Verification and Synthesis of Distributed Systems with Weak Synchronisation

Corto Mascle

LaBRI

université de BORDEAUX

Motivation: reactive synthesis [Church '63]

- A system interacts with an environment
- A controller has to choose actions at runtime to achieve a given specification

Motivation: reactive synthesis [Church '63]

- A system interacts with an environment
- A controller has to choose actions at runtime to achieve a given specification

System

Specification

"Maintain direction between NE and NW and speed at most v"

Motivation: reactive synthesis [Church '63]

- A system interacts with an environment
- A controller has to choose actions at runtime to achieve a given specification

System

Specification

"Maintain direction between NE and NW and speed at most v"

Goal

Automatically design controllers from the system and the specification.

Motivation: reactive synthesis [Church '63]

- A system interacts with an environment
- A controller has to choose actions at runtime to achieve a given specification

System

Specification

"Maintain direction between NE and NW and speed at most v"

Goal

Automatically design controllers from the system and the specification.

Motivation: reactive synthesis [Church '63]

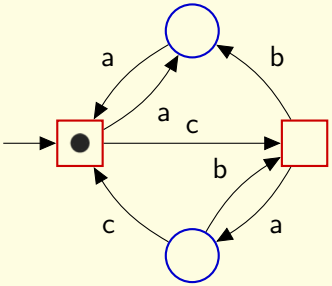
"Maintain direction between NE and NW and speed at most v"

Interpret this as a game between two players. **[Büchi, Landweber '69]**

Motivation: reactive synthesis [Church '63]

"Maintain direction between NE and NW and speed at most v"

Interpret this as a game between two players. [Büchi, Landweber '69]



Nodes = Configurations

◯ = Controller chooses the next action

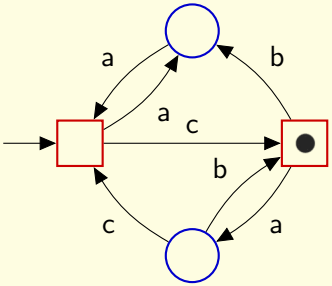
□ = Environment chooses

Specification = constraint on the sequence of nodes visited

Motivation: reactive synthesis [Church '63]

"Maintain direction between NE and NW and speed at most v"

Interpret this as a game between two players. [Büchi, Landweber '69]



Nodes = Configurations

◯ = Controller chooses the next action

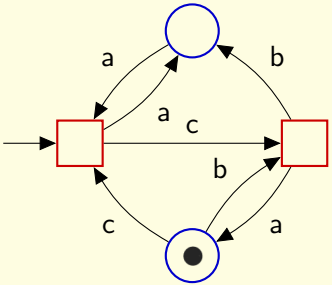
□ = Environment chooses

Specification = constraint on the sequence of nodes visited

Motivation: reactive synthesis [Church '63]

"Maintain direction between NE and NW and speed at most v"

Interpret this as a game between two players. **[Büchi, Landweber '69]**



Nodes = Configurations

◯ = Controller chooses the next action

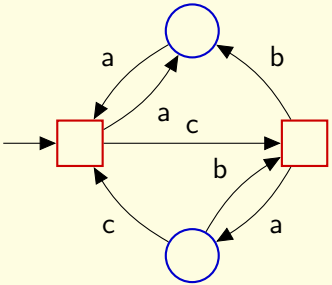
□ = Environment chooses

Specification = constraint on the sequence of nodes visited

Motivation: reactive synthesis [Church '63]

"Maintain direction between NE and NW and speed at most v"

Interpret this as a game between two players. **[Büchi, Landweber '69]**

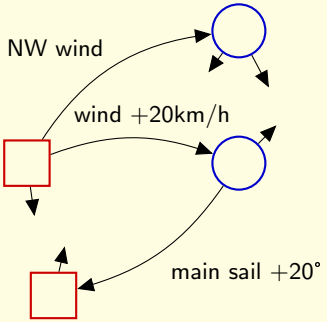


Nodes = Configurations

◯ = Controller chooses the next action

□ = Environment chooses

Specification = constraint on the sequence of nodes visited



Distributed controller synthesis

- Several components interact with an environment
- Each component has a controller
- Each controller only has a partial view of the system

$\rightsquigarrow$ Multiplayer games with partial information (e.g.,[Mohalik, Walukiewicz '03])

Reduction to two-player games does not apply!

Distributed synthesis

- Fixed number of processes

Distributed synthesis

- Fixed number of processes
- Global objective

Distributed synthesis

- Fixed number of processes
- Global objective
- Are there local strategies (one for each process) guaranteeing the objective?

Distributed synthesis

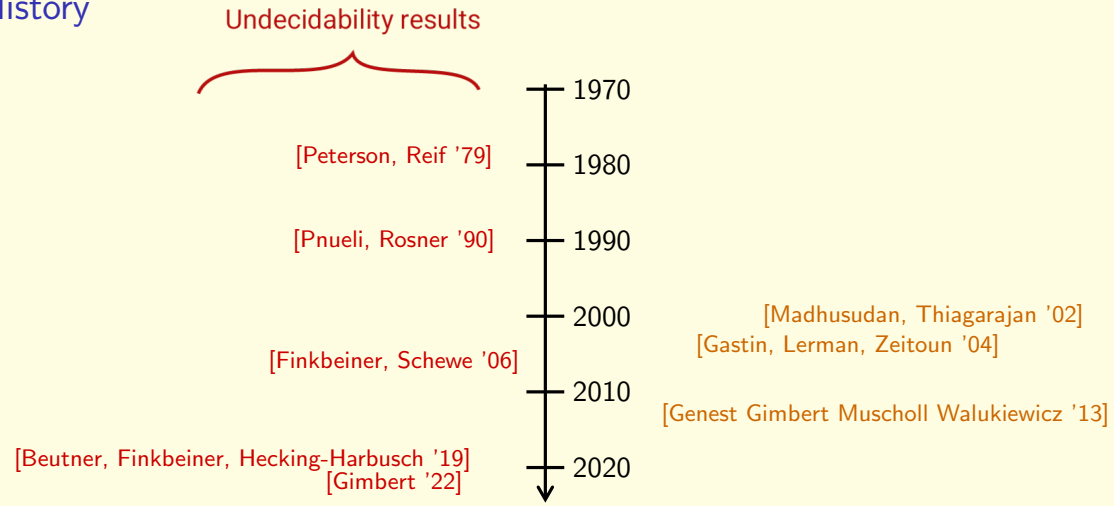
- Fixed number of processes
- Global objective
- Are there local strategies (one for each process) guaranteeing the objective?

Parameterised distributed synthesis

(e.g. [Bertrand, Fournier, Sangnier '15], [Stan '17])

- Unknown number of identical processes
- Global objective
- Is there a local strategy (the same for all processes) guaranteeing the objective?

History


Undecidability results
1970
[Peterson, Reif '79]
1980
[Pnueli, Rosner '90]
1990
2000
[Madhusudan, Thiagarajan '02]
[Gastin, Lerman, Zeitoun '04]
[Finkbeiner, Schewe '06]
2010
[Genest Gimbert Muscholl Walukiewicz '13]
[Beutner, Finkbeiner, Hecking-Harbusch '19]
[Gimbert '22]
2020

What if...

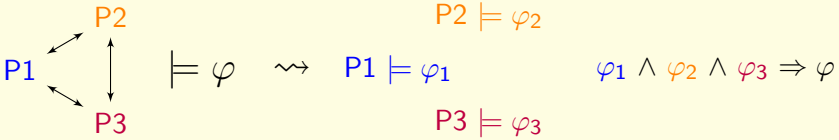
What if...

... we could still reduce to two-player games?

What if...

... we could still reduce to two-player games?

- Decompose the global specification into local objectives for each component
- Solve local control problems



Plan

Consider **asynchronous** models of distributed systems, with weak synchronisation.

1. Design local invariants that guarantee the specification
2. Use those invariants as local objectives for each process
3. Solve local synthesis problems on single processes

Plan

Consider **asynchronous** models of distributed systems, with weak synchronisation.

1. Design local invariants that guarantee the specification
2. Use those invariants as local objectives for each process
3. Solve local synthesis problems on single processes

Broadcast Networks of Register Automata

Lock-sharing systems

Dynamic lock-sharing systems

Plan

Consider **asynchronous** models of distributed systems, with weak synchronisation.

1. Design local invariants that guarantee the specification
2. Use those invariants as local objectives for each process
3. Solve local synthesis problems on single processes

Broadcast Networks of Register Automata
A parameterised model

Lock-sharing systems

A model with fixed processes

Dynamic lock-sharing systems
A model with dynamic process creation

Plan

Consider **asynchronous** models of distributed systems, with weak synchronisation.

1. Design local invariants that guarantee the specification
2. Use those invariants as local objectives for each process
3. Solve local synthesis problems on single processes

Broadcast Networks of Register Automata
A parameterised model

Lock-sharing systems

A model with fixed processes

Dynamic lock-sharing systems
A model with dynamic process creation

Let's apply it!



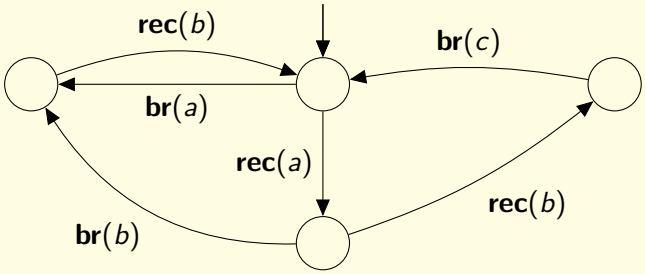

Part I.A

Broadcast Networks

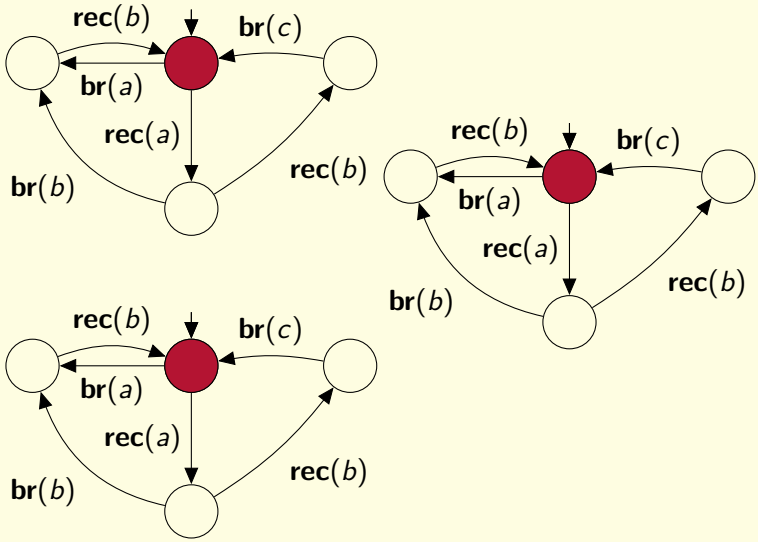




Reconfigurable Broadcast Networks [Delzanno, Sangnier, Traverso '10]

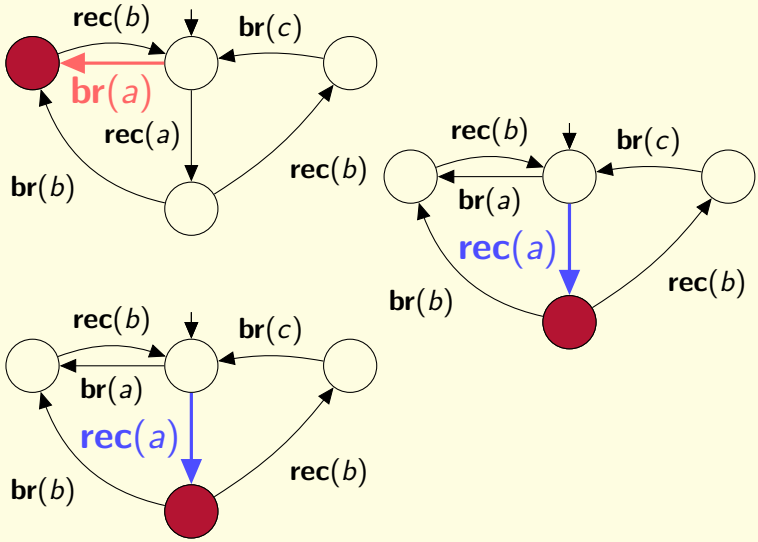


Reconfigurable Broadcast Networks



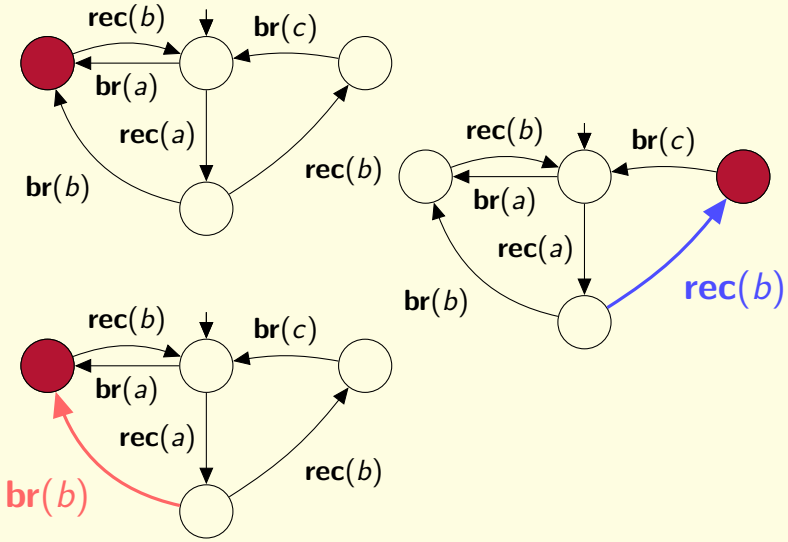
- Arbitrary number of processes at the start
- Lossy broadcast
- "Equivalent" to systems with a shared variable
  [Balasubramanian, Weil-Kennedy '21]

Reconfigurable Broadcast Networks



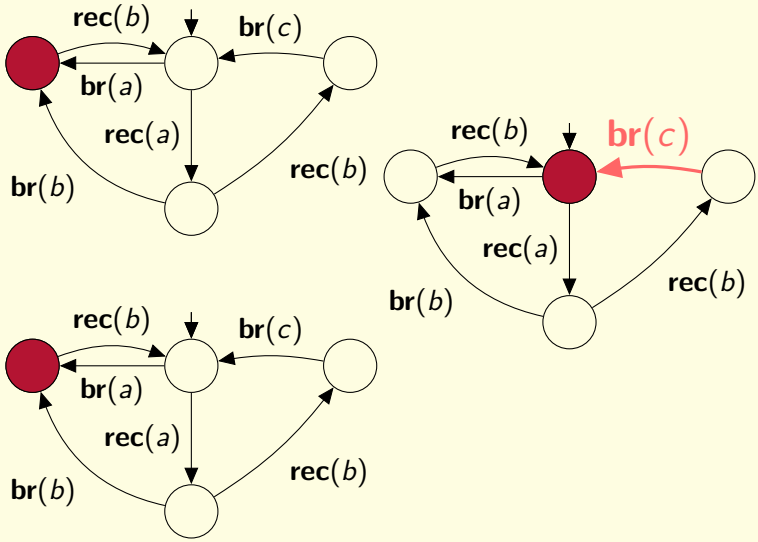
- Arbitrary number of processes at the start
- Lossy broadcast
- "Equivalent" to systems with a shared variable
  [Balasubramanian, Weil-Kennedy '21]

Reconfigurable Broadcast Networks



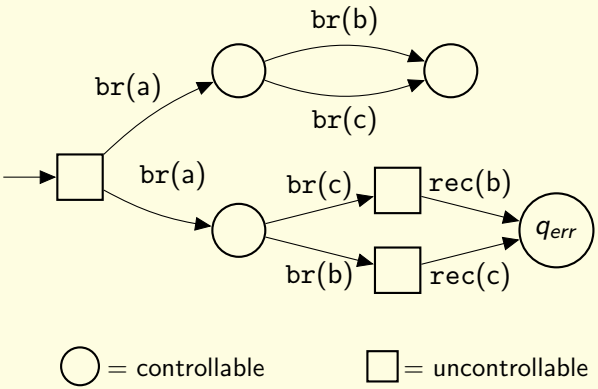
- Arbitrary number of processes at the start
- Lossy broadcast
- "Equivalent" to systems with a shared variable
  [Balasubramanian, Weil-Kennedy '21]

Reconfigurable Broadcast Networks



- Arbitrary number of processes at the start
- Lossy broadcast
- "Equivalent" to systems with a shared variable
  [Balasubramanian, Weil-Kennedy '21]

Reconfigurable Broadcast Games



Δ = set of transitions

Strategy = function
$\sigma : \Delta^* \to \Delta$.

σ-run = run where every process chooses transitions from controllable states according to σ

Reconfigurable Broadcast Games

$\mathcal{G}_\sigma$ = the (potentially infinite) RBN obtained by applying strategy σ in RBG $\mathcal{G}$.

Reconfigurable Broadcast Games

$\mathcal{G}_\sigma$ = the (potentially infinite) RBN obtained by applying strategy σ in RBG $\mathcal{G}$.

Problems

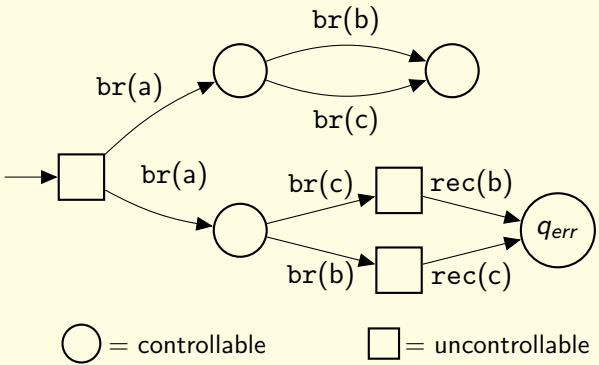
SafeStrat: Given an RBG $\mathcal{G}$, is there a strategy σ such that q_{err} is not coverable in $\mathcal{G}_\sigma$?

Reconfigurable Broadcast Games

$\mathcal{G}_\sigma$ = the (potentially infinite) RBN obtained by applying strategy σ in RBG $\mathcal{G}$.

Problems

SafeStrat: Given an RBG $\mathcal{G}$, is there a strategy σ such that q_{err} is not coverable in $\mathcal{G}_\sigma$?

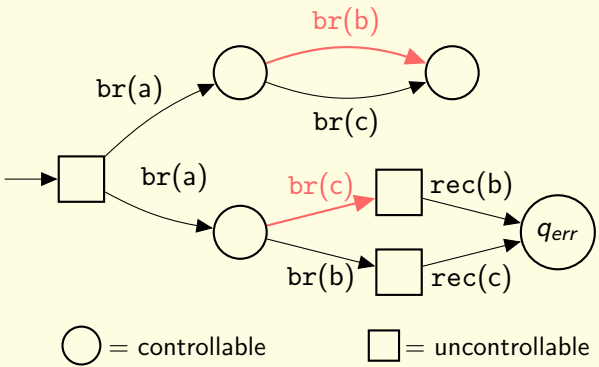


Reconfigurable Broadcast Games

$\mathcal{G}_\sigma$ = the (potentially infinite) RBN obtained by applying strategy σ in RBG $\mathcal{G}$.

Problems

SafeStrat: Given an RBG $\mathcal{G}$, is there a strategy σ such that q_{err} is not coverable in $\mathcal{G}_\sigma$?

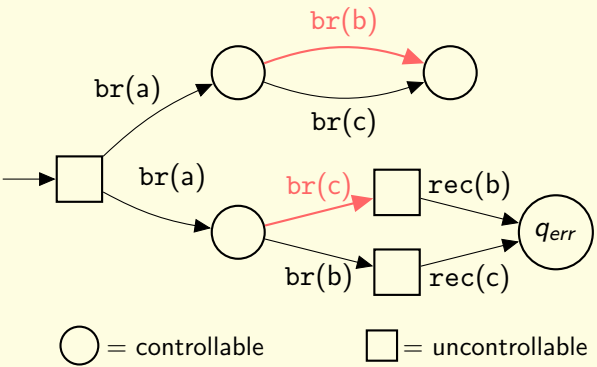


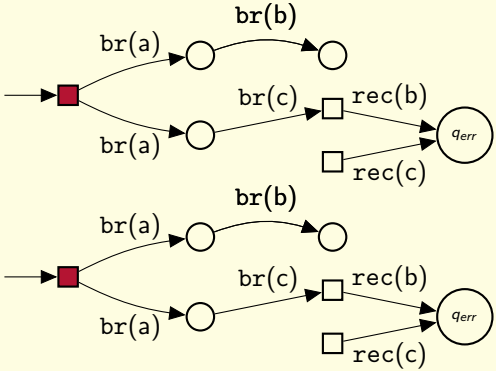
Reconfigurable Broadcast Games

$\mathcal{G}_\sigma$ = the (potentially infinite) RBN obtained by applying strategy σ in RBG $\mathcal{G}$.

Problems

SafeStrat: Given an RBG $\mathcal{G}$, is there a strategy σ such that q_{err} is not coverable in $\mathcal{G}_\sigma$?



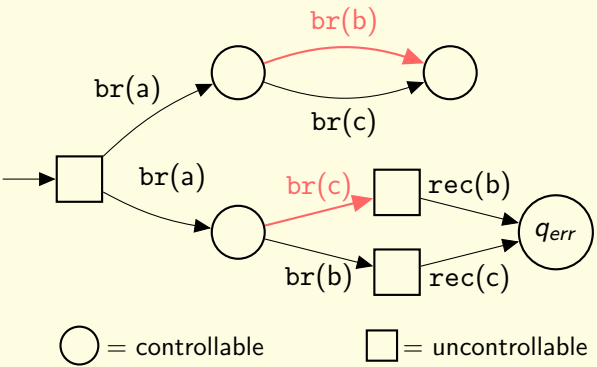


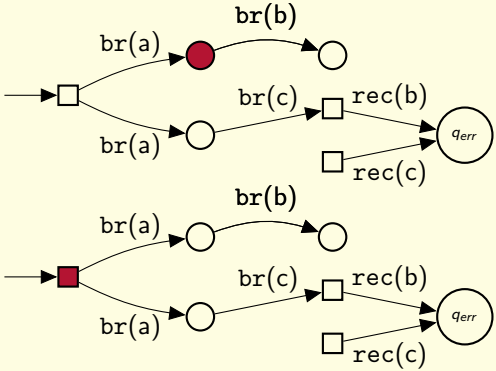
Reconfigurable Broadcast Games

$\mathcal{G}_\sigma$ = the (potentially infinite) RBN obtained by applying strategy σ in RBG $\mathcal{G}$.

Problems

SafeStrat: Given an RBG $\mathcal{G}$, is there a strategy σ such that q_{err} is not coverable in $\mathcal{G}_\sigma$?



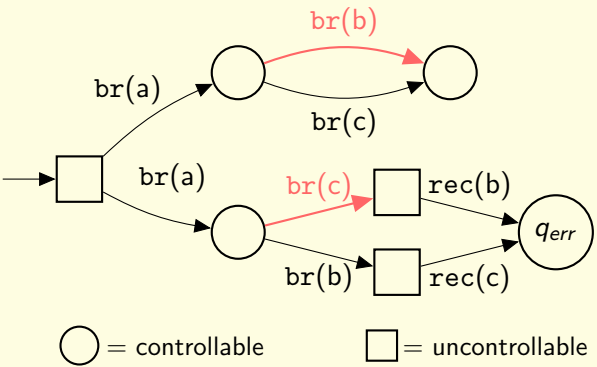


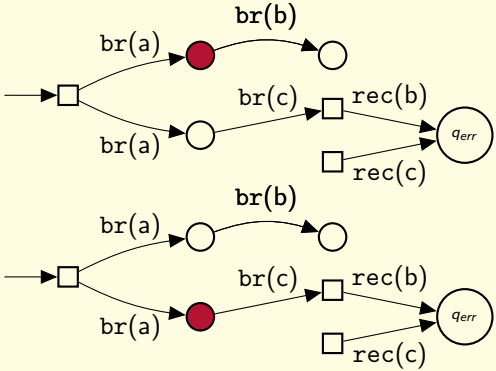
Reconfigurable Broadcast Games

$\mathcal{G}_\sigma$ = the (potentially infinite) RBN obtained by applying strategy σ in RBG $\mathcal{G}$.

Problems

SafeStrat: Given an RBG $\mathcal{G}$, is there a strategy σ such that q_{err} is not coverable in $\mathcal{G}_\sigma$?



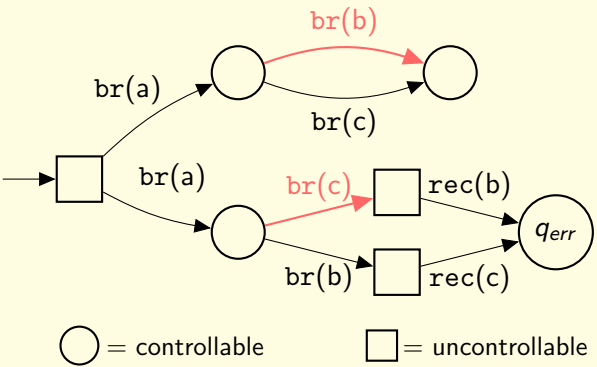


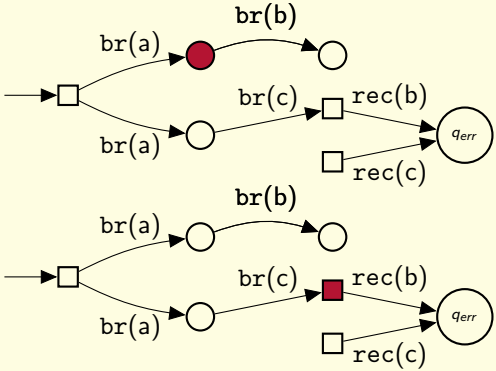
Reconfigurable Broadcast Games

$\mathcal{G}_\sigma$ = the (potentially infinite) RBN obtained by applying strategy σ in RBG $\mathcal{G}$.

Problems

SafeStrat: Given an RBG $\mathcal{G}$, is there a strategy σ such that q_{err} is not coverable in $\mathcal{G}_\sigma$?



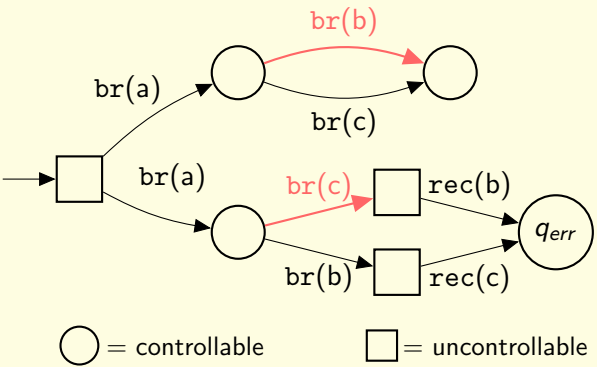


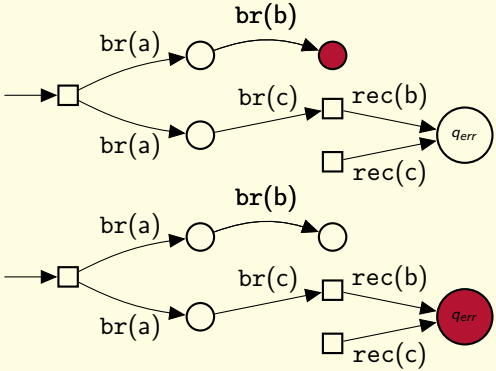
Reconfigurable Broadcast Games

$\mathcal{G}_\sigma$ = the (potentially infinite) RBN obtained by applying strategy σ in RBG $\mathcal{G}$.

Problems

SafeStrat: Given an RBG $\mathcal{G}$, is there a strategy σ such that q_{err} is not coverable in $\mathcal{G}_\sigma$?



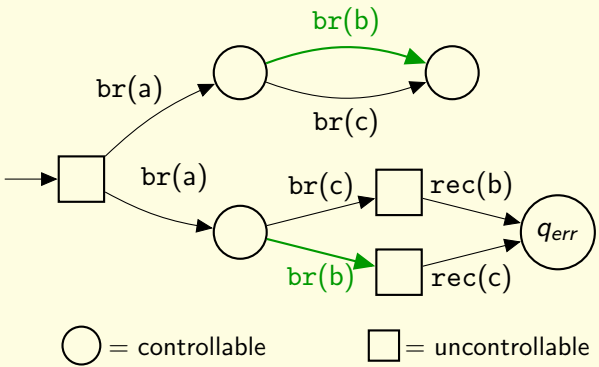


Reconfigurable Broadcast Games

$\mathcal{G}_\sigma$ = the (potentially infinite) RBN obtained by applying strategy σ in RBG $\mathcal{G}$.

Problems

SafeStrat: Given an RBG $\mathcal{G}$, is there a strategy σ such that q_{err} is not coverable in $\mathcal{G}_\sigma$?



Decidability

Lemma

There is no run covering q_{err} if and only if there exists an **invariant** $\mathcal{I} \subseteq \Sigma$ *such that*

1. *Every local run that only receives messages of $\mathcal{I}$ broadcasts only messages of $\mathcal{I}$*
2. *Every local run reaching q_{err} uses a reception* $\mathbf{rec}(m)$ *with* $m \notin \mathcal{I}$

Decidability

Lemma

There is no run covering q_{err} if and only if there exists an **invariant** $\mathcal{I} \subseteq \Sigma$ *such that*

1. *Every local run that only receives messages of $\mathcal{I}$ broadcasts only messages of $\mathcal{I}$*
2. *Every local run reaching q_{err} uses a reception* **rec**(m) *with $m \notin \mathcal{I}$*

$\mathcal{R}$ is safe $\Leftrightarrow \exists \mathcal{I}$ s.t. $\mathcal{R}$ satisfies 1 and 2 w.r.t $\mathcal{I}$.

Decidability

Lemma

There is no run covering q_{err} if and only if there exists an **invariant** $\mathcal{I} \subseteq \Sigma$ *such that*

1. *Every local run that only receives messages of $\mathcal{I}$ broadcasts only messages of $\mathcal{I}$*
2. *Every local run reaching q_{err} uses a reception* **rec**(m) *with $m \notin \mathcal{I}$*

$\mathcal{R}$ is safe $\Leftrightarrow \exists \mathcal{I}$ s.t. $\mathcal{R}$ satisfies 1 and 2 w.r.t $\mathcal{I}$.

$\exists \sigma$ s.t. $\mathcal{G}_\sigma$ is safe $\Leftrightarrow \exists \sigma, \exists \mathcal{I}$ s.t. $\mathcal{G}_\sigma$ satisfies 1 and 2 w.r.t $\mathcal{I}$.

Decidability

Lemma

There is no run covering q_{err} if and only if there exists an **invariant** $\mathcal{I} \subseteq \Sigma$ *such that*

1. *Every local run that only receives messages of $\mathcal{I}$ broadcasts only messages of $\mathcal{I}$*
2. *Every local run reaching q_{err} uses a reception* **rec**(m) *with $m \notin \mathcal{I}$*

$$\mathcal{R} \text{ is safe} \Leftrightarrow \exists \mathcal{I} \text{ s.t. } \mathcal{R} \text{ satisfies 1 and 2 w.r.t } \mathcal{I} .$$
$$\exists \sigma \text{ s.t. } \mathcal{G}_\sigma \text{ is safe} \Leftrightarrow \exists \sigma, \exists \mathcal{I} \text{ s.t. } \mathcal{G}_\sigma \text{ satisfies 1 and 2 w.r.t } \mathcal{I} .$$
$$\exists \sigma \text{ s.t. } \mathcal{G}_\sigma \text{ is safe} \Leftrightarrow \exists \mathcal{I} \text{ , } \exists \sigma \text{ s.t. } \mathcal{G}_\sigma \text{ satisfies 1 and 2 w.r.t } \mathcal{I} .$$

Decidability

Lemma

There is no run covering q_{err} if and only if there exists an **invariant** $\mathcal{I} \subseteq \Sigma$ *such that*

1. *Every local run that only receives messages of $\mathcal{I}$ broadcasts only messages of $\mathcal{I}$*
2. *Every local run reaching q_{err} uses a reception* **rec**(m) *with $m \notin \mathcal{I}$*

$$\mathcal{R} \text{ is safe} \Leftrightarrow \exists \mathcal{I} \text{ s.t. } \mathcal{R} \text{ satisfies 1 and 2 w.r.t } \mathcal{I} .$$
$$\exists \sigma \text{ s.t. } \mathcal{G}_\sigma \text{ is safe} \Leftrightarrow \exists \sigma, \exists \mathcal{I} \text{ s.t. } \mathcal{G}_\sigma \text{ satisfies 1 and 2 w.r.t } \mathcal{I} .$$
$$\exists \sigma \text{ s.t. } \mathcal{G}_\sigma \text{ is safe} \Leftrightarrow \exists \mathcal{I} \text{ , } \exists \sigma \text{ s.t. } \mathcal{G}_\sigma \text{ satisfies 1 and 2 w.r.t } \mathcal{I} .$$

- For each invariant $\mathcal{I}$, check if there exists σ such that 1 and 2 are satisfied
  $\rightarrow$ Regular 2-player game.

Decidability

Lemma

There is no run covering q_{err} if and only if there exists an **invariant** $\mathcal{I} \subseteq \Sigma$ *such that*

1. *Every local run that only receives messages of $\mathcal{I}$ broadcasts only messages of $\mathcal{I}$*
2. *Every local run reaching q_{err} uses a reception* **rec**(m) *with $m \notin \mathcal{I}$*

$$\mathcal{R} \text{ is safe} \Leftrightarrow \exists \mathcal{I} \text{ s.t. } \mathcal{R} \text{ satisfies 1 and 2 w.r.t } \mathcal{I} .$$
$$\exists \sigma \text{ s.t. } \mathcal{G}_\sigma \text{ is safe} \Leftrightarrow \exists \sigma, \exists \mathcal{I} \text{ s.t. } \mathcal{G}_\sigma \text{ satisfies 1 and 2 w.r.t } \mathcal{I} .$$
$$\exists \sigma \text{ s.t. } \mathcal{G}_\sigma \text{ is safe} \Leftrightarrow \exists \mathcal{I} \text{ , } \exists \sigma \text{ s.t. } \mathcal{G}_\sigma \text{ satisfies 1 and 2 w.r.t } \mathcal{I} .$$

- For each invariant $\mathcal{I}$, check if there exists σ such that 1 and 2 are satisfied
  $\rightarrow$ Regular 2-player game.

Theorem

The safe strategy synthesis problem for RBG is NP-complete.




Part I.B

Broadcast Networks with Registers




Broadcast Networks of Register Automata (BNRA)[1]

Each process has *registers* $\square_1, \ldots, \square_r$, containing data in an infinite dataset $\mathbb{D}$.

[1] Delzanno, Sangnier, Traverso, RP'13

Broadcast Networks of Register Automata (BNRA)[1]

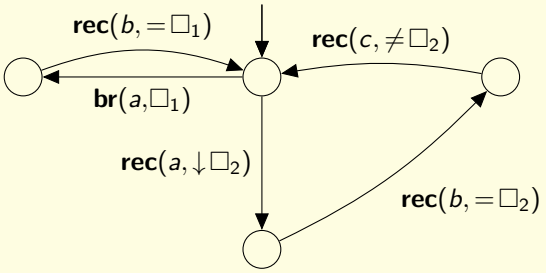
Each process has *registers* $\square_1, \ldots, \square_r$, containing data in an infinite dataset $\mathbb{D}$.
Initially, the registers of each process contain a unique datum.

[1] Delzanno, Sangnier, Traverso, RP'13

Broadcast Networks of Register Automata (BNRA)[1]

Each process has *registers* $\square_1, \ldots, \square_r$, containing data in an infinite dataset $\mathbb{D}$.
Initially, the registers of each process contain a unique datum.

Message = letter + datum: $(m, d) \in \Sigma \times \mathbb{D}$.
A process can:



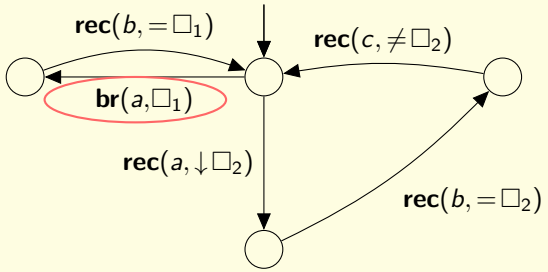
[1] Delzanno, Sangnier, Traverso, RP'13

Broadcast Networks of Register Automata (BNRA)[1]

Each process has *registers* $\square_1, \ldots, \square_r$, containing data in an infinite dataset $\mathbb{D}$.
Initially, the registers of each process contain a unique datum.

Message = letter + datum: $(m, d) \in \Sigma \times \mathbb{D}$.
A process can:

- Broadcast a message m with a register content



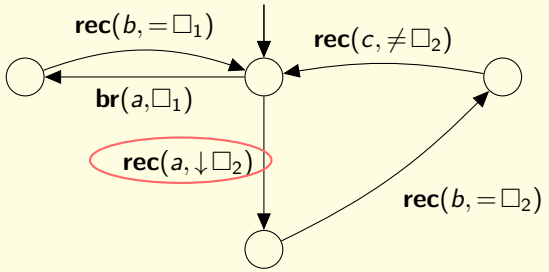
[1] Delzanno, Sangnier, Traverso, RP'13

Broadcast Networks of Register Automata (BNRA)[1]

Each process has *registers* $\square_1, \ldots, \square_r$, containing data in an infinite dataset $\mathbb{D}$.
Initially, the registers of each process contain a unique datum.

Message = letter + datum: $(m, d) \in \Sigma \times \mathbb{D}$.
A process can:

- Broadcast a message m with a register content
- Receive a message m while either
  - storing the attached datum $\downarrow$,



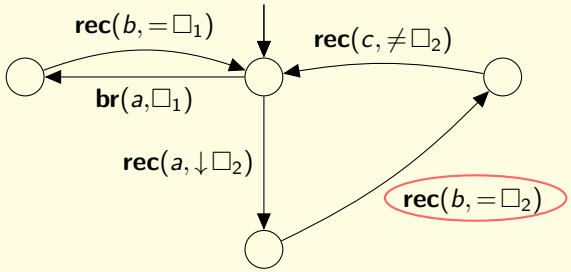
[1] Delzanno, Sangnier, Traverso, RP'13

Broadcast Networks of Register Automata (BNRA)[1]

Each process has *registers* $\square_1, \ldots, \square_r$, containing data in an infinite dataset $\mathbb{D}$.
Initially, the registers of each process contain a unique datum.

Message = letter + datum: $(m, d) \in \Sigma \times \mathbb{D}$.
A process can:

- Broadcast a message m with a register content
- Receive a message m while either
  - storing the attached datum $\downarrow$,
  - or testing it for equality $=, \neq$



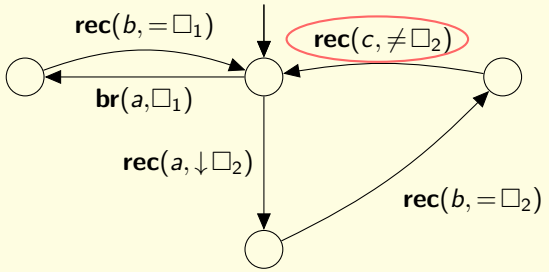
[1] Delzanno, Sangnier, Traverso, RP'13

Broadcast Networks of Register Automata (BNRA)[1]

Each process has *registers* $\square_1, \ldots, \square_r$, containing data in an infinite dataset $\mathbb{D}$.
Initially, the registers of each process contain a unique datum.

Message = letter + datum: $(m, d) \in \Sigma \times \mathbb{D}$.
A process can:

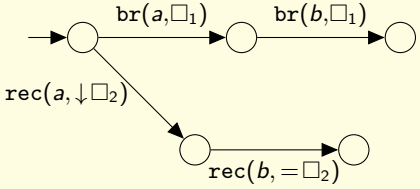
- Broadcast a message m with a register content
- Receive a message m while either
  - storing the attached datum $\downarrow$,
  - or testing it for equality $=, \neq$



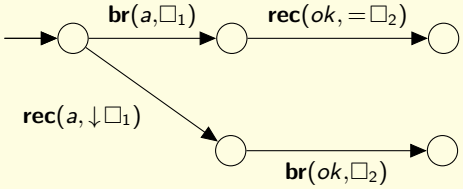
[1] Delzanno, Sangnier, Traverso, RP'13

Things we can do

Check that a sequence of messages all come from the same process.



Check that a sequence of messages we sent was received.



Invariants (in a subcase)

A word u is a *subword* of v if it can be obtained by removing letters from v.
$\triangleright$ *ababb* is a subword of *bbabaabbab*

Invariants (in a subcase)

A word u is a *subword* of v if it can be obtained by removing letters from v.
$\triangleright$ *ababb* is a subword of *bbabaabbab*

- Define run transformations to put them in a normal form.
- Use the lossy messages and the arbitrary number of agents.

Invariants (in a subcase)

A word u is a *subword* of v if it can be obtained by removing letters from v.
$\triangleright$ *ababb* is a subword of *bbabaabbab*

- Define run transformations to put them in a normal form.
- Use the lossy messages and the arbitrary number of agents.

Invariant = non-empty set $\mathcal{I} \subseteq \Sigma^*$ that is *closed under subwords*.

Invariants (in a subcase)

A word u is a *subword* of v if it can be obtained by removing letters from v.
$\triangleright$ *ababb* is a subword of *bbabaabbab*

- Define run transformations to put them in a normal form.
- Use the lossy messages and the arbitrary number of agents.

Invariant = non-empty set $\mathcal{I} \subseteq \Sigma^*$ that is *closed under subwords*.

Lemma

There is no run covering q_{err} if and only if there exists an **invariant** *$\mathcal{I} \subseteq \Sigma^*$ such that*

1. *Every local run that only receives words of $\mathcal{I}$ broadcasts only words of $\mathcal{I}$*
2. *Every local run reaching q_{err} receives a word $w \notin I$*

Broadcast / receive $w \in \Sigma^*$ = broadcast / receive each letter of w with the same datum

Invariants

Lemma

There is no run covering q_{err} if and only if there exists an **invariant** $\mathcal{I} \subseteq \Sigma^*$ *such that*

1. *Every local run that only receives words of $\mathcal{I}$ broadcasts only words of $\mathcal{I}$*
2. *Every local run reaching q_{err} receives a word $w \notin I$*

Invariants

Lemma

There is no run covering q_{err} if and only if there exists an **invariant** $\mathcal{I} \subseteq \Sigma^*$ *such that*

1. *Every local run that only receives words of $\mathcal{I}$ broadcasts only words of $\mathcal{I}$*
2. *Every local run reaching q_{err} receives a word $w \notin I$*

- Design appropriate game to check 1 and 2
- Well quasi-orders + regular game arguments to bound the size of $\mathcal{I}$

Invariants

Lemma

There is no run covering q_{err} if and only if there exists an **invariant** $\mathcal{I} \subseteq \Sigma^*$ *such that*

1. *Every local run that only receives words of $\mathcal{I}$ broadcasts only words of $\mathcal{I}$*
2. *Every local run reaching q_{err} receives a word $w \notin I$*

- Design appropriate game to check 1 and 2
- Well quasi-orders + regular game arguments to bound the size of $\mathcal{I}$

Theorem

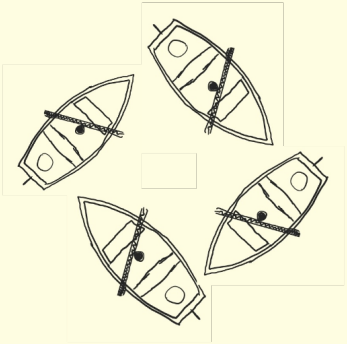
Safe strategy synthesis is decidable for broadcast networks with registers.

	0 registers	**1 registers**	**$\geq$ 2 registers**
Coverability	PTIME*	NP	$\mathbf{F}_{\omega^\omega}$
Safe strat.	NP	NEXPTIME	$\mathbf{F}_{\omega^\omega}$

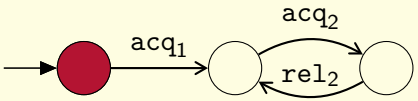
} [Guillou, M., Waldburger '23]

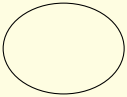
Part II

Locks

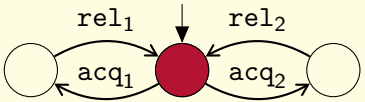


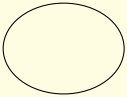
Lock-sharing systems[2]





P



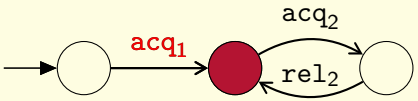


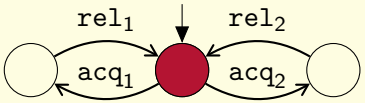
Q

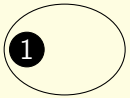
❶ ❷

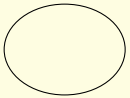
[2] Kahlon, Ivancic, Gupta CAV 2005

Lock-sharing systems[2]



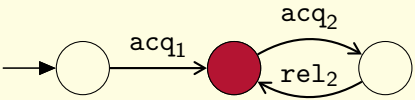


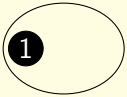




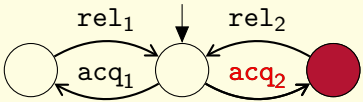
[2]Kahlon, Ivancic, Gupta CAV 2005

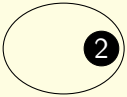
Lock-sharing systems[2]





P

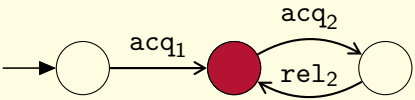


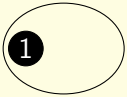


Q

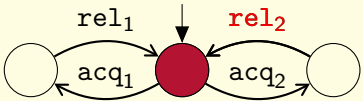
[2] Kahlon, Ivancic, Gupta CAV 2005

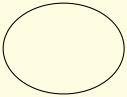
Lock-sharing systems[2]





P



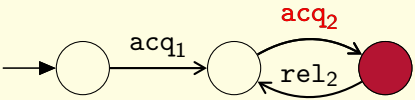


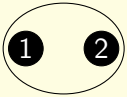
Q

2

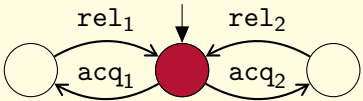
[2] Kahlon, Ivancic, Gupta CAV 2005

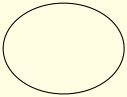
Lock-sharing systems[2]





P





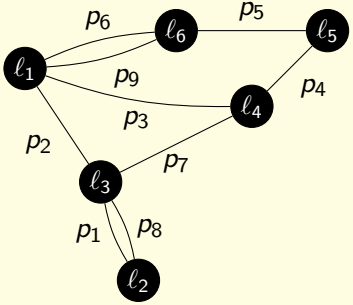
Q

[2] Kahlon, Ivancic, Gupta CAV 2005

Restrictions

Two locks/process

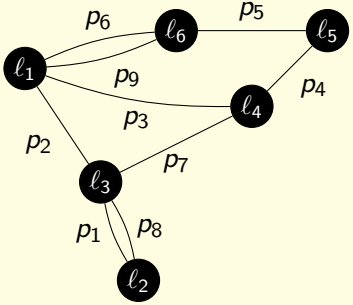
Each process uses at most two locks.
A system defines a *locking graph*.



Restrictions

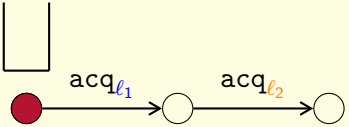
Two locks/process

Each process uses at most two locks.
A system defines a *locking graph*.



Nested locking

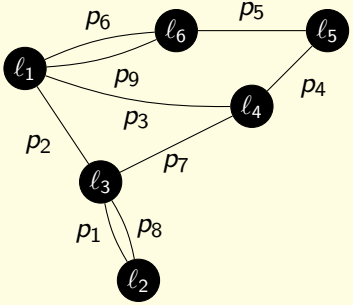
All processes acquire and release locks in a **stack-like order**, i.e., a process can only release the lock it acquired the latest.



Restrictions

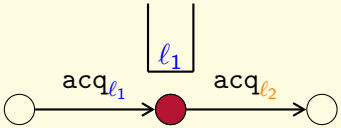
Two locks/process

Each process uses at most two locks.
A system defines a *locking graph*.



Nested locking

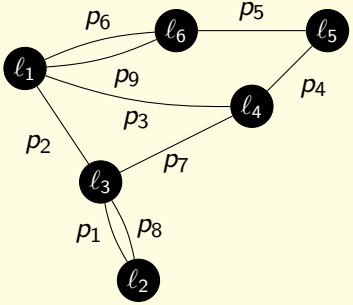
All processes acquire and release locks in a **stack-like order**, i.e., a process can only release the lock it acquired the latest.



Restrictions

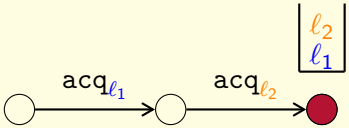
Two locks/process

Each process uses at most two locks.
A system defines a *locking graph*.



Nested locking

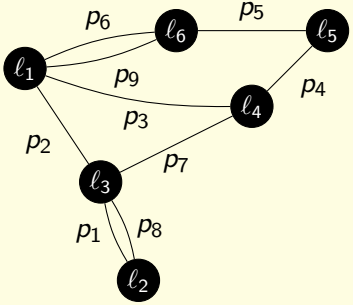
All processes acquire and release locks in a **stack-like order**, i.e., a process can only release the lock it acquired the latest.



Restrictions

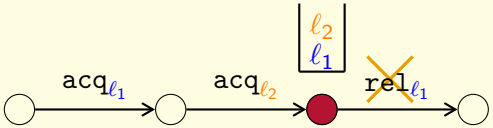
Two locks/process

Each process uses at most two locks.
A system defines a *locking graph*.



Nested locking

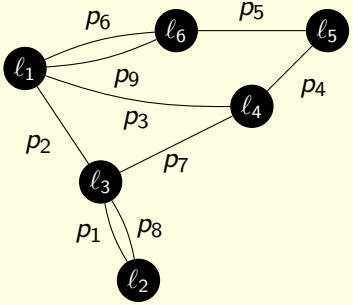
All processes acquire and release locks in a **stack-like order**, i.e., a process can only release the lock it acquired the latest.



Restrictions

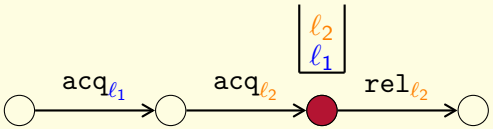
Two locks/process

Each process uses at most two locks.
A system defines a *locking graph*.



Nested locking

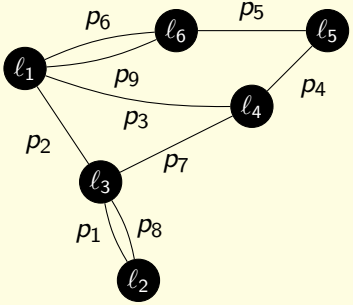
All processes acquire and release locks in a **stack-like order**, i.e., a process can only release the lock it acquired the latest.



Restrictions

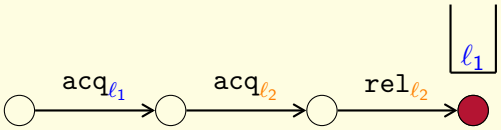
Two locks/process

Each process uses at most two locks.
A system defines a *locking graph*.



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.



Patterns

Each local run has a type, called a *pattern*.

Schedulability lemma

Given a family of local runs $(u_p)_{p \in Proc}$, whether we can form a valid global run by interleaving them only depends on their *patterns*.

Patterns

Each local run has a type, called a *pattern*.

Schedulability lemma

Given a family of local runs $(u_p)_{p \in Proc}$, whether we can form a valid global run by interleaving them only depends on their *patterns*.

Two locks/process

We colour the graph with the pattern of each run and analyse cycles.

Nested locking

Patterns $\sim$ partial orders on locks

Schedulability $\sim$ finding a common order

Patterns

Each local run has a type, called a *pattern*.

Schedulability lemma

Given a family of local runs $(u_p)_{p \in Proc}$, whether we can form a valid global run by interleaving them only depends on their *patterns*.

Two locks/process

We colour the graph with the pattern of each run and analyse cycles.

Nested locking

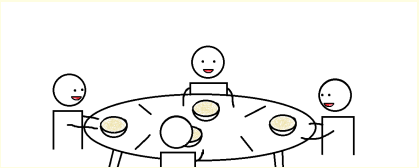
Patterns $\sim$ partial orders on locks

Schedulability $\sim$ finding a common order

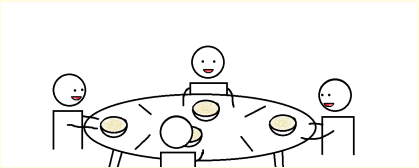
Synthesis $\sim$ find sets of patterns $(\Pi_p)_{p \in Proc}$ such that:

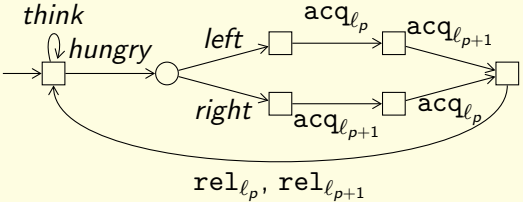
- No bad run can be obtained from local runs with those patterns
- For all p there is a local strategy σ_p such that all local runs of p are in Π_p

Example

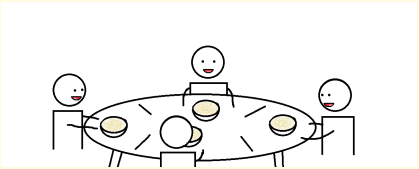


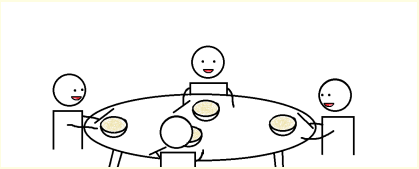
Example

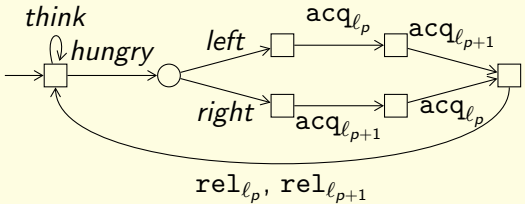



think
hungry
left
right
$\texttt{acq}_{\ell_p}$
$\texttt{acq}_{\ell_{p+1}}$
$\texttt{acq}_{\ell_{p+1}}$
$\texttt{acq}_{\ell_p}$
$\texttt{rel}_{\ell_p}, \texttt{rel}_{\ell_{p+1}}$

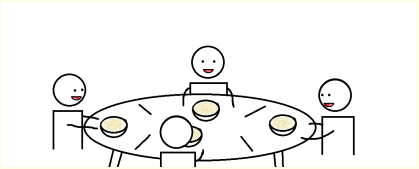
Example

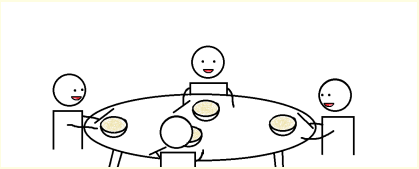


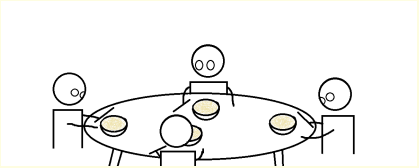


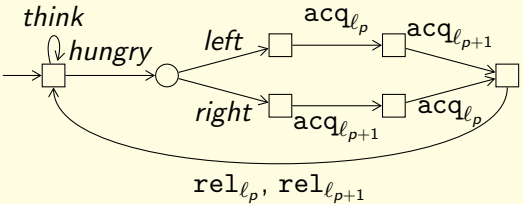

think
hungry
left
right
$\texttt{acq}_{\ell_p}$
$\texttt{acq}_{\ell_{p+1}}$
$\texttt{acq}_{\ell_{p+1}}$
$\texttt{acq}_{\ell_p}$
$\texttt{rel}_{\ell_p}, \texttt{rel}_{\ell_{p+1}}$

Example

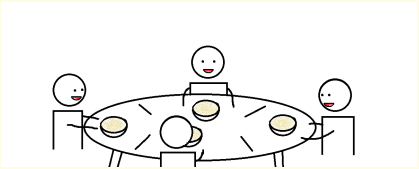


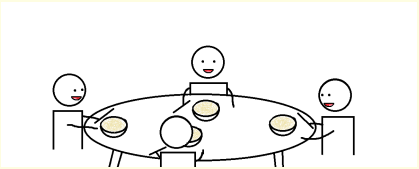


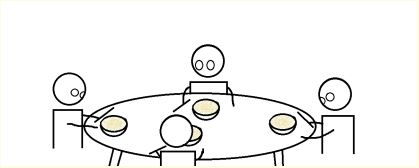


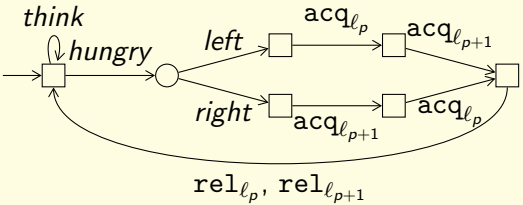

think
hungry
left
right
$\texttt{acq}_{\ell_p}$
$\texttt{acq}_{\ell_{p+1}}$
$\texttt{acq}_{\ell_{p+1}}$
$\texttt{acq}_{\ell_p}$
$\texttt{rel}_{\ell_p}, \texttt{rel}_{\ell_{p+1}}$

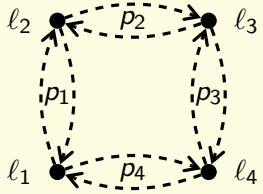
Example



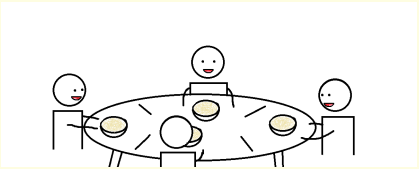


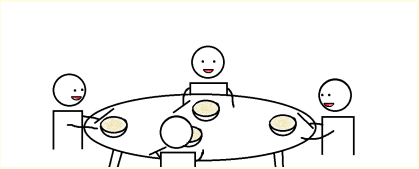


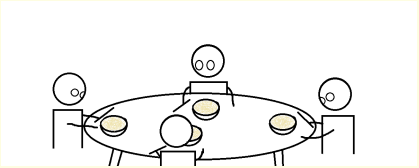

think
hungry
left
right
$\texttt{acq}_{\ell_p}$
$\texttt{acq}_{\ell_{p+1}}$
$\texttt{acq}_{\ell_{p+1}}$
$\texttt{acq}_{\ell_p}$
$\texttt{rel}_{\ell_p}, \texttt{rel}_{\ell_{p+1}}$

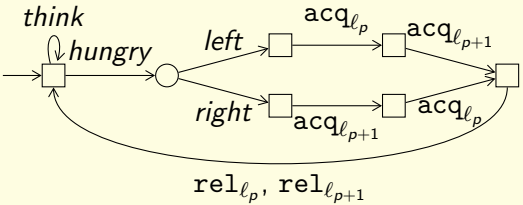

ℓ_2
p_2
ℓ_3
p_1
p_3
ℓ_1
p_4
ℓ_4

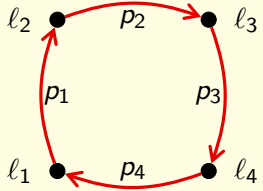
Example



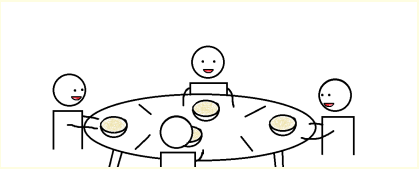


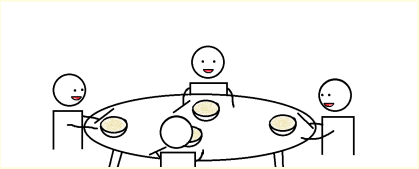


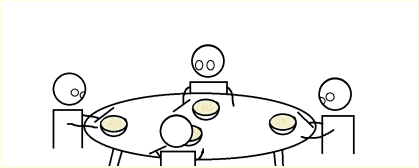

think
hungry
left
right
$\texttt{acq}_{\ell_p}$
$\texttt{acq}_{\ell_{p+1}}$
$\texttt{acq}_{\ell_{p+1}}$
$\texttt{acq}_{\ell_p}$
$\texttt{rel}_{\ell_p}, \texttt{rel}_{\ell_{p+1}}$

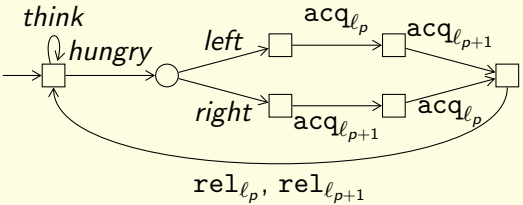

ℓ_2
p_2
ℓ_3
p_1
p_3
ℓ_1
p_4
ℓ_4

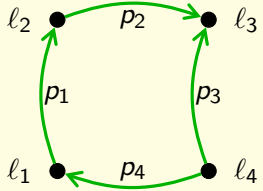
Example








think
hungry
left
right
$\texttt{acq}_{\ell_p}$
$\texttt{acq}_{\ell_{p+1}}$
$\texttt{acq}_{\ell_{p+1}}$
$\texttt{acq}_{\ell_p}$
$\texttt{rel}_{\ell_p}, \texttt{rel}_{\ell_{p+1}}$


ℓ_2
p_2
ℓ_3
p_1
p_3
ℓ_1
p_4
ℓ_4

Results

Objectives

Regular objective = Boolean combination of local regular conditions

Global deadlock = Every process is blocked

Process deadlock = Process p is blocked

Results

Objectives

Regular objective = Boolean combination of local regular conditions

Global deadlock = Every process is blocked

Process deadlock = Process p is blocked

Theorem

Synthesis is undecidable for lock-sharing systems.

Theorem

Synthesis for global deadlock avoidance is decidable in PTIME for locally live exclusive 2LSS.

Results

Objectives

Regular objective = Boolean combination of local regular conditions

Global deadlock = Every process is blocked

Process deadlock = Process p is blocked

Theorem

Synthesis is undecidable for lock-sharing systems.

Theorem

Synthesis for global deadlock avoidance is decidable in PTIME for locally live exclusive 2LSS.

Verification	**Reg. obj.**	**Global**	**Process**
LSS	PSPACE	PSPACE	PSPACE
Nested	NP	NP	NP
2LSS	NP	NP	NP
Locally live 2LSS	NP	PTIME	NP
Exclusive 2LSS	NP	NP	PTIME

Synthesis	**Reg. obj.**	**Global**	**Process**
LSS	Undec.	Undec.	Undec.
Nested	NEXP	Σ_2^P	Σ_2^P
2LSS	NEXP	Σ_2^P	Σ_2^P
Locally live 2LSS	NEXP	NP	Σ_2^P
Exclusive 2LSS	NEXP	Σ_2^P	NP

Results

Objectives

Regular objective = Boolean combination of local regular conditions

Global deadlock = Every process is blocked

Process deadlock = Process p is blocked

Theorem

Synthesis is undecidable for lock-sharing systems.

Theorem

Synthesis for global deadlock avoidance is decidable in PTIME for locally live exclusive 2LSS.

Verification	**Reg. obj.**	**Global**	**Process**
LSS	PSPACE	PSPACE	PSPACE
Nested	NP	NP	NP
2LSS	NP	NP	NP
Locally live 2LSS	NP	PTIME	NP
Exclusive 2LSS	NP	NP	PTIME

Synthesis	**Reg. obj.**	**Global**	**Process**
LSS	Undec.	Undec.	Undec.
Nested	NEXP	Σ_2^P	Σ_2^P
2LSS	NEXP	Σ_2^P	Σ_2^P
Locally live 2LSS	NEXP	NP	Σ_2^P
Exclusive 2LSS	NEXP	Σ_2^P	NP

Results

Objectives

Regular objective = Boolean combination of local regular conditions

Global deadlock = Every process is blocked

Process deadlock = Process p is blocked

Theorem

Synthesis is undecidable for lock-sharing systems.

Theorem

Synthesis for global deadlock avoidance is decidable in PTIME for locally live exclusive 2LSS.

Verification	Reg. obj.	Global	Process
LSS	PSPACE	PSPACE	PSPACE
Nested	NP	NP	NP
2LSS	NP	NP	NP
Locally live 2LSS	NP	PTIME	NP
Exclusive 2LSS	NP	NP	PTIME

Synthesis	Reg. obj.	Global	Process
LSS	Undec.	Undec.	Undec.
Nested	NEXP	Σ_2^P	Σ_2^P
2LSS	NEXP	Σ_2^P	Σ_2^P
Locally live 2LSS	NEXP	NP	Σ_2^P
Exclusive 2LSS	NEXP	Σ_2^P	NP

Results

Objectives

Regular objective = Boolean combination of local regular conditions

Global deadlock = Every process is blocked

Process deadlock = Process p is blocked

Theorem

Synthesis is undecidable for lock-sharing systems.

Theorem

Synthesis for global deadlock avoidance is decidable in PTIME for locally live exclusive 2LSS.

Verification	**Reg. obj.**	**Global**	**Process**
LSS	PSPACE	PSPACE	PSPACE
Nested	NP	NP	NP
2LSS	NP	NP	NP
Locally live 2LSS	NP	PTIME	NP
Exclusive 2LSS	NP	NP	PTIME

Synthesis	**Reg. obj.**	**Global**	**Process**
LSS	Undec.	Undec.	Undec.
Nested	NEXP	Σ_2^P	Σ_2^P
2LSS	NEXP	Σ_2^P	Σ_2^P
Locally live 2LSS	NEXP	NP	Σ_2^P
Exclusive 2LSS	NEXP	Σ_2^P	NP

Results

Objectives

Regular objective = Boolean combination of local regular conditions

Global deadlock = Every process is blocked

Process deadlock = Process p is blocked

Theorem

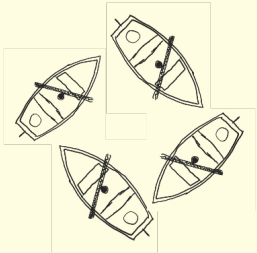
Synthesis is undecidable for lock-sharing systems.

Theorem

Synthesis for global deadlock avoidance is decidable in PTIME for locally live exclusive 2LSS.

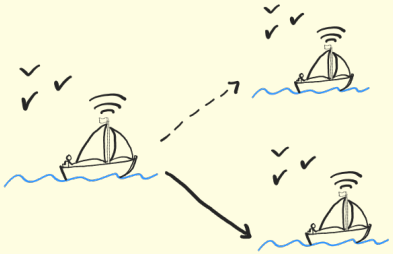
Verification	**Reg. obj.**	**Global**	**Process**
LSS	PSPACE	PSPACE	PSPACE
Nested	NP	NP	NP
2LSS	NP	NP	NP
Locally live 2LSS	NP	PTIME	NP
Exclusive 2LSS	NP	NP	PTIME

Synthesis	**Reg. obj.**	**Global**	**Process**
LSS	Undec.	Undec.	Undec.
Nested	NEXP	Σ_2^P	Σ_2^P
2LSS	NEXP	Σ_2^P	Σ_2^P
Locally live 2LSS	NEXP	NP	Σ_2^P
Exclusive 2LSS	NEXP	Σ_2^P	NP



Part III

Locks and process spawning



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.

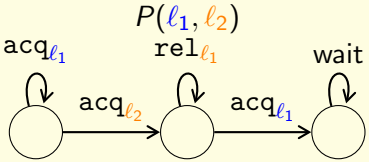
▷ A process can spawn other processes

▷ A process takes parameters, represented by *lock variables*

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

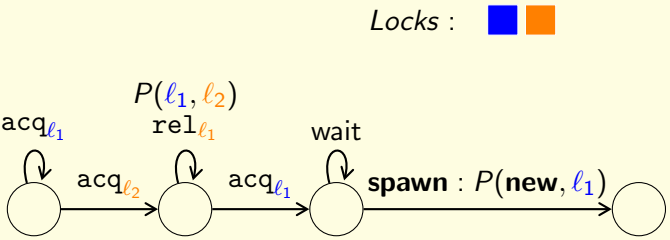
Dynamic LSS[3]

Locks : 



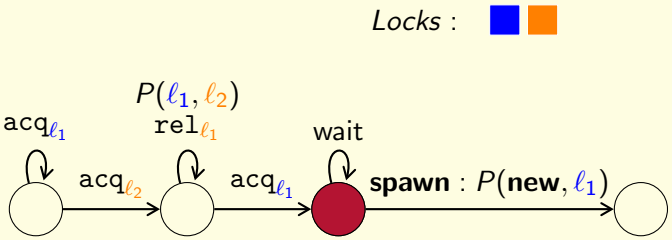
[3] Bouajjani, Müller-Olm, Touili, CONCUR 2005 + Kenter's thesis 2022

Dynamic LSS[3]



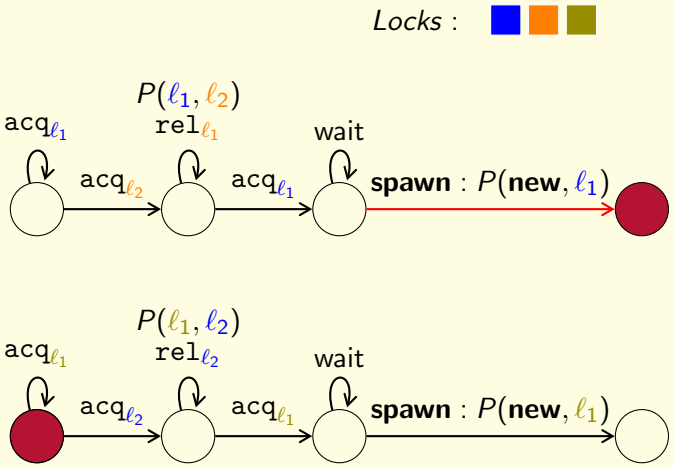
[3] Bouajjani, Müller-Olm, Touili, CONCUR 2005 + Kenter's thesis 2022

Dynamic LSS[3]



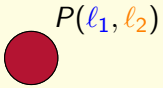
[3] Bouajjani, Müller-Olm, Touili, CONCUR 2005 + Kenter's thesis 2022

Dynamic LSS[3]

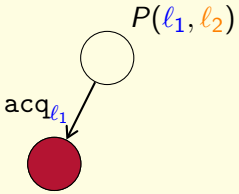


[3] Bouajjani, Müller-Olm, Touili, CONCUR 2005 + Kenter's thesis 2022

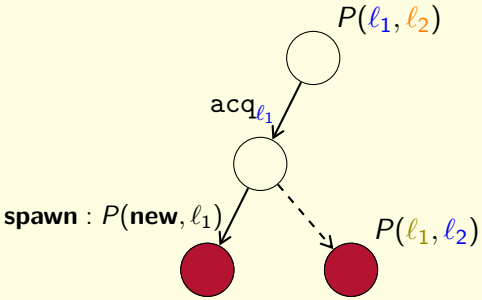
Tree representation



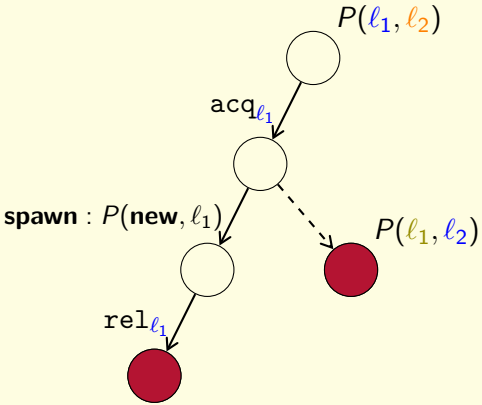
Tree representation



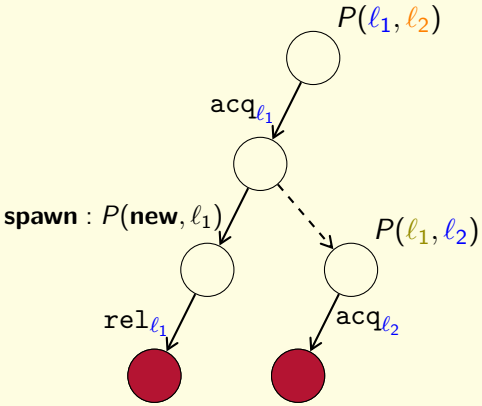
Tree representation



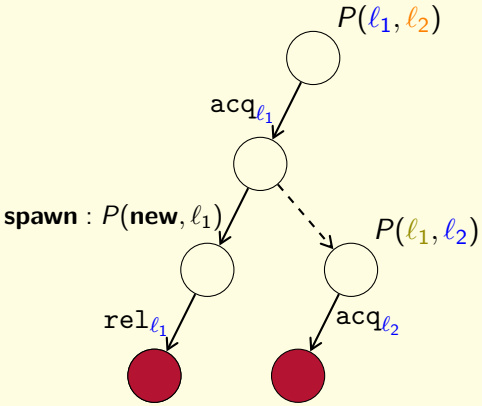
Tree representation



Tree representation

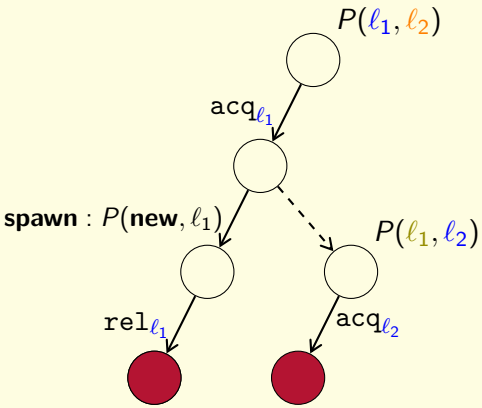


Tree representation



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

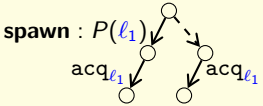
Input: A DLSS $\mathcal{D}$ and a parity tree automaton $\mathcal{A}$.
Output: Is there a run of $\mathcal{D}$ accepted by $\mathcal{A}$?

Problem: characterise trees that represent actual executions.

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 $\mathcal{A}$?

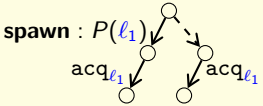
Problem: characterise trees that represent actual executions.

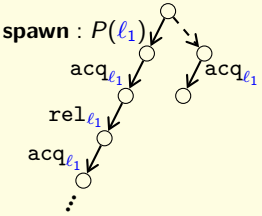


Regular model-checking problem

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.

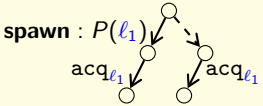


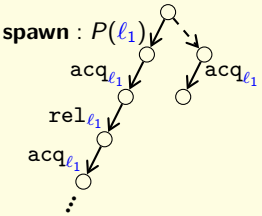


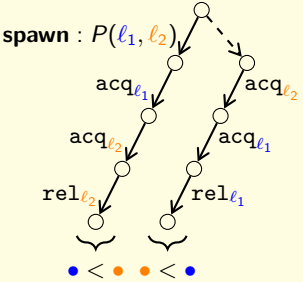
Regular model-checking problem

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.



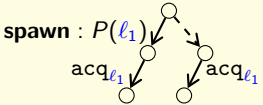


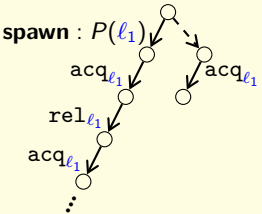


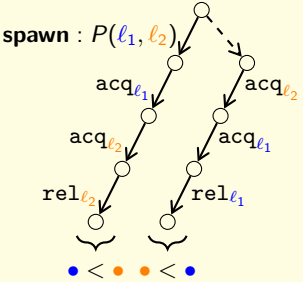
Regular model-checking problem

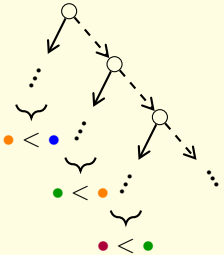
Input: A DLSS $\mathcal{D}$ and a parity tree automaton $\mathcal{A}$.
Output: Is there a run of $\mathcal{D}$ accepted by $\mathcal{A}$?

Problem: characterise trees that represent actual executions.





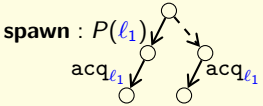


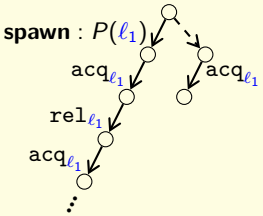


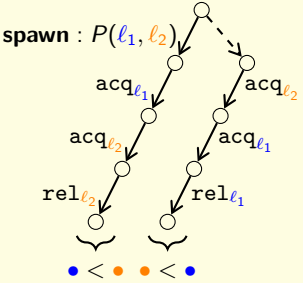
Regular model-checking problem

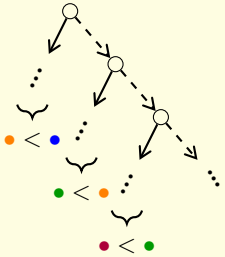
Input: A DLSS $\mathcal{D}$ and a parity tree automaton $\mathcal{A}$.
Output: Is there a run of $\mathcal{D}$ accepted by $\mathcal{A}$?

Problem: characterise trees that represent actual executions.









Theorem

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

Theorem

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

Theorem

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

What about pushdown processes?

Theorem

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

What about pushdown processes?

Theorem

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

Theorem

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

What about pushdown processes?

Theorem

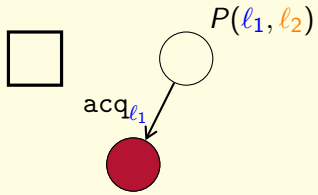
Regular model-checking of nested **pushdown** DLSS is EXPTIME-complete, and PTIME for fixed number of locks per process and parity index.

What about shared variables?

DLSS with variables

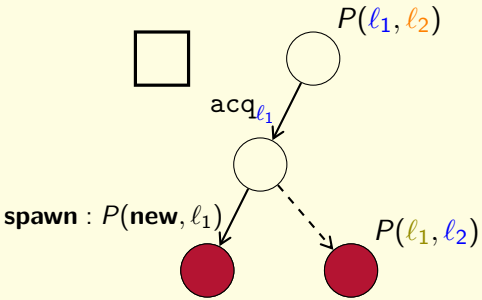
We add a variable and operations `wr` and `rd` writing and reading letters from a finite alphabet in the register.

DLSS with variables



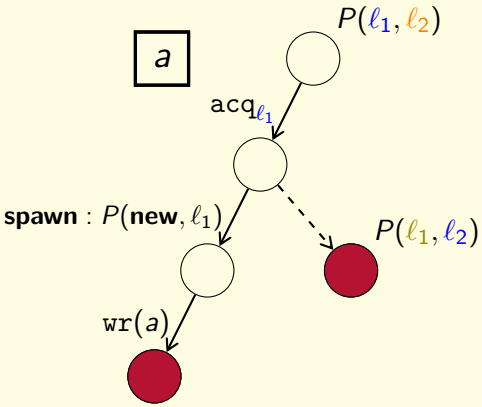
We add a variable and operations `wr` and `rd` writing and reading letters from a finite alphabet in the register.

DLSS with variables



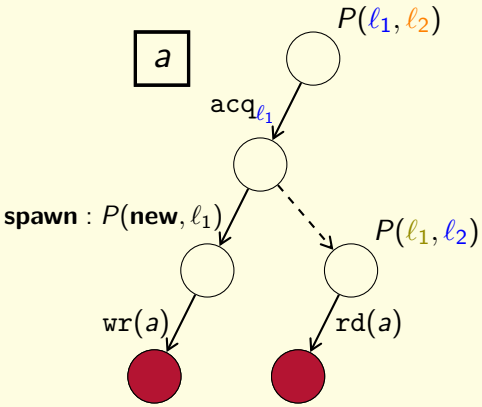
We add a variable and operations `wr` and `rd` writing and reading letters from a finite alphabet in the register.

DLSS with variables



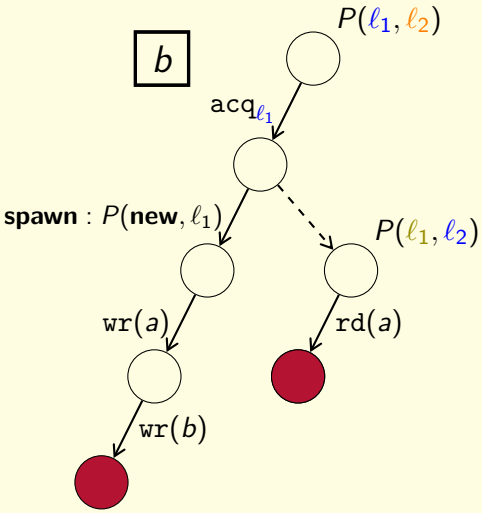
We add a variable and operations `wr` and `rd` writing and reading letters from a finite alphabet in the register.

DLSS with variables



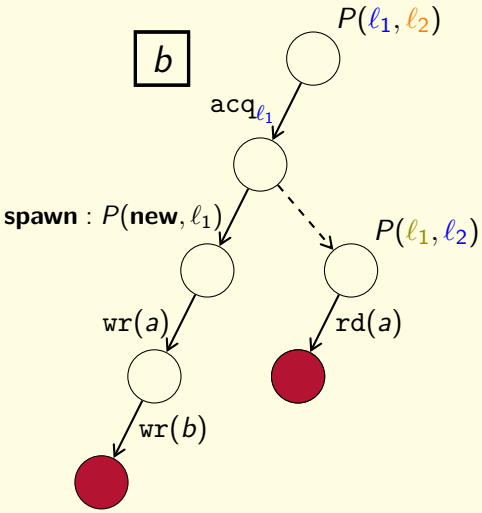
We add a variable and operations `wr` and `rd` writing and reading letters from a finite alphabet in the register.

DLSS with variables



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

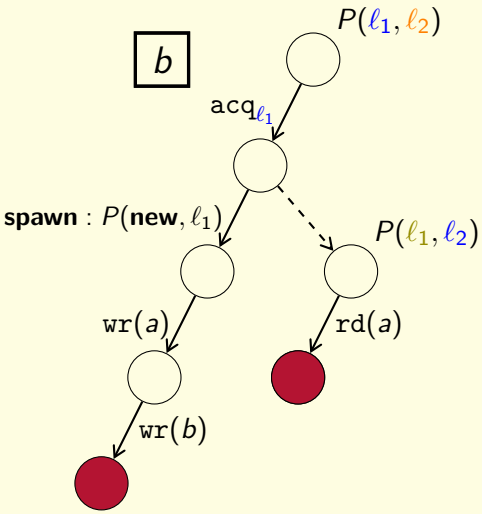
DLSS with variables



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

Sets of execution trees are **not** regular.

DLSS with variables



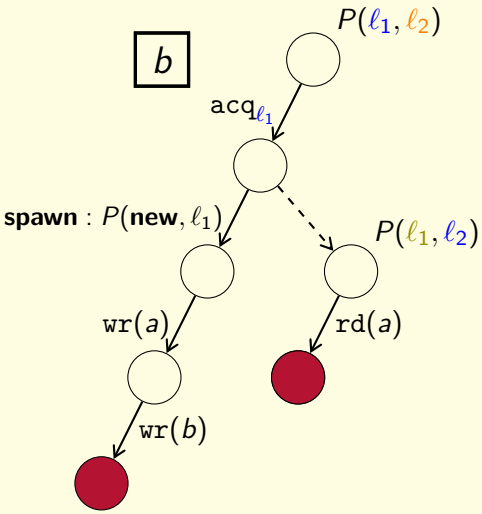
We add a variable and operations `wr` and `rd` writing and reading letters from a finite alphabet in the register.

Sets of execution trees are **not** regular.

Theorem

State reachability is undecidable for DLSS with variables.

DLSS with variables



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

Sets of execution trees are **not** regular.

Theorem

State reachability is undecidable for DLSS with variables.

Conjecture

Verification is decidable when the number of time the writing process switches is bounded.

Synthesis

- Processes have controllable and uncontrollable states
- Local strategies
- Every copy of each process uses the same strategy

Synthesis

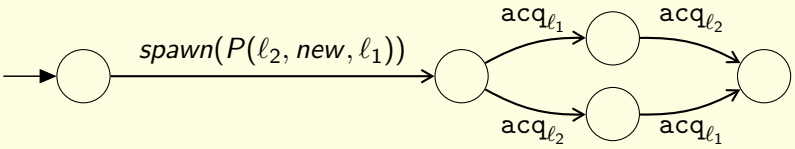
- Processes have controllable and uncontrollable states
- Local strategies
- Every copy of each process uses the same strategy

Problem

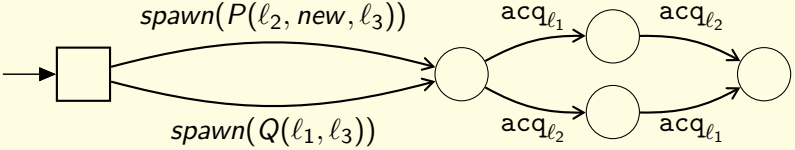
Is there a strategy $\sigma = (\sigma_p)_{p \in Proc}$ ensuring that there is no execution accepted by $\mathcal{A}$?

Example: Arbitrarily many dining philosophers

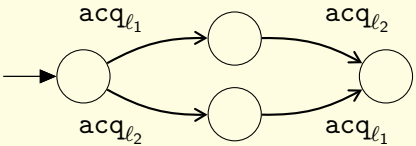
$P_{init}(\ell_1, \ell_2)$:

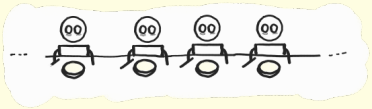


$P(\ell_1, \ell_2, \ell_3)$:



$Q(\ell_1, \ell_2)$:





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.

- Characterise sets of local runs with which we can form an execution of $\mathcal{D}$ accepted by $\mathcal{A}$.
- Check the existence of a strategy σ such that σ-local runs cannot form a tree accepted by $\mathcal{T}$, using a regular 2-player game.

Synthesis

Theorem

The controller synthesis problem is decidable in 2EXPTIME over DLSS.

Conclusion

Past

- New approach to distributed synthesis, applicable to models with
  - Fixed architectures
  - Parameterised number of processes
  - Dynamic set of processes
- Invariants for communication by locks, lossy broadcasts and shared variables.

Conclusion

Past

- New approach to distributed synthesis, applicable to models with
  - Fixed architectures
  - Parameterised number of processes
  - Dynamic set of processes
- Invariants for communication by locks, lossy broadcasts and shared variables.

Future

- Add variables to the models with locks
- Implementation of deadlock detection using SAT solvers
- Connection between BNRA and population protocols with data

Conclusion

Past

- New approach to distributed synthesis, applicable to models with
  - Fixed architectures
  - Parameterised number of processes
  - Dynamic set of processes
- Invariants for communication by locks, lossy broadcasts and shared variables.

Future

- Add variables to the models with locks
- Implementation of deadlock detection using SAT solvers
- Connection between BNRA and population protocols with data

Thank you for your attention!