemma

The set of execution trees of a DLSS is recognised by a Büchi tree automaton of exponential size.

With locks only (DLSS)

Problem

Is there a strategy $\sigma = (\sigma_p)_{p \in Proc}$ ensuring that there is no execution accepted by $\mathcal{A}$?

Lemma

The set of execution trees of a DLSS is recognised by a Büchi tree automaton of exponential size.

Corollary

There is a tree automaton $\mathcal{T}$ recognising executions of $\mathcal{D}$ that are accepted by $\mathcal{A}$.

With locks only (DLSS)

Problem

Is there a strategy $\sigma = (\sigma_p)_{p \in Proc}$ ensuring that there is no execution accepted by $\mathcal{A}$?

Lemma

The set of execution trees of a DLSS is recognised by a Büchi tree automaton of exponential size.

Corollary

There is a tree automaton $\mathcal{T}$ recognising executions of $\mathcal{D}$ that are accepted by $\mathcal{A}$.

Strategy $(\sigma_p)_{p \in Proc} \rightarrow$ set of local runs

Is there a strategy such that we cannot form a tree accepted by $\mathcal{T}$ whose left branches are those local runs?

With locks only (DLSS)

Corollary

There is a parity tree automaton $\mathcal{T}$ recognising executions of $\mathcal{D}$ that are accepted by $\mathcal{A}$.

With locks only (DLSS)

Corollary

There is a parity tree automaton $\mathcal{T}$ recognising executions of $\mathcal{D}$ that are accepted by $\mathcal{A}$.

A strategy σ defines a set of local runs, i.e., left branches of execution trees.

A *labelled* left branch is a left branch annotated with states of $\mathcal{T}$.

With locks only (DLSS)

Corollary

There is a parity tree automaton $\mathcal{T}$ recognising executions of $\mathcal{D}$ that are accepted by $\mathcal{A}$.

A strategy σ defines a set of local runs, i.e., left branches of execution trees.
A *labelled* left branch is a left branch annotated with states of $\mathcal{T}$.

Definition

The *profile* of a labelled left branch is a tuple (p, s, π) with:

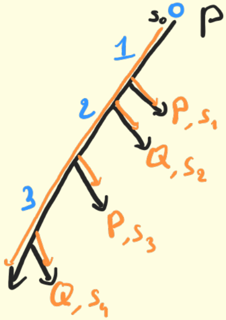
- p is the process executing in this branch
- s is the state labelling the first node of the branch
- $\pi : Prio \to 2^{Proc \times S_{\mathcal{T}}}$ is a function mapping each priority i to the set of (p', s') such that p' is spawned at a node labelled by s' while the highest priority seen before is i.

With locks only (DLSS)

Corollary

There is a parity tree automaton $\mathcal{T}$ recognising executions of $\mathcal{D}$ that are accepted by $\mathcal{A}$.

A strategy σ defines a set of local runs, i.e., left branches of execution trees.
A *labelled* left branch is a left branch annotated with states of $\mathcal{T}$.



$$\rightsquigarrow \left(P,\ s_0,\ \begin{array}{l} 0 \longrightarrow \emptyset \\ 1 \longrightarrow \{(s_1, P), (s_2, Q)\} \\ 2 \longrightarrow \{(s_3, P)\} \\ 3 \longrightarrow \{(s_4, Q)\} \end{array}\right)$$

With locks only (DLSS)

The behaviour of a strategy $\sigma = (\sigma_p)_{p \in Proc}$ is the set of profiles of accepting labelled left branches (= local runs) compatible with it.

Lemma

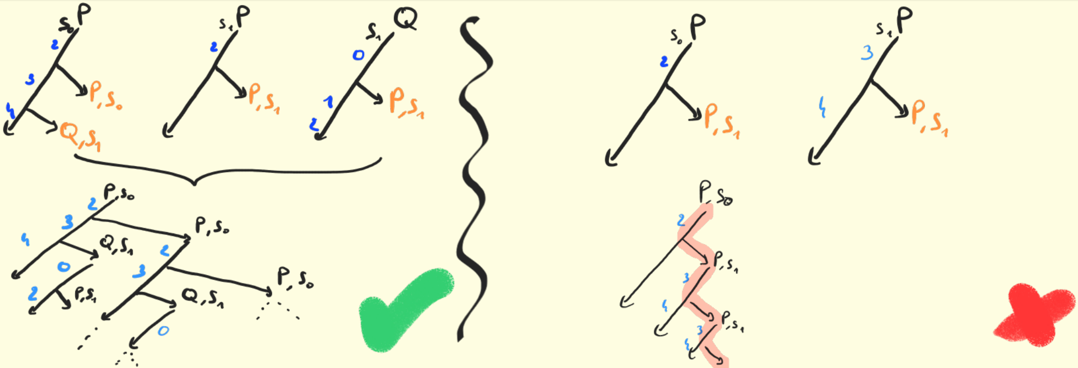
Whether $(\sigma_p)_{p \in Proc}$ is winning only depends on its behaviour.

With locks only (DLSS)

The behaviour of a strategy $\sigma = (\sigma_p)_{p \in Proc}$ is the set of profiles of accepting labelled left branches (= local runs) compatible with it.

Lemma

Whether $(\sigma_p)_{p \in Proc}$ is winning only depends on its behaviour.



With locks only (DLSS)

Theorem

The controller synthesis problem is decidable over DLSS.

With locks only (DLSS)

Theorem

The controller synthesis problem is decidable over DLSS.

Algorithm:

- Enumerate sets of profiles
- For each one, test whether there is a strategy yielding that set of profiles
- If there is one, $\underbrace{\text{there is one with bounded memory}}_{\text{From } \omega\text{-regular games}}$: check whether it is winning.

With locks and variables

With variables, none of this works!

- Sets of execution trees are not regular
- "Pumping argument" used for verification does not extend to adversarial setting.

What is left to do

Conjecture

Verification of nested DLSS with variables and bounded writer reversals against ω-regular tree specifications is decidable.

What is left to do

Conjecture

Verification of nested DLSS with variables and bounded writer reversals against ω-regular tree specifications is decidable.

Conjecture

Controller synthesis of nested DLSS with variables and bounded writer reversals against ω-regular tree specifications is decidable.

Formal problem

Problem

Given a system with processes, locks and variables and a specification φ, can we find a family of strategies $(\sigma_p)_{p \in \mathtt{Proc}}$ guaranteeing φ?

Problem 1

I: A system $\mathcal{S}$, $K \in \mathbb{N}$ and a specification φ
O: Is there a family of strategies $(\sigma_p)_{p \in \mathtt{Proc}}$ guaranteeing φ over runs with $\leq K$ writer reversals?

Problem 2

I: A system $\mathcal{S}$, $K \in \mathbb{N}$ and a specification φ
O: Is there a family of strategies $(\sigma_p)_{p \in \mathtt{Proc}}$ guaranteeing φ and that all runs have $\leq K$ writer reversals?

Problem 1

I: A system $\mathcal{S}$, $K \in \mathbb{N}$ and a specification φ
O: Is there a family of strategies $(\sigma_p)_{p \in \mathtt{Proc}}$ guaranteeing φ over runs with $\leq K$ writer reversals?

Problem 2

I: A system $\mathcal{S}$, $K \in \mathbb{N}$ and a specification φ
O: Is there a family of strategies $(\sigma_p)_{p \in \mathtt{Proc}}$ guaranteeing φ and that all runs have $\leq K$ writer reversals?

Algorithm:

For $K = 0$ to $+\infty$ do
    If $\exists(\sigma_p)$ such that $\phi \wedge \leq K$ $\rightarrow$ return YES
    If $\nexists(\sigma_p)$ such that $\leq K \Rightarrow \phi$ $\rightarrow$ return NO

Problem 1

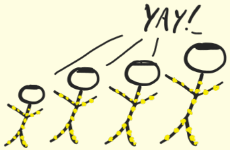
I: A system $\mathcal{S}$, $K \in \mathbb{N}$ and a specification φ
O: Is there a family of strategies $(\sigma_p)_{p \in \mathtt{Proc}}$ guaranteeing φ over runs with $\leq K$ writer reversals?

Problem 2

I: A system $\mathcal{S}$, $K \in \mathbb{N}$ and a specification φ
O: Is there a family of strategies $(\sigma_p)_{p \in \mathtt{Proc}}$ guaranteeing φ and that all runs have $\leq K$ writer reversals?

Algorithm:

For $K = 0$ to $+\infty$ do
    If $\exists (\sigma_p)$ such that $\phi \wedge \leq K$ $\rightarrow$ return YES
    If $\nexists (\sigma_p)$ such that $\leq K \Rightarrow \phi$ $\rightarrow$ return NO



Other directions

- General approach to local controller synthesis
- Parameterised complexity
- Strategies using more information

Other directions

- General approach to local controller synthesis
- Parameterised complexity
- Strategies using more information

Thanks!