Formal Proofs for Mobile Robot Swarms

Lionel Rieg

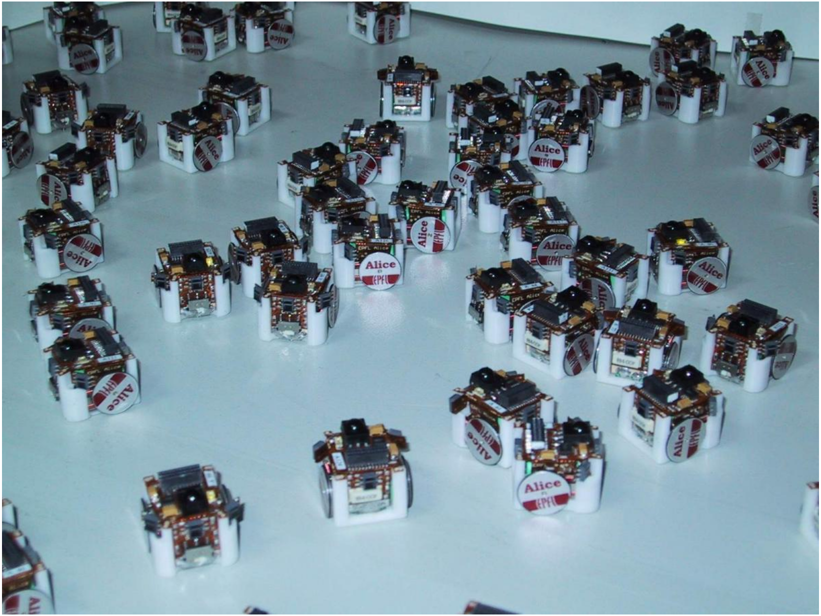
ENS de Lyon, 23 Oct. 2019

Outline

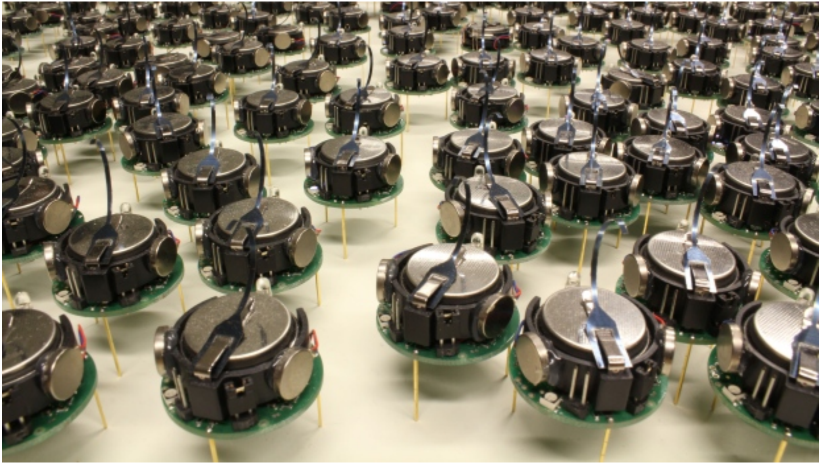
Inspiration: Swarms of Mobile Robots

- Swarms?

Inspiration: Swarms of Mobile Robots



Inspiration: Swarms of Mobile Robots



Inspiration: Swarms of Mobile Robots

- Swarms?
  - Lots of (small) identical robots

Inspiration: Swarms of Mobile Robots

- Swarms?
  - Lots of (small) identical robots
- Where?

Inspiration: Swarms of Mobile Robots

Inspiration: Swarms of Mobile Robots

Inspiration: Swarms of Mobile Robots

- Swarms?
  - Lots of (small) identical robots
- Where?
  - Entertainment
  - Rescue
  - Exploration
  - ...

Inspiration: Swarms of Mobile Robots

- Swarms?
  - Lots of (small) identical robots
- Where?
  - Entertainment
  - Rescue
  - Exploration
  - ...
- Opportunities?
  - Cooperative behavior (swarm intelligence)
  - Resilience

Inspiration: Swarms of Mobile Robots

- Swarms?
  - Lots of (small) identical robots
- Where?
  - Entertainment
  - Rescue
  - Exploration
  - ...
- Opportunities?
  - Cooperative behavior (swarm intelligence)
  - Resilience
- Main challenge?
  - Understand what happens!

Why Formal Methods for Mobile Robots Swarms?

Many mobile robots network models:

- Space — discrete/continuous, bounded/unbounded, topology, ...
- Sensors — multiplicity, range, accuracy, orientation, ...
- Faults — none, crash, Byzantine, ...
- Execution — synchronous/asynchronous, fairness, interruption, ...

Why Formal Methods for Mobile Robots Swarms?

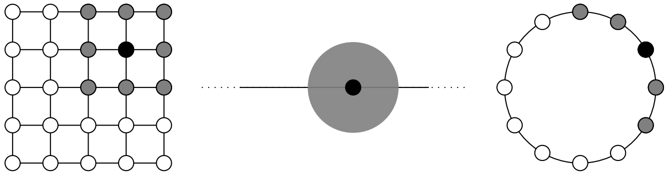
Many mobile robots network models:

- **Space** discrete/continuous, bounded/unbounded, topology, ...
- **Sensors** multiplicity, range, accuracy, orientation, ...
- **Faults** none, crash, Byzantine, ...
- **Execution** synchronous/asynchronous, fairness, interruption, ...

Why Formal Methods for Mobile Robots Swarms?

Many mobile robots network models:

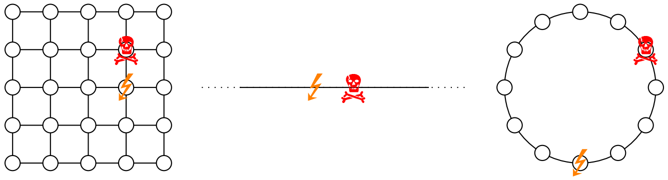
- Space — discrete/continuous, bounded/unbounded, topology, ...
- Sensors — multiplicity, range, accuracy, orientation, ...
- Faults — none, crash, Byzantine, ...
- Execution — synchronous/asynchronous, fairness, interruption, ...



Why Formal Methods for Mobile Robots Swarms?

Many mobile robots network models:

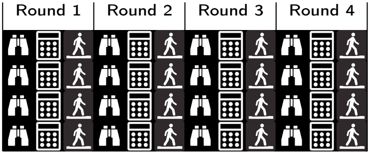
- Space — discrete/continuous, bounded/unbounded, topology, ...
- Sensors — multiplicity, range, accuracy, orientation, ...
- Faults — none, crash, Byzantine, ...
- Execution — synchronous/asynchronous, fairness, interruption, ...

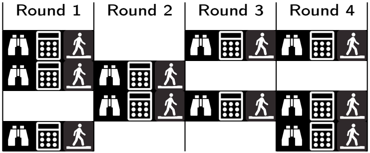


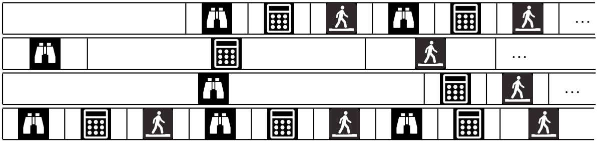
Why Formal Methods for Mobile Robots Swarms?

Many mobile robots network models:

- Space discrete/continuous, bounded/unbounded, topology, . . .
- Sensors multiplicity, range, accuracy, orientation, . . .
- Faults none, crash, Byzantine, . . .
- Execution synchronous/asynchronous, fairness, interruption, . . .







Why Formal Methods for Mobile Robots Swarms?

Many mobile robots network models:

- Space — discrete/continuous, bounded/unbounded, topology, ...
- Sensors — multiplicity, range, accuracy, orientation, ...
- Faults — none, crash, Byzantine, ...
- Execution — synchronous/asynchronous, fairness, interruption, ...

⚠

- Subtle differences in models $\rightsquigarrow$ very error-prone
- Careful of mismatch spec/proof!
- Lots of proof cases — geometry

$\Rightarrow$ Formal methods can help

Why Formal Methods for Mobile Robots Swarms?

Many mobile robots network models:

- Space — discrete/continuous, bounded/unbounded, topology, ...
- Sensors — multiplicity, range, accuracy, orientation, ...
- Faults — none, crash, Byzantine, ...
- Execution — synchronous/asynchronous, fairness, interruption, ...

⚠
- Subtle differences in models $\rightsquigarrow$ very error-prone
- Careful of mismatch spec/proof!
- Lots of proof cases — geometry

$\Rightarrow$ Formal methods can help

Which model? (process algebra, TLA, ...)
Which tool? (model checking, proof assistant, ...)

We need a suitable framework — What are the requirements?

Outline

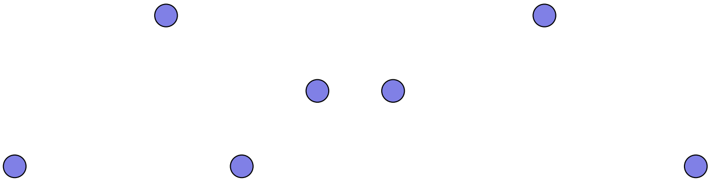
Robot Model

[Suzuki, Yamashita 99]

Robot Model

[Suzuki, Yamashita 99]

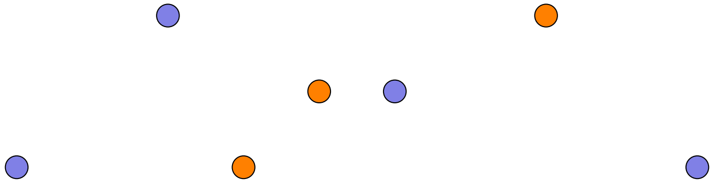
- Points



Robot Model

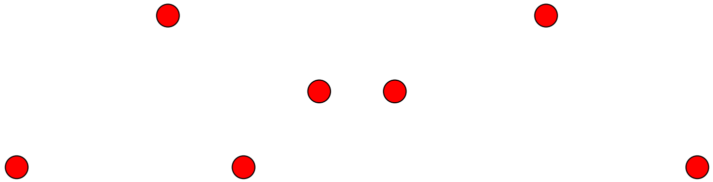
[Suzuki, Yamashita 99]

- Points
- With Byzantine faults (or crash)



Robot Model [Suzuki, Yamashita 99]

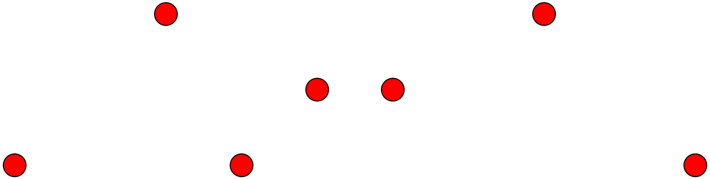
- Points
- With Byzantine faults (or crash)
- Anonymous



Robot Model

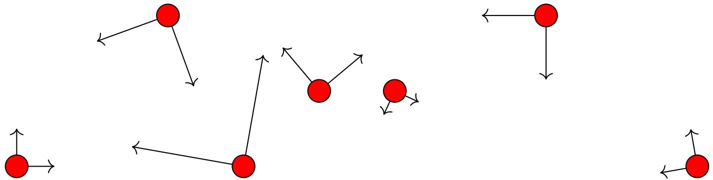
[Suzuki, Yamashita 99]

- Points
- With Byzantine faults (or crash)
- Anonymous
- No direct communication



Robot Model [Suzuki, Yamashita 99]

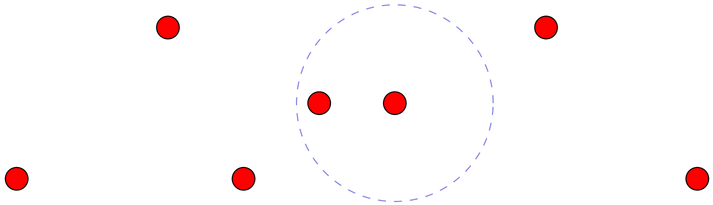
- Points
- With Byzantine faults (or crash)
- Anonymous
- No direct communication
- No common frame/direction
  $\rightsquigarrow$ local coordinates



Robot Model

[Suzuki, Yamashita 99]

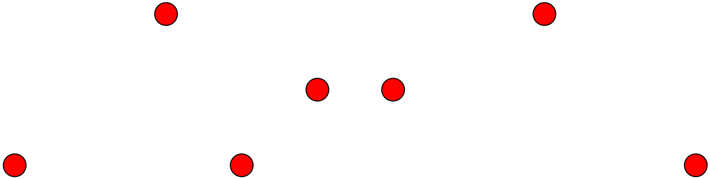
- Points
- With Byzantine faults (or crash)
- Anonymous
- No direct communication
- No common frame/direction
  $\rightsquigarrow$ local coordinates
- Limited/unlimited vision? multiplicity?



Robot Model

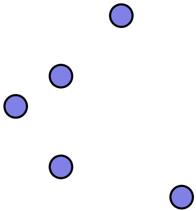
[Suzuki, Yamashita 99]

- Points
- With Byzantine faults (or crash)
- Anonymous
- No direct communication
- No common frame/direction
  $\rightsquigarrow$ local coordinates
- Limited/unlimited vision? multiplicity?
- Same (deterministic) program everywhere



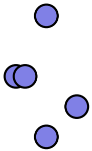
Example: Gathering

- Setting: $\mathbb{R}^2$, no Byzantine
- Objective: Have all robots reach in finite time the same location (unknown ahead of time) and then stay there



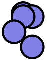
Example: Gathering

- Setting: $\mathbb{R}^2$, no Byzantine
- Objective: Have all robots reach in finite time the same location (unknown ahead of time) and then stay there



Example: Gathering

- Setting: $\mathbb{R}^2$, no Byzantine
- Objective: Have all robots reach in finite time the same location (unknown ahead of time) and then stay there

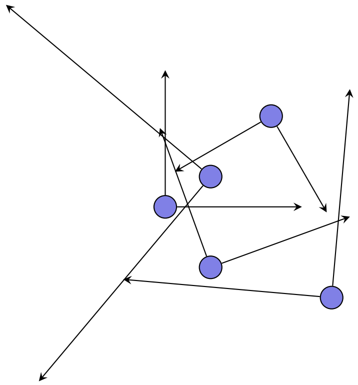


Example: Gathering

- Setting: $\mathbb{R}^2$, no Byzantine
- Objective: Have all robots reach in finite time the same location (unknown ahead of time) and then stay there

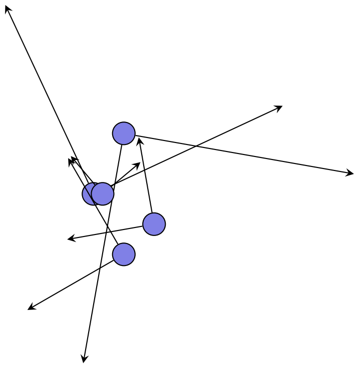
Example: Gathering

- Setting: $\mathbb{R}^2$, no Byzantine
- Objective: Have all robots reach in finite time the same location (unknown ahead of time) and then stay there



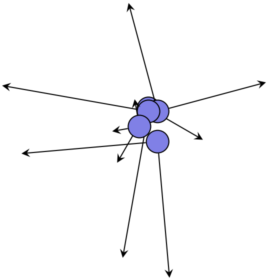
Example: Gathering

- Setting: $\mathbb{R}^2$, no Byzantine
- Objective: Have all robots reach in finite time the same location (unknown ahead of time) and then stay there



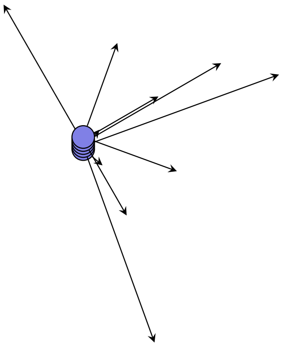
Example: Gathering

- Setting: $\mathbb{R}^2$, no Byzantine
- Objective: Have all robots reach in finite time the same location (unknown ahead of time) and then stay there



Example: Gathering

- Setting: $\mathbb{R}^2$, no Byzantine
- Objective: Have all robots reach in finite time the same location (unknown ahead of time) and then stay there



Execution Models (1) [Suzuki, Yamashita 99]

3 phases for each robot:

1. Look: observe its surrounding
   - Indirect communication
   - Depends on sensor capabilities
2. Compute: choose what to do
   - Choose an objective
   - Depends on observation, program
3. Move: do it (or try to)
   - Try to reach your target
   - Depends on the environment

Execution Models (1) [Suzuki, Yamashita 99]

3 phases for each robot:

1. Look: observe its surrounding
   - Indirect communication
   - Depends on sensor capabilities
2. Compute: choose what to do
   - Choose an objective
   - Depends on observation, program
3. Move: do it (or try to)
   - Try to reach your target
   - Depends on the environment

and repeat

Execution Models (2): Scheduling [Suzuki, Yamashita 99]

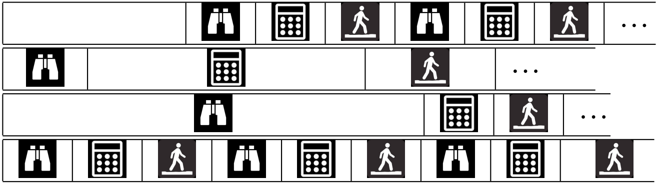
Scheduling of robots is

Execution Models (2): Scheduling [Suzuki, Yamashita 99]

Scheduling of robots is

Either ASYNC: full interleaving

- Most general/realistic but hardest



Execution Models (2): Scheduling [Suzuki, Yamashita 99]

Scheduling of robots is

Either ASYNC: full interleaving

- Most general/realistic but hardest

or Same phase for all active robots

- Time split into rounds

Round 1	Round 2	Round 3	Round 4	Round 5

Execution Models (2): Scheduling [Suzuki, Yamashita 99]

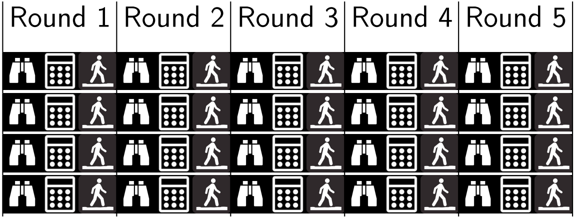
Scheduling of robots is

Either ASYNC: full interleaving

- Most general/realistic but hardest

or Same phase for all active robots

- Time split into rounds
- FSYNC: all robots are activated each round



Execution Models (2): Scheduling [Suzuki, Yamashita 99]

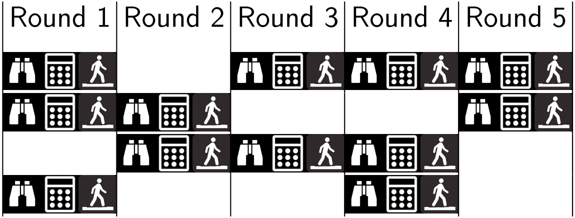
Scheduling of robots is

Either ASYNC: full interleaving

- Most general/realistic but hardest

or Same phase for all active robots

- Time split into rounds
- FSYNC: all robots are activated each round
- SSYNC: only a subset is activated
  ⇝ Fairness assumptions on the scheduling (demon)



The Rest of the Vocabulary

- Robogram: robot program
- Demon/scheduler: environment (adversary part)
  a sequence of demonic actions (one for each round)
- Configuration: the states of all robots
  (includes locations and ids)
  $\rightsquigarrow$ full snapshot of the system
- Observation: information available to robograms
  (depends on sensors, no identifers)
  $\rightsquigarrow$ degraded form of configuration
- Execution: a sequence of configurations
  (usually given by the robogram and the demon)

Outline

Coq: a Proof Assistant

Definition (Proof Assistant (Wikipedia))

In computer science and mathematical logic, a proof assistant or interactive theorem prover is a software tool to assist with the development of formal proofs by human-machine collaboration. This involves some sort of interactive proof editor, or other interface, with which a human can guide the search for proofs, the details of which are stored in, and some steps provided by, a computer.

The Coq proof assistant:

- 4-color theorem, Feit-Thomson theorem, CompCert compiler
- Functional programming language
- Proof = program — Curry-Howard
- Build programs/proofs with tactics (reasoning steps)
- ... or just program them!

Pactole: a Coq Framework for Mobile Robots

Very Parametric (but still useful):

- Space
- State of Robots (memory, battery level, etc.)
- Sensors
- Environment (adversary)
- How states are updated during the move phase

Main Ingredients:

- Robogram
- Demon (scheduler)
- Round
- Execution
- Properties and Proofs

Pactole: a Coq Framework for Mobile Robots

Main Ingredients:

- Robogram
- Demon (scheduler)
- Round
- Execution
- Properties and Proofs

Defining Robograms

A robogram is simply a function:

```
Definition robogram := observation → location .
```

- We can use all the expressiveness of Coq to define robograms.
- They can be extracted to OCaml/Haskell.
- We should only use geometric shapes that are invariant by change of frame of reference.

Defining Robograms

A robogram is simply a function:

```
Definition robogram := observation → location .
```

- We can use all the expressiveness of Coq to define robograms.
- They can be extracted to OCaml/Haskell.
- We should only use geometric shapes that are invariant by change of frame of reference.

Example (Convergence in $\mathbb{R}^2$)

```
(* observation = set of inhabited location *)
Definition convergeR2_pgm (obs : observation) : R2 :=
  barycenter (elements s).
```

Defining Robograms

A robogram is simply a function:

```
Definition robogram := observation → location .
```

- We can use all the expressiveness of Coq to define robograms.
- They can be extracted to OCaml/Haskell.
- We should only use geometric shapes that are invariant by change of frame of reference.

+ a technical detail: compatibility with equivalence (Proper)

Example (Convergence in $\mathbb{R}^2$)

```
(* observation = set of inhabited location *)
Definition convergeR2_pgm (obs : observation) : R2 :=
  barycenter (elements s).
```

Pactole: a Coq Framework for Mobile Robots

Main Ingredients:

- Robogram
- Demon (scheduler)
- Round
- Execution
- Properties and Proofs

observation → location

Description of a Round

What happens in a round for a robot?

Demonic action: what does the demon decide in each round?

Description of a Round

What happens in a round for a robot?

1. If it is not activated, some update happens ASYNC only

Demonic action: what does the demon decide in each round?

Description of a Round

What happens in a round for a robot?

1. If it is not activated, some update happens ASYNC only

Demonic action: what does the demon decide in each round?

1. Pick which robots are activated
2. Decide how to update inactive robots ASYNC only

Description of a Round

What happens in a round for a robot?

1. If it is not activated, some update happens — ASYNC only
2. If it is activated and Byzantine, the demon gives its new state

Demonic action: what does the demon decide in each round?

1. Pick which robots are activated
2. Decide how to update inactive robots — ASYNC only

Description of a Round

What happens in a round for a robot?

1. If it is not activated, some update happens — ASYNC only
2. If it is activated and Byzantine, the demon gives its new state

Demonic action: what does the demon decide in each round?

1. Pick which robots are activated
2. Decide how to update inactive robots — ASYNC only
3. Update Byzantine robots as it wishes

Description of a Round

What happens in a round for a robot?

1. If it is not activated, some update happens ASYNC only
2. If it is activated and Byzantine, the demon gives its new state
3. If it is activated and not Byzantine (i.e. Good),
   a. **Look**: get information from its surrounding
   b. **Compute** its destination
   c. **Move** to the destination

Demonic action: what does the demon decide in each round?

1. Pick which robots are activated
2. Decide how to update inactive robots ASYNC only
3. Update Byzantine robots as it wishes

Description of a Round

What happens in a round for a robot?

1. If it is not activated, some update happens ASYNC only
2. If it is activated and Byzantine, the demon gives its new state
3. If it is activated and not Byzantine (i.e. Good),
   a. **Look**: get information from its surrounding
   b. **Compute** its destination
   c. **Move** to the destination

Demonic action: what does the demon decide in each round?

1. Pick which robots are activated
2. Decide how to update inactive robots ASYNC only
3. Update Byzantine robots as it wishes
4. Select the new frame of reference for non-Byzantine robots
5. Decide how to update them depending on their destination

Demonic action

Demonic action: what does the demon decide in each round?

1. Pick which robots are activated
2. Decide how to update inactive robots
3. Update Byzantine robots as it wishes
4. Select the new frame of reference for non-Byzantine robots
5. Decide how to update them depending on their destination

Demonic action

Demonic action: what does the demon decide in each round?

1. Pick which robots are activated
2. Decide how to update inactive robots
3. Update Byzantine robots as it wishes
4. Select the new frame of reference for non-Byzantine robots
5. Decide how to update them depending on their destination

```
(** Select which robots are activated *)
activate : ident → bool;
(** Update the state of inactive robots *)
choose_inactive : configuration → ident → info;
(** Update the state of (activated) byzantine robots *)
relocate_byz : configuration → B → info;
(** Local referential for (activated) good robots in the compute phase *)
change_frame : configuration → G → bijection location;
(** Update the state of (activated) good robots in the move phase *)
choose_update : configuration → G → location → info;
```

+ compatibility properties (Proper)

The Core of Pactole: the Round Function

```
Definition round (r : robogram) (da : demonic_action) (config : configuration )
  : configuration :=
  fun id ⇒                    (* for a given robot, we compute the new state *)
```

The Core of Pactole: the Round Function

```
Definition round (r : robogram) (da : demonic_action) (config : configuration )
 : configuration :=
 fun id ⇒                          (* for a given robot, we compute the new state *)
   if da.( activate ) id              (* first see whether the robot is activated *)
   then











   else inactive config id (da.(choose_inactive) config id ).
```

The Core of Pactole: the Round Function

```
Definition round (r : robogram) (da : demonic_action) (config : configuration )
 : configuration :=
 fun id ⇒                          (* for a given robot, we compute the new state *)
   if da.( activate ) id              (* first see whether the robot is activated *)
  then
    match id with
      | Byz b ⇒ da.(relocate_byz) config b (* byzantine robots *)
      | Good g ⇒








    end
  else inactive config id (da.(choose_inactive) config id ).
```

The Core of Pactole: the Round Function

```
Definition round (r : robogram) (da : demonic_action) (config : configuration )
 : configuration :=
 fun id ⇒                        (* for a given robot, we compute the new state *)
   if da.( activate ) id            (* first see whether the robot is activated *)
  then
    match id with
      | Byz b ⇒ da.(relocate_byz) config b (* byzantine robots *)
      | Good g ⇒
        (* change the frame of reference *)
        let frame_choice := da.(change_frame) config g in
        let new_frame := frame_choice_bijection frame_choice in
        let local_config := map_config (lift new_frame ( ... )) config in
        let local_state := local_config (Good g) in









    end
  else inactive config id (da.(choose_inactive) config id ).
```

The Core of Pactole: the Round Function

```
Definition round (r : robogram) (da : demonic_action) (config : configuration )
 : configuration :=
 fun id ⇒                          (* for a given robot, we compute the new state *)
   if da.( activate ) id              (* first see whether the robot is activated *)
  then
    match id with
      | Byz b ⇒ da.(relocate_byz) config b (* byzantine robots *)
      | Good g ⇒
        (* change the frame of reference *)
        let frame_choice := da.(change_frame) config g in
        let new_frame := frame_choice_bijection frame_choice in
        let local_config := map_config (lift new_frame ( ... )) config in
        let local_state := local_config (Good g) in
        (* compute the observation *)
        let obs := obs_from_config local_config local_state in







    end
  else inactive config id (da.(choose_inactive) config id ).
```

The Core of Pactole: the Round Function

```
Definition round (r : robogram) (da : demonic_action) (config : configuration )
 : configuration :=
 fun id ⇒                          (* for a given robot, we compute the new state *)
   if da.( activate ) id              (* first see whether the robot is activated *)
   then
     match id with
       | Byz b ⇒ da.(relocate_byz) config b (* byzantine robots *)
       | Good g ⇒
         (* change the frame of reference *)
         let frame_choice := da.(change_frame) config g in
         let new_frame := frame_choice_bijection frame_choice in
         let local_config := map_config (lift new_frame ( ... )) config in
         let local_state := local_config (Good g) in
         (* compute the observation *)
         let obs := obs_from_config local_config local_state in
         (* apply r on observation *)
         let decision := r obs in







     end
   else inactive config id (da.(choose_inactive) config id ).
```

The Core of Pactole: the Round Function

```
Definition round (r : robogram) (da : demonic_action) (config : configuration )
  : configuration :=
  fun id ⇒                         (* for a given robot, we compute the new state *)
    if da.( activate ) id            (* first see whether the robot is activated *)
    then
      match id with
        | Byz b ⇒ da.(relocate_byz) config b (* byzantine robots *)
        | Good g ⇒
          (* change the frame of reference *)
          let frame_choice := da.(change_frame) config g in
          let new_frame := frame_choice_bijection frame_choice in
          let local_config := map_config (lift new_frame ( ... )) config in
          let local_state := local_config (Good g) in
          (* compute the observation *)
          let obs := obs_from_config local_config local_state in
          (* apply r on observation *)
          let decision := r obs in
          (* the demon chooses how to perform the state update *)
          let choice := da.(choose_update) local_config g decision in




      end
    else inactive config id (da.(choose_inactive) config id ).
```

The Core of Pactole: the Round Function

```
Definition round (r : robogram) (da : demonic_action) (config : configuration )
 : configuration :=
 fun id ⇒                         (* for a given robot, we compute the new state *)
   if da.( activate ) id              (* first see whether the robot is activated *)
   then
     match id with
       | Byz b ⇒ da.(relocate_byz) config b (* byzantine robots *)
       | Good g ⇒
         (* change the frame of reference *)
         let frame_choice := da.(change_frame) config g in
         let new_frame := frame_choice_bijection frame_choice in
         let local_config := map_config (lift new_frame ( ... )) config in
         let local_state := local_config (Good g) in
         (* compute the observation *)
         let obs := obs_from_config local_config local_state in
         (* apply r on observation *)
         let decision := r obs in
         (* the demon chooses how to perform the state update *)
         let choice := da.(choose_update) local_config g decision in
         (* the actual update of the robot state is performed by the update function *)
         let new_local_state := update local_config g frame_choice decision choice in


     end
   else inactive config id (da.(choose_inactive) config id ).
```

The Core of Pactole: the Round Function

```
Definition round (r : robogram) (da : demonic_action) (config : configuration )
 : configuration :=
 fun id ⇒                          (* for a given robot, we compute the new state *)
   if da.( activate ) id              (* first see whether the robot is activated *)
  then
    match id with
      | Byz b ⇒ da.(relocate_byz) config b (* byzantine robots *)
      | Good g ⇒
        (* change the frame of reference *)
        let frame_choice := da.(change_frame) config g in
        let new_frame := frame_choice_bijection frame_choice in
        let local_config := map_config (lift new_frame ( ... )) config in
        let local_state := local_config (Good g) in
        (* compute the observation *)
        let obs := obs_from_config local_config local_state in
        (* apply r on observation *)
        let decision := r obs in
        (* the demon chooses how to perform the state update *)
        let choice := da.(choose_update) local_config g decision in
        (* the actual update of the robot state is performed by the update function *)
        let new_local_state := update local_config g frame_choice decision choice in
        (* return to the global frame of reference *)
        lift (new_frame ^-1) (...) new_local_state
    end
  else inactive config id (da.(choose_inactive) config id ).
```

How to Constrain the Demon to Follow the Model?

```
(** Update the state of good robots in the move phase *)
choose_update : configuration → G → location → info
```

This is too powerful: the demon could do anything!

How to Constrain the Demon to Follow the Model?

```
(** Update the state of good robots in the move phase *)
choose_update : configuration → G → location → Tactive
```

This is too powerful: the demon could do anything!

Instead, the demon just give orders as abstract datatypes.
Then an update function performs the update.

```
(** Updates for active and inactive robots *)
update :  configuration → G → location → Tactive → info;
inactive  :  configuration → ident → Tinactive → info;
```

How to Constrain the Demon to Follow the Model?

```
(** Update the state of good robots in the move phase *)
choose_update : configuration → G → location → Tactive
```

This is too powerful: the demon could do anything!

Instead, the demon just give orders as abstract datatypes.
Then an update function performs the update.

```
(** Updates for active and inactive robots *)
update : configuration → G → location → Tactive → info;
 inactive : configuration → ident → Tinactive → info;
```

Example

When active, the demon chooses half/full move: `Tactive := bool`

Nothing happens when inactive: `Tinactive := unit`

```
update := fun _ _ target choice ⇒
   if choice then target ratio_1 else target (1 /r 2);
 inactive := fun config id _ ⇒ config id;
```

Demonic action

Demonic action: what does the demon decide in each round?

1. Pick which robots are activated
2. Decide how to update inactive robots
3. Update Byzantine robots as it wishes
4. Select the new frame of reference for non-Byzantine robots
5. Decide how to update them depending on their destination

```
(** Select which robots are activated *)
activate : ident → bool;
(** Update the state of inactive robots *)
choose_inactive : configuration → ident → info;
(** Update the state of (activated) byzantine robots *)
relocate_byz : configuration → B → info;
(** Local referential for (activated) good robots in the compute phase *)
change_frame : configuration → G → bijection location;
(** Update the state of (activated) good robots in the move phase *)
choose_update : configuration → G → location → info;
```

+ compatibility properties (Proper)

Demonic action

Demonic action: what does the demon decide in each round?

1. Pick which robots are activated
2. Decide how to update inactive robots
3. Update Byzantine robots as it wishes
4. Select the new frame of reference for non-Byzantine robots
5. Decide how to update them depending on their destination

```
(** Select which robots are activated *)
activate : ident → bool;
(** Update the state of inactive robots *)
choose_inactive : configuration → ident → Tinactive;
(** Update the state of (activated) byzantine robots *)
relocate_byz : configuration → B → info;
(** Local referential for (activated) good robots in the compute phase *)
change_frame : configuration → G → Tframe;
(** Update the state of (activated) good robots in the move phase *)
choose_update : configuration → G → location → Tactive;
```

+ compatibility properties (Proper)

Generalizing the Robogram

We can use the same trick for the robogram!

Change robogam : observation $\rightarrow$ location

to robogam : observation $\rightarrow$ Trobot

This way, we can model:

- curved trajectories
- direction of movement
- changes of orientation/color/memory
- ...

Generalizing the Robogram

We can use the same trick for the robogram!

Change `robogam : observation → location`

to `robogam : observation → Trobot`

This way, we can model:

- curved trajectories
- direction of movement
- changes of orientation/color/memory
- ...

Don't forget to also change

```
choose_update : configuration → G → Trobot → Tactive;
update : configuration → G → Trobot → Tactive → info;
```

Pactole: a Coq Framework for Mobile Robots

Main Ingredients:

- Robogram
- Demon (scheduler)
- Round
- Execution
- Properties and Proofs

observation → Trobot

Execution

Execution = infinite sequence of configurations
⇝ a stream of configurations

```
Definition execution := Stream.t configuration .
```

Execution

Execution = infinite sequence of configurations
$\rightsquigarrow$ a stream of configurations

```
Definition execution := Stream.t configuration .
```

How to build streams?

Execution

Execution = infinite sequence of configurations
$\rightsquigarrow$ a stream of configurations

```
Definition execution := Stream.t configuration .
```

How to build streams?

- Coinductive type
- Constructors

```
Definition Stream.cons : A → Stream.t A → Stream.t A.
Definition Stream.constant : A → Stream.t A.
Definition Stream.alternate : A → A → Stream.t A.
```

Pactole: a Coq Framework for Mobile Robots

Main Ingredients:

- Robogram
- Demon (scheduler)
- Round
- Execution
- Properties and Proofs

observation → Trobot

Stream.t configuration

Expressing Properties

We can use all of Coq (inductive/coinductive, higher-order, etc.)
$\rightsquigarrow$ Follow the mathematical definition

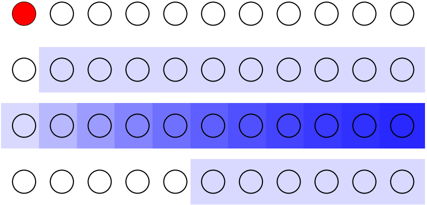
Expressing Properties

We can use all of Coq (inductive/coinductive, higher-order, etc.)
$\rightsquigarrow$ Follow the mathematical definition

For streams (demon/execution), we define stream operators:

- P : configuration → Prop
- P : execution → Prop

- Stream.instant P:
- Stream.next P:
- Stream.forever P:
- Stream.eventually P:



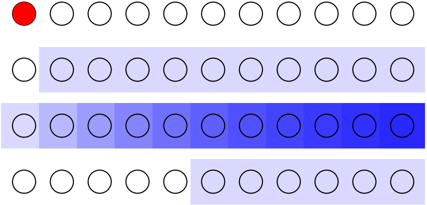
Expressing Properties

We can use all of Coq (inductive/coinductive, higher-order, etc.)
⇝ Follow the mathematical definition

For streams (demon/execution), we define stream operators:

- P : configuration → Prop
- P : execution → Prop

- Stream.instant P:
- Stream.next P:
- Stream.forever P:
- Stream.eventually P:



Also useful for defining fairness conditions over demons

(cf. exercises)

Example of Properties: Gathering

Definition (Gathering Problem)

Robots gather if all (non byzantine) robots reach in finite time the same location (unknown ahead of time) and then stay there.

Example of Properties: Gathering

Definition (Gathering Problem)

Robots gather if all (non byzantine) robots reach in finite time the same location (unknown ahead of time) and then stay there.

Example of Properties: Gathering

Definition (Gathering Problem)

Robots gather if all (non byzantine) robots reach in finite time the same location (unknown ahead of time) and then stay there.

```
(* All good robots are at the same location pt (exactly). *)
Definition gathered_at (pt : loc) (config : configuration) :=
  ∀ g, get_location (config (Good g)) = pt.

(* At all rounds of the execution e, robots are gathered at pt. *)
Definition Gather (pt : loc) (e : execution) : Prop :=
  Stream.forever (Stream.instant (gathered_at pt)) e.

(* The (infinite) execution e is *eventually* Gathered. *)
Definition WillGather (pt : loc) (e : execution) : Prop :=
  Stream.eventually (Gather pt) e.
```

Proving in Pactole

0. Instantiate your setting

Correctness proof:

1. Formalize your problem
2. Write your algorithm

4. Prove that your algorithm solves your problem
   following your paper proof

Proving in Pactole

0. Instantiate your setting

Correctness proof:

1. Formalize your problem
2. Write your algorithm
3. Express your algorithm in the global frame of reference

4. Prove that your algorithm solves your problem
   following your paper proof

Two different points of view

Global View (demon)	**Local View** (robots)
absolute location	local frame of reference
robots: Byzantine or not (B / G)	indistinguishable robots
identifiers ident = $G + B$	
configuration = ident → location	local configuration
	observation (abstract type)
	robogram
	: observation → Trobot
round r d : config → config	
execution = Stream.t configuration	

Proving in Pactole

0. Instantiate your setting

Correctness proof:

1. Formalize your problem
2. Write your algorithm
3. Express your algorithm in the global frame of reference
   Use geometric patterns that are invariant by change of frame

   ```
   Lemma round_simplify :∀d config, round r d config == ...
   ```
4. Prove that your algorithm solves your problem
   following your paper proof

Proving in Pactole

0. Instantiate your setting

Correctness proof:

1. Formalize your problem
2. Write your algorithm
3. Express your algorithm in the global frame of reference
   Use geometric patterns that are invariant by change of frame

   ```
   Lemma round_simplify :∀d config, round r d config == ...
   ```

4. Prove that your algorithm solves your problem
   following your paper proof

Impossibility proof:

1. Formalize your problem
2. Assume given a robogram (a variable) + its properties
3. Prove that the algorithm does not solve the problem

Universal Algorithms

We can easily formalize what a universal algorithm is:

- Under some conditions, the problem is unsolvable
- Outside these conditions, the algorithm works

Universal Algorithms

We can easily formalize what a universal algorithm is:

- Under some conditions, the problem is unsolvable
- Outside these conditions, the algorithm works

```
Lemma impossibility : ∀ r, ∃ d, ∀ config,
   invalid config → ¬good_execution (execute r d config).

Lemma correctness r : ∀ d, ∀ config,
  ¬invalid config → good_execution (execute r d config).
```

Conclusion About Pactole

- ⊕ Designed for mobile robot networks
- ⊕ Ease of use for specification
- ⊕ Broadly applicable

Conclusion About Pactole

- Designed for mobile robot networks
  -  cycle built-in
  - Other features are possible memory, battery, . . .
- Ease of use for specification
- Broadly applicable

Conclusion About Pactole

- ⊕ Designed for mobile robot networks
  -  cycle built-in
  - Other features are possible — memory, battery, . . .
- ⊕ Ease of use for specification
  - Expressive logic
  - Maths can be directly expressed
  - Principled bottom-up approach
- ⊕ Broadly applicable

Conclusion About Pactole

⊕ Designed for mobile robot networks

-  cycle built-in
- Other features are possible — memory, battery, ...

⊕ Ease of use for specification

- Expressive logic
- Maths can be directly expressed
- Principled bottom-up approach

⊕ Broadly applicable

- Highly parametric — space, sensors, execution model, ...
- Very expressive
- Common base of definition (no more mismatches)

⇒ A junction point for several formal results

Conclusion About Pactole

- ⊕ Designed for mobile robot networks
  -  cycle built-in
  - Other features are possible — memory, battery, ...
- ⊕ Ease of use for specification
  - Expressive logic
  - Maths can be directly expressed
  - Principled bottom-up approach
- ⊕ Broadly applicable
  - Highly parametric — space, sensors, execution model, ...
  - Very expressive
  - Common base of definition (no more mismatches)

$\Rightarrow$ A junction point for several formal results

- ⊖ Caveat
  - No fully automated procedure (yet)
  - Building proofs is a lot of work

Outline

Gathering

Objective

Have all (non byzantine) robots reach the same location in finite time and then stay there.

Gathering

Objective

Have all (non byzantine) robots reach the same location in finite time and then stay there.

Let's start simple:

- On a real line
- No byzantine/crash
- FSYNC execution
- As much info as you want (but still anonymous)

How to do it?

Gathering

Objective

Have all (non byzantine) robots reach the same location in finite time and then stay there.

Let's start simple:

- On a real line
- No byzantine/crash
- FSYNC execution
- As much info as you want (but still anonymous)

How to do it? Easy: move to the (bary)center

Gathering

Objective

Have all (non byzantine) robots reach the same location in finite time and then stay there.

Let's start simple:

- On a real line
- No byzantine/crash
- FSYNC execution
- As much info as you want (but still anonymous)

How to do it? Easy: move to the (bary)center

What about SSYNC?

Gathering

Objective

Have all (non byzantine) robots reach the same location in finite time and then stay there.

Let's start simple:

- On a real line
- No byzantine/crash
- FSYNC execution
- As much info as you want (but still anonymous)

How to do it? Easy: move to the (bary)center

What about SSYNC? Impossible!

Gathering

Objective

Have all (non byzantine) robots reach the same location in finite time and then stay there.

Let's start simple:

- On a real line
- No byzantine/crash
- FSYNC execution
- As much info as you want (but still anonymous)

How to do it? Easy: move to the (bary)center

What about SSYNC? Impossible!

Theorem [Suzuki & Yamashita 99]

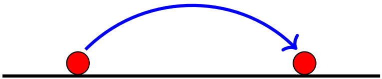
Gathering is impossible even for 2 robots only.

Proof

By symmetry, both robots act the same.

Two cases:

Proof

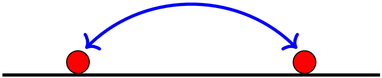


By symmetry, both robots act the same.

Two cases:

1. Left robot moves to the right one

Proof



By symmetry, both robots act the same.

Two cases:

1. Left robot moves to the right one
   activate both: swap locations

Proof

By symmetry, both robots act the same.

Two cases:

2. Left robot goes anywhere else

Proof

By symmetry, both robots act the same.

Two cases:

2. Left robot goes anywhere else
   activate only one: same configuration up to scale

Proof

By symmetry, both robots act the same.

Two cases:

1. Left robot moves to the right one
   activate both: swap locations
2. Left robot goes anywhere else
   activate only one: same configuration up to scale

In both cases, similar configuration at the next round

Proof

By symmetry, both robots act the same.

Two cases:

1. Left robot moves to the right one
   activate both: swap locations
2. Left robot goes anywhere else
   activate only one: same configuration up to scale

In both cases, similar configuration at the next round

Generalizations:

- even number of robots
- type of line ($\mathbb{Q}$ ou $\mathbb{R}$)

Going Beyond Impossibility

Gathering is impossible in general

Why?

Going Beyond Impossibility

Gathering is impossible in general

Why? Because we cannot break symmetry

Here: two towers of the same size (bivalent config)

What about other configurations?

Going Beyond Impossibility

Gathering is impossible in general

Why? Because we cannot break symmetry

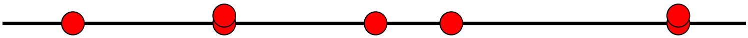
Here: two towers of the same size (bivalent config)

What about other configurations?

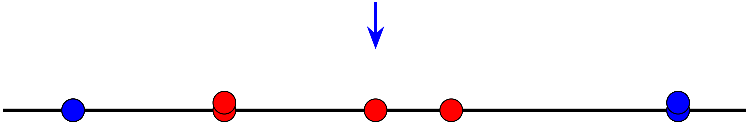
There is an algorithm! no byzantine, #robots ≥ 3

```
Definition solGathering (r : robogram) :=
  ∀ d : demon, SSYNC d → Fair d →
  ∀ config , ¬bivalent conf → ∃pt, WillGather pt (execute r d config).
```

Gathering Algo

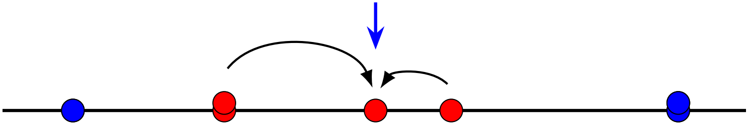


Gathering Algo



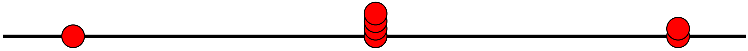
1. Find the middle of the extreme location.

Gathering Algo



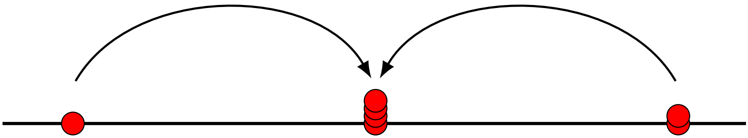
1. Find the middle of the extreme location.
   Move non extreme robots there.

Gathering Algo



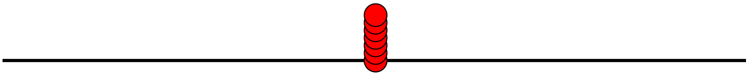
1. Find the middle of the extreme location.
   Move non extreme robots there.

Gathering Algo



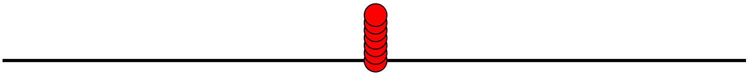
1. Find the middle of the extreme location.
   Move non extreme robots there.
2. If 3 equidistant towers, move extreme robots to the middle.

Gathering Algo



1. Find the middle of the extreme location.
   Move non extreme robots there.
2. If 3 equidistant towers, move extreme robots to the middle.

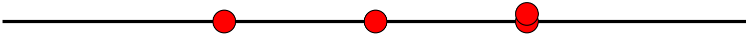
Gathering Algo



Careful to the bivalent config! SSYNC

1. Find the middle of the extreme location.
   Move non extreme robots there.
2. If 3 equidistant towers, move extreme robots to the middle.

Gathering Algo

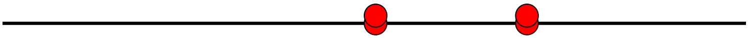


Careful to the bivalent config!

SSYNC

1. Find the middle of the extreme location.
   Move non extreme robots there.
2. If 3 equidistant towers, move extreme robots to the middle.

Gathering Algo

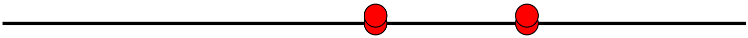


Careful to the bivalent config!

SSYNC

1. Find the middle of the extreme location.
   Move non extreme robots there.
2. If 3 equidistant towers, move extreme robots to the middle.

Gathering Algo

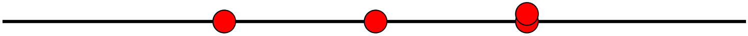


Careful to the bivalent config!

SSYNC

0. If there is a (unique) majority tower, go there.
1. Find the middle of the extreme location.
   Move non extreme robots there.
2. If 3 equidistant towers, move extreme robots to the middle.

Gathering Algo

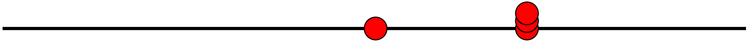


Careful to the bivalent config!

SSYNC

0. If there is a (unique) majority tower, go there.
1. Find the middle of the extreme location.
   Move non extreme robots there.
2. If 3 equidistant towers, move extreme robots to the middle.

Gathering Algo



Careful to the bivalent config!

SSYNC

0. If there is a (unique) majority tower, go there.
1. Find the middle of the extreme location.
   Move non extreme robots there.
2. If 3 equidistant towers, move extreme robots to the middle.

Why Does It Work? FSYNC Case

3 configurations to consider:

Why Does It Work? FSYNC Case

3 configurations to consider:

1. If majority tower:
   all robots move toward to same location

Why Does It Work? FSYNC Case

3 configurations to consider:

1. If majority tower: ✓ in 1 round
   all robots move toward to same location

Why Does It Work? FSYNC Case

3 configurations to consider:

1. If majority tower: ✓ in 1 round
   all robots move toward to same location
2. Otherwise, If 3 towers (no majority):
   idem

Why Does It Work? FSYNC Case

3 configurations to consider:

1. If majority tower: ✓ in 1 round
   all robots move toward to same location
2. Otherwise, If 3 towers (no majority): ✓ in 1 round
   idem

Why Does It Work? FSYNC Case

3 configurations to consider:

1. If majority tower: ✓ in 1 round
   all robots move toward to same location
2. Otherwise, If 3 towers (no majority): ✓ in 1 round
   idem
3. Otherwise, general case:
   back to case 1 or 2 at the next round

Why Does It Work? FSYNC Case

3 configurations to consider:

1. If majority tower: ✓ in 1 round
   all robots move toward to same location
2. Otherwise, If 3 towers (no majority): ✓ in 1 round
   idem
3. Otherwise, general case: ✓ in 2 rounds
   back to case 1 or 2 at the next round

Why Does It Work? SSYNC Case

Idea: adapt FSYNC proofs

Issue: Movement are not done in a single step
$\rightsquigarrow$ may lead to interference

Why Does It Work? SSYNC Case

Idea: adapt FSYNC proofs

Issue: Movement are not done in a single step
$\rightsquigarrow$ may lead to interference

Hopefully,

- in each case, we want to reach a configuration that does not depend on the robots that should move
- there is no memory
- we never backtrack

$\rightsquigarrow$ Just wait long enough — fairness

Why Does It Work? SSYNC Case

Idea: adapt FSYNC proofs

Issue: Movement are not done in a single step
$\rightsquigarrow$ may lead to interference

Hopefully,

- in each case, we want to reach a configuration
  that does not depend on the robots that should move
- there is no memory
- we never backtrack

$\rightsquigarrow$ Just wait long enough — fairness

And formally?

Why Does It Work? SSYNC Case

Idea: adapt FSYNC proofs

Issue: Movement are not done in a single step
$\rightsquigarrow$ may lead to interference

Hopefully,

- in each case, we want to reach a configuration that does not depend on the robots that should move
- there is no memory
- we never backtrack

$\rightsquigarrow$ Just wait long enough fairness

And formally?
Separate proofs for correctness and termination

Correctness and Termination

(Partial) Correctness: if there is a result, it is correct easy!

- non bivalent + non gathered $\implies$ a robot should move
- OK by contrapositive

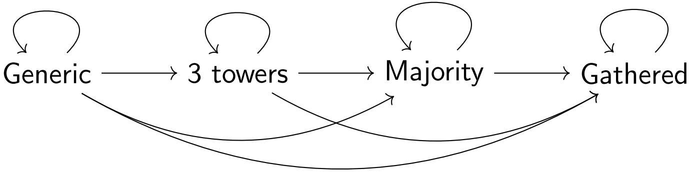
Correctness and Termination

(Partial) Correctness: if there is a result, it is correct easy!

- non bivalent + non gathered $\implies$ a robot should move
- OK by contrapositive

Termination: there is a result hard!

- lexicographic order on configurations
- movement $\implies$ smaller configuration



Main Steps of the Coq Formalization

1. Ability to detect majority towers   observation = multisets
2. Express the configuration after one round
   $\rightsquigarrow$ local/global frame of reference
   $\rightsquigarrow$ depends on the demon (which robots are activated?)
3. In a round, robots always move toward a single location
4. Bivalent configuration cannot appear
5. No backtrack to previous configurations (lexico order)
6. By fairness of the demon, a robot will eventually move

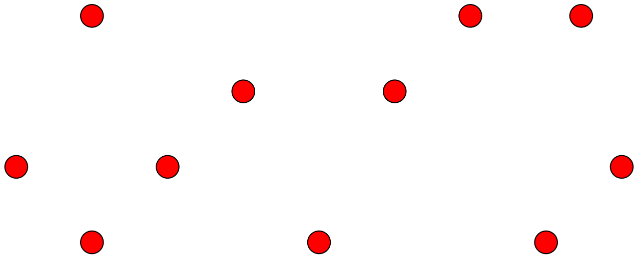
Going to 2D

Going to 2D

Switch « middle » for « center of the smallest enclosing circle ».

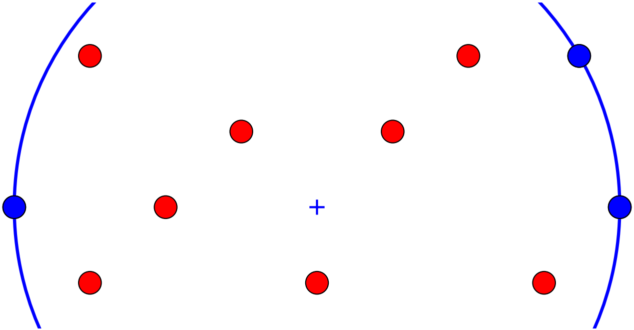
Going to 2D

Switch « middle » for « center of the smallest enclosing circle ».



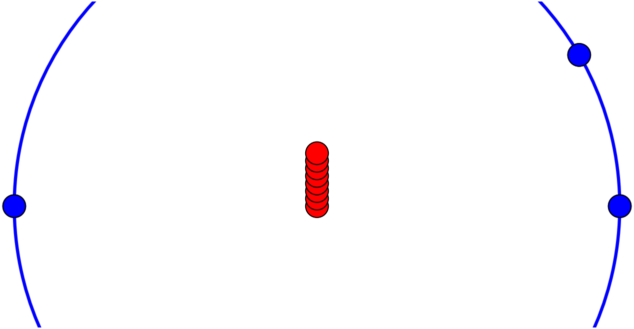
Going to 2D

Switch « middle » for « center of the smallest enclosing circle ».



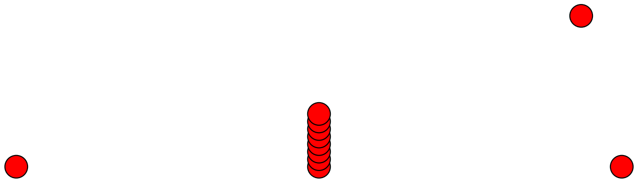
Going to 2D

Switch « middle » for « center of the smallest enclosing circle ».



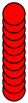
Going to 2D

Switch « middle » for « center of the smallest enclosing circle ».



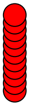
Going to 2D

Switch « middle » for « center of the smallest enclosing circle ».



Going to 2D

Switch « middle » for « center of the smallest enclosing circle ».



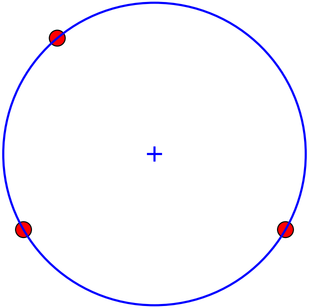
Problem in the last step!

Problem in 2D

It is possible to backtrack on the two phases of the algorithm.

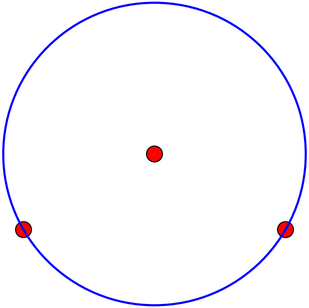
Problem in 2D

It is possible to backtrack on the two phases of the algorithm.



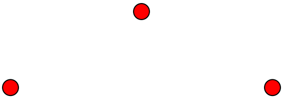
Problem in 2D

It is possible to backtrack on the two phases of the algorithm.



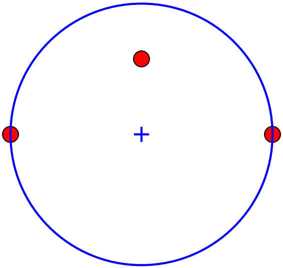
Problem in 2D

It is possible to backtrack on the two phases of the algorithm.



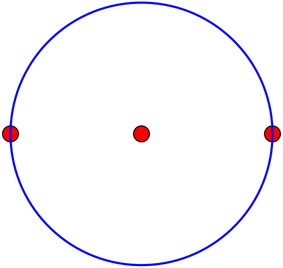
Problem in 2D

It is possible to backtrack on the two phases of the algorithm.



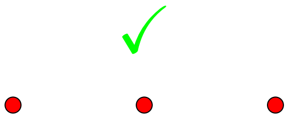
Problem in 2D

It is possible to backtrack on the two phases of the algorithm.



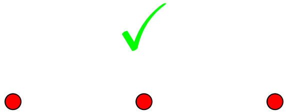
Problem in 2D

It is possible to backtrack on the two phases of the algorithm.



Problem in 2D

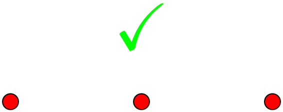
It is possible to backtrack on the two phases of the algorithm.



Does it work in general?

Problem in 2D

It is possible to backtrack on the two phases of the algorithm.



Does it work in general?

Yes, but we need to change the proof cases

Problem in 2D Solved

Key Ideas

- #robots on the circle decreases
  $\rightsquigarrow$ big cases of the proof

Problem in 2D Solved

Key Ideas

- #robots on the circle decreases
  $\rightsquigarrow$ big cases of the proof

n/a Majority

2. Diameter
3. Triangle
   - scalene, isosceles
   - equilateral

4+. Generic

Most cases = same as 1D

Problem in 2D Solved

Key Ideas

- #robots on the circle decreases $\rightsquigarrow$ big cases of the proof
- Clean/dirty config in each case
  - clean: config $\subseteq$ circle $\cup$ center
  - dirty: otherwise

n/a Majority

2. Diameter
3. Triangle
   - scalene, isosceles
   - equilateral

4+. Generic

Most cases = same as 1D

Problem in 2D Solved

Key Ideas

- #robots on the circle decreases
  $\rightsquigarrow$ big cases of the proof
- Clean/dirty config in each case
  - clean: config $\subseteq$ circle $\cup$ center
  - dirty: otherwise
- Avoid looping with triangles

n/a Majority

2. Diameter
3. Triangle
   - scalene, isosceles
   - equilateral

4+. Generic

Most cases = same as 1D

Problem in 2D Solved

Key Ideas

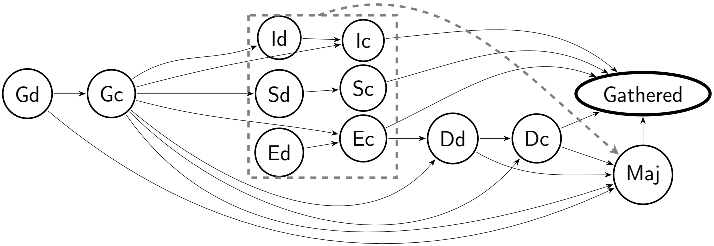
- #robots on the circle decreases
  ⇝ big cases of the proof
- Clean/dirty config in each case
  - clean: config ⊆ circle ∪ center
  - dirty: otherwise
- Avoid looping with triangles

n/a. Majority

2. Diameter
3. Triangle
   - scalene, isosceles
   - equilateral

4+. Generic

Most cases = same as 1D



Overall, What Do We Need?

Lots of geometric properties:

- Invariance through frame change (similarity)
- Evolution of configurations along the algorithm (SEC)
- Some classical properties (barycenter)

A few other things:

- Permutations
- Except for termination, everything is easy (like in 1D)

Outline

Before the Practice Session

- Make sure you have downloaded and extracted the Pactole package from the course website

  `https://perso.liris.cnrs.fr/xavier.urbain/ens/m2ensl.html`

  or from the following link (from the pad of the class)

  `https://www-verimag.imag.fr/~riegl/teaching/package.tgz`

- Start compiling Pactole:
  Use `make` at the root of the package $\approx$ 5-6 minutes

- Possible internships
  - Topics (flexible): (non-euclidian) geometry
    certification for randomized algo
    automated proofs (graphs + swarms)
  - ANR project SAPPORO (France/Japan)
  - Collaboration between:
    Lyon 1 / Sorbonne univ. / CNAM Paris / Tokyo-Tech.

Structure of the libraries

Pactole:

- `Util/`
  Complements to Coq's libraries, external libraries
- `Core/`
  Core of the formalism
- `Spaces/`
  The spaces in which robots evolve
- `Observations/`
  The information available for robograms
- `Models/`
  Additional constraints on models
- `CaseStudies/`
  - `Gathering`
  - `Convergence`
  - `Exploration`

What do you need to set it up?

Parameters that need to be imported/instantiated:

- The core of the formalism
- A space in which robots evolve
- A type of observation
- (optional) Extra constraints on your setting

- The number of robots in your case (can be parameters)
- A type of state for robots (containing the space)
- The choices made by the demon:
  - the change of frame: frame_choice
  - the choices for update for active and inactive robots: update_choice and inactive_choice
- The `update` and `inactive` functions

What do you need to set it up?

Parameters that need to be imported/instantiated:

- The core of the formalism
- A space in which robots evolve
- A type of observation
- (optional) Extra constraints on your setting

- The number of robots in your case (can be parameters)
- A type of state for robots (containing the space)
- The choices made by the demon:
  - the change of frame: frame_choice
  - the choices for update for active and inactive robots: update_choice and inactive_choice
- The `update` and `inactive` functions

Everything is handled through typeclasses
(typeclasses = mechanism for modularity and overloading)

Why typeclasses?

Advantages:

- Glues everything together
- Does not require a specific order
- Better separation of concerns
- More flexibility for partial instances

But:

- Infinite loops if missing instances
- Unpredictable results if more than one instance
- Use About rather than Check
- Rather large and unpleasant unfoldings

How to do it? (1/6)

2 steps:

1. Require Import the files you need
2. Define the adequate Instances

How to do it? (1/6)

2 steps:

1. `Require Import` the files you need
2. Define the adequate `Instance`s

First step: Require Import

- Formalism core

```
Require Import Pactole.Setting.
```

- Space

```
Require Import Pactole.Spaces.XXX.
```

- Observations

```
Require Import Pactole.Observations.XXX.
```

- Extra constraints

```
Require Import Pactole.Models.XXX.
```

How to do it? (2/6)

Second step: Instances

- ~~Formalism core~~
- Space — type with decidable equivalence

```
Class Location := {
  location : Type;
  location_Setoid :> Setoid location ;
  location_EqDec :> EqDec location_Setoid }.
```

```
Instance XXX : Location := XXX.
```

Often: `Instance XXX : Location := make_Location XXX.`

- ~~Observation~~
- Extra contraints (depends on what you want)

```
Instance Update : RigidSetting .
```

How to do it? (3/6): Number of Robots

```
Instance XXX : Names := Robots nG nB.
```

nG = number of good robots
nB = number of Byzantine robots

How to do it? (3/6): Number of Robots

```
Instance XXX : Names := Robots nG nB.
```

nG = number of good robots
nB = number of Byzantine robots

nG and nB can be left as variables

```
Parameter n : nat.
Hypothesis n_non_0 : n ≠ 0.
Instance MyRobots : Names := Robots (2 * n) 0.
```

How to do it? (4/6): State of Robots

```
Instance XXX : State info := XXX.

Instance Info : State location := OnlyLocation _.
```

How to do it? (4/6): State of Robots

```
Instance XXX : State info := XXX.

Instance Info : State location := OnlyLocation _.
```

```
Class State {Loc : Location} info := {
  get_location : info → location ;










    ... }.
```

How to do it? (4/6): State of Robots

```
Instance XXX : State info := XXX.

Instance Info : State location := OnlyLocation _.
```

```
Class State {Loc : Location} info := {
  get_location : info → location ;
  (** States are equipped with a decidable equality *)
  state_Setoid :> Setoid info ;
  state_EqDec :> EqDec state_Setoid;









    ... }.
```

How to do it? (4/6): State of Robots

```
Instance XXX : State info := XXX.

Instance Info : State location := OnlyLocation _.
```

```
Class State {Loc : Location} info := {
  get_location : info → location ;
  (** States are equipped with a decidable equality *)
  state_Setoid :> Setoid info ;
  state_EqDec :> EqDec state_Setoid;
  (** Lifting a change of frame from a location to a full state *)
  precondition : (location → location) → Prop;
  lift : forall f, precondition f → info → info ;




    ... }.
```

How to do it? (4/6): State of Robots

```
Instance XXX : State info := XXX.

Instance Info : State location := OnlyLocation _.
```

```
Class State {Loc : Location} info := {
  get_location : info → location ;
  (** States are equipped with a decidable equality *)
  state_Setoid :> Setoid info ;
  state_EqDec :> EqDec state_Setoid;
  (** Lifting a change of frame from a location to a full state *)
  precondition : (location → location) → Prop;
  lift : forall f, precondition f → info → info ;
  (** Properties (compatibility, ...) *)
  lift_id : forall Pid, @lift id Pid == id;
  get_location_lift : forall f (Pf : precondition f) state,
    get_location (@lift f Pf state) == f (get_location state);
   ... }.
```

In Practice: Template.v (1/2)

```
Require Import Pactole.Setting.

(* Number of robots *)
Parameter n : nat.
Axiom n_non_0 : n <> 0.
Instance MyRobots : Names := Robots (2 * n) 0.

(* Space and state and robot choice *)
Require Import Pactole.Spaces.R2.
Close Scope R_scope.
Instance Loc : Location := make_Location R2.
Instance Info : State location := OnlyLocation (...).
Instance RC : robot_choice location := {
  robot_choice_Setoid := location_Setoid }.

(* Type of observations *)
Require Import Pactole.Observations.MultisetObservation.
```

How to do it? (5/6): Demon & Robot Choices

```
Instance XXX : robot_choice Trobot := XXX.
Instance XXX : frame_choice Tframe := XXX.
Instance XXX : update_choice Tactive := XXX.
Instance XXX : inactive_choice Tinactive := XXX.

Instance FC : frame_choice (Similarity.similarity location) :=
  FrameChoiceSimilarity.
Instance UC : update_choice unit := {
  update_choice_EqDec := unit_eqdec}.
```

```
Class frame_choice Tframe := {
  frame_choice_bijection : Tframe → bijection location;
  frame_choice_Setoid : Setoid Tframe;
  frame_choice_bijection_compat :
    Proper (equiv ==> equiv) frame_choice_bijection }.

Class update_choice Tactive := {
  update_choice_Setoid : Setoid Tactive;
  update_choice_EqDec : EqDec update_choice_Setoid }.
Class inactive_choice Tinactive := { ... }.
Class robot_choice Trobot := { ... }.
```

How to do it? (6/6): Update Functions

```
Instance  XXX : update_function Tactive := XXX.
Instance  XXX : inactive_function  Tinactive  := XXX.

Instance  UpdateFun : update_function bool := {
  update := fun _ _ target choice ⇒
     if  choice  then  target  ratio_1  else  target  (1 /r 2) }
Instance  UpdateFun : inactive_function  unit  := {
   inactive  := fun  config  id  _  ⇒ config  id  }.
```

```
Class  update_function ‘{robot_choice} ‘{frame_choice} ‘{update_choice} :=
  update :> configuration  → G → Tframe → Trobot →  Tactive  →  info ;
  update_compat :> Proper (equiv ==⇒ Logic.eq ==⇒ equiv ==⇒
                                  equiv  ==⇒ equiv ==⇒ equiv) update }.
Class  inactive_function   ‘{ inactive_choice} := {
   inactive   :> configuration  →  ident  →  Tinactive  →  info ;
  inactive_compat :> Proper (equiv ==⇒ Logic.eq ==⇒
                                    equiv  ==⇒ equiv)  inactive  }.
```

In Practice: `Template.v` (2/2)

```
(* Demon choices *)
Require Import Pactole.Models.Similarity.
Instance FC : frame_choice (Similarity.similarity location) :=
  FrameChoiceSimilarity.
Instance UC : update_choice unit := {update_choice_EqDec := unit_eqdec}.
Instance IC : inactive_choice unit := {inactive_choice_EqDec := unit_eqdec}.

(* Update functions *)
Instance UpdateFun : update_function unit := {
  update := fun _ _ _ pt _ ⇒pt }.
Proof. now repeat intro. Defined.

Instance InactiveFun : inactive_function unit := {
  inactive := fun config id _ ⇒ config id }.
Proof. now repeat intro; subst. Defined.

(* Properties about the framework *)
Require Import Pactole.Models.Rigid.
Instance Update : RigidSetting.
Proof. split. now intros. Qed.
```

Another Example: Gathering/Definitions.v

```
Require Export Pactole.Setting.
Require Export Pactole.Spaces.RealMetricSpace.
Require Pactole.Spaces.Similarity.

Section GatheringDefinitions.

(* We only required the space to be a real metric space.
   The actual number of robots is arbitrary. *)
Context {Tactive Tinactive : Type}.
Context {N : Names}.
Context {Loc : Location}.
Context {RMS : RealMetricSpace location}.

Global Instance Info : State location := OnlyLocation.

(** The observation and state updates are still arbitrary. *)
Context {Obs : Observation}.
Context {UC : update_choice Tactive}.
Context {IC : inactive_choice Tinactive}.
Context {UpdFun : update_functions Tactive Tinactive}.
```

Exercises

See file `exercises.v`**.**

Exercises

See file `exercises.v`**.**

- Modeling of problems
  - gathering
  - convergence
  - exploration with Stop
  - perpetual exploration
  - safety
- Properties of streams and demons
  - until / weak until
  - fully-synchronous / centralized demon
- State with lights
  - Internal: only visible by self (akin to memory)
  - External: only visible by others
  - Full: visible by all
- Rendezvous by Viglietta

Solution: Gathering

```
(* All good robots are at the same location [pt] (exactly). *)
Definition gathered_at (pt : location) (config : configuration) :=
  ∀ g, get_location (config (Good g)) == pt.

(* At all rounds of the execution [e], robots are gathered at [pt]. *)
Definition Gather (pt : location) (e : execution) : Prop :=
  Stream.forever (Stream.instant (gathered_at pt)) e.

(* The infinite execution [e] is *eventually* [Gather]ed. *)
Definition WillGather (pt : location) (e : execution) : Prop :=
  Stream.eventually (Gather pt) e.

Definition gathering (e : execution) := ∃ pt, WillGather pt e.
```

Solution: Convergence

```
(* All robots are contained in the disk defined by [center] and [radius]. *
Definition contained (center : location) (radius : R) config :=
  ∀ g, dist center (get_location (config (Good g))) ≤ radius.

(* All good robots stay confined in a small disk. *)
Definition imprisoned (center : location) (radius : R) (e : execution) :=
  Stream.forever (Stream.instant (contained center radius)) e.

(* The execution will end in a small disk. *)
Definition attracted (c : location) (r : R) (e : execution) : Prop :=
  Stream.eventually (imprisoned c r) e.

Definition convergence (e : execution) :=
  ∀ ε: R, 0 < ε → ∃ pt : location, attracted pt ε e

(* A solution ensures convergence for any demon and configuration. *)
Definition convergence_sol (r : robogram) : Prop :=
  ∀ d, Fair d → ∀ config, convergence (execute r d config).
```

Solution: Exploration with Stop

```
Definition visited pt config :=
  ∃ g, get_location (config (Good g)) == pt.
Definition will_be_visited pt e : Prop :=
  Stream.eventually (Stream.instant (visited pt)) e.

Definition stall (e : execution) :=
  Stream.hd e == Stream.hd (Stream.tl e).
Definition stopped (e : execution) : Prop :=
  Stream.forever stall e.
Definition will_stop (e : execution) : Prop :=
  Stream.eventually stopped e.

Definition Explore_and_Stop e :=
  (∀ pt, will_be_visited pt e) ∧
  will_stop e.
Definition is_solution (r : robogram) :=
∀ d config, Fair d → Explore_and_Stop (execute r d config).
```

Solution: Exploration with Stop & Safety

```
(** Perpetual exploration:
    each location is visited infinitely often. *)
Definition perpetual_exploration e :=
  forall pt, Stream.forever (Stream.eventually
               (Stream.instant (visited pt))) e.

(** Safety: stay outside of a given set [danger] of states. *)
Definition safe_config (danger : location → Prop) config :=
  forall g, ¬danger (config (Good g)).

Definition safe danger e :=
 Stream.forever (Stream.instant (safe_config danger)) e.
```

Solution: Until + FSYNC + Centralized

```
Inductive until (P Q : t A → Prop) (s : t A) : Prop :=
  | NotYet : P s → until P Q (tl s) → until P Q s
  | YesNow : Q s → until P Q s.
Definition weak_until P Q s := Stream.forever P s ∨ until P Q s.

Definition FSYNC_da da : Prop :=
 ∀ config g, activate da config g = true.
Definition FullySynchronous : demon → Prop :=
 Stream.forever (Stream.instant FSYNC_da).

Definition centralized_da da :=
 ∀ id1 id2, activate da id1 = true →
            activate da id2 = true → id1 = id2.
Definition centralized (d : demon) :=
 Stream.forever (Stream.instant centralized_da) d.
```

Solution: Lights

```
(* The simple version: only location and lights. *)
Context `{Location}.
Context {nbLights : nat}.
Definition lights := {k : nat | k < nbLights}.

Instance lights_Setoid : Setoid lights := @sig_Setoid _ _ _.
Instance lights_EqDec : EqDec lights_Setoid := sig_EqDec _ _.

Instance InfoWithLights : State (location * lights) := {|
  get_location := fst;
  state_Setoid := prod_Setoid location_Setoid lights_Setoid;
  precondition := fun _ => True;
  lift := fun f info => (projT1 f (fst info), snd info) |}.
Proof.
+ now intros [].
+ now intros f [].
+ now intros [] [] [].
+ intros f g Hfg [] [] []. simpl. split; trivial; []. now apply Hfg.
Defined.
```

Table of Contents