Formal Proofs for Mobile Robot Swarms

Lionel Rieg

ENS de Lyon, 23 Oct. 2019
Motivations

Overview of the Model

The Pactole Formalism

Case Study: Gathering

Pactole in Practice
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!
Inspiration: Swarms of Mobile Robots
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!
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
  - ...

- Opportunities?
  - Cooperative behavior (swarm intelligence)
  - Resilience

- Main challenge?
  - Understand what happens!
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!
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** → *very* error-prone
- **Careful of mismatch spec/proof!**
- **Lots of proof cases**

⇒ Formal methods can help
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 \(\leadsto\) very error-prone

- Careful of mismatch spec/proof!
- Lots of proof cases

\(\Rightarrow\) 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

Motivations

Overview of the Model

The Pactole Formalism

Case Study: Gathering

Pactole in Practice
Points

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

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

With Byzantine faults (or crash)
Robot Model

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

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

- Points
- With Byzantine faults (or crash)
- Anonymous
- No direct communication
- No common frame/direction
  \[\leadsto\text{local coordinates}\]
Points

With Byzantine faults (or crash)

Anonymous

No direct communication

No common frame/direction

\[ \rightsquigarrow \text{local coordinates} \]

Limited/unlimited vision? multiplicity?
Points
- With Byzantine faults (or crash)
- Anonymous
- No direct communication
- No common frame/direction
  - \(\leadsto\) 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
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
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
Scheduling of robots is
Scheduling of robots is

Either **ASYNC**: full interleaving

- Most general/realistic but hardest
Scheduling of robots is

Either **ASYNC**: full interleaving
  - Most general/realistic but hardest

or **Same phase for all active robots**
  - Time split into rounds
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
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)
  \(\sim\) full snapshot of the system

- **Observation**: information available to robogams
  (depends on sensors, no identifiers)
  \(\sim\) degraded form of configuration

- **Execution**: a sequence of configurations
  (usually given by the robogram and the demon)
Outline

Motivations

Overview of the Model

The Pactole Formalism

Case Study: Gathering

Pactole in Practice
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
- 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
Main Ingredients:
- **Robogram**
- Demon (scheduler)
- Round
- Execution
- Properties and Proofs
A robogram is simply a function:

\textbf{Definition} \ robogram := \text{observation} \rightarrow \text{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 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) : \( \mathbb{R}^2 \) :=
    barycenter (elements s).
```
A robogram is simply a function:

\textbf{Definition} \ robogram := \text{observation} \rightarrow \text{location}.

\begin{itemize}
  \item We can use all the expressiveness of Coq to define robograms.
  \item They can be extracted to OCaml/Haskell.
  \item We should only use geometric shapes that are invariant by change of frame of reference.
\end{itemize}

+ a technical detail: compatibility with equivalence (Proper)

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

\texttt{(\ast \text{observation} = \text{set of inhabited location} \ast)}

\textbf{Definition} \ convergeR2\_pgm (obs : observation) : \mathbb{R}^2 := barycenter (elements \ s).

\texttt{(\ast \text{observation} = \text{set of inhabited location} \ast)}
Pactole: a Coq Framework for Mobile Robots

Main Ingredients:
- Robogram
- Demon (scheduler)
- Round
- Execution
- Properties and Proofs

Very Parametric (but still useful):
- Space
- State of Robots (memory, battery level, etc.)
- Sensors
- Environment (adversary)

How states are updated during the move phase

Stream.t configuration

observation $\rightarrow$ 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

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

1. Pick which robots are activated
2. Decide how to update inactive robots
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
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**: 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: 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)
Definition \( \text{round} \ (r : \text{robogram}) \ (da : \text{demonic\_action}) \ (\text{config} : \text{configuration}) \) : \text{configuration} :=
\text{fun} \ id \ \Rightarrow \ (\ast \ \text{for a given robot, we compute the new state} \ast)
The Core of Pactole: the Round Function

**Definition** \( \text{round} \) \((r : \text{robogram}) (da : \text{demonic\_action}) (config : \text{configuration})\)

: configuration :=

fun id ⇒
  if da.(activate) id
  then
  else inactive config id (da.(choose_inactive) config id).

(* for a given robot, we compute the new state *)

(* first see whether the robot is activated *)

Definition round (r : robogram) (da : demonic_action) (config : configuration) : configuration :=
fun id =>
  (* for a given robot, we compute the new state *)
  (* first see whether the robot is activated *)
  if da.(activate) id then
    match id with
    | Byz b => da.(relocate_byz) config b (* byzantine robots *)
    | Good g =>
  else inactive config id (da.(choose_inactive) config id).
Definition \( \text{round} \) (\( r : \text{robogram} \)) (\( \text{da : demonic\_action} \)) (\( \text{config : configuration} \)) : configuration :=

\[
\text{fun id ⇒}
\]

\[
\text{if da.(activate) id then}
\]

\[
\text{match id with}
\]

\[
| \text{Byz b} \Rightarrow \text{da.(relocate\_byz) config b (* byzantine robots *)}
\]

\[
| \text{Good g} \Rightarrow
\]

\[
(* \text{change the frame of reference *})
\]

\[
\text{let frame\_choice := da.(change\_frame) config g in}
\]

\[
\text{let new\_frame := frame\_choice\_bijection frame\_choice in}
\]

\[
\text{let local\_config := map\_config (lift new\_frame ( ... )) config in}
\]

\[
\text{let local\_state := local\_config (Good g) in}
\]

\[
\text{else inactive config id (da.(choose\_inactive) config id ).}
\]
Definition \( \text{round} \ (r : \text{robogram}) \ (da : \text{demonic\_action}) \ (\text{config} : \text{configuration}) \): \text{configuration} :=

\[
\text{fun \ id \ ⇒ } \begin{cases} 
    \text{if } da.(\text{activate}) \ \text{id} & \text{(** for a given robot, we compute the new state **)} \\
    \text{then} & \text{(** first see whether the robot is activated **)} \\
    \text{match id with} & \\
    | \text{Byz} \ b & \text{da.} (\text{relocate\_byz}) \ \text{config} \ b \ (** \text{byzantine robots **)} \\
    | \text{Good} \ g & \text{(** change the frame of reference **)} \\
    & \text{let} \ \text{frame\_choice} := \text{da.} (\text{change\_frame}) \ \text{config} \ g \ \text{in} \\
    & \text{let} \ \text{new\_frame} := \text{frame\_choice\_bijection} \ \text{frame\_choice} \ \text{in} \\
    & \text{let} \ \text{local\_config} := \text{map\_config} (\text{lift new\_frame} (\ldots)) \ \text{config} \ \text{in} \\
    & \text{let} \ \text{local\_state} := \text{local\_config} (\text{Good} \ g) \ \text{in} \\
    & \text{(** compute the observation **)} \\
    & \text{let} \ \text{obs} := \text{obs\_from\_config} \ \text{local\_config} \ \text{local\_state} \ \text{in} \\
    \end{cases}
\]

\begin{cases} 
    \text{else} \ \text{inactive} \ \text{config} \ \text{id} (\text{da.} (\text{choose\_inactive}) \ \text{config} \ \text{id}) .
\end{cases}
Definition \( \text{round} \ (r \ : \ \text{robogram}) \ (da \ : \ \text{demonic\_action}) \ (\text{config} \ : \ \text{configuration}) \) :

\[
\begin{align*}
\text{fun} \ id \ \Rightarrow \ & \text{(* for a given robot, we compute the new state *)} \\
\text{if} \ da.(\text{activate}) \ id \ & \text{(* first see whether the robot is activated *)} \\
\text{then} \ & \\
\text{match} \ id \ \text{with} \ & \\
| \text{Byz} \ b \ \Rightarrow \ & da.(\text{relocate\_byz}) \ \text{config} \ b \ (* \ \text{byzantine robots} *) \\
| \text{Good} \ g \ \Rightarrow \ & \text{(* change the frame of reference *)} \\
& \text{let} \ \text{frame\_choice} := da.(\text{change\_frame}) \ \text{config} \ g \ \text{in} \\
& \text{let} \ \text{new\_frame} := \text{frame\_choice\_bijection} \ \text{frame\_choice} \ \text{in} \\
& \text{let} \ \text{local\_config} := \text{map\_config} \ (\text{lift} \ \text{new\_frame} \ (\ldots)) \ \text{config} \ \text{in} \\
& \text{let} \ \text{local\_state} := \text{local\_config} \ (\text{Good} \ g) \ \text{in} \\
& \text{(* compute the observation *)} \\
& \text{let} \ \text{obs} := \text{obs\_from\_config} \ \text{local\_config} \ \text{local\_state} \ \text{in} \\
& \text{(* apply } r \text{ on observation *)} \\
& \text{let} \ \text{decision} := r \ \text{obs} \ \text{in} \\
\text{else} \ \text{inactive} \ \text{config} \ id \ (da.(\text{choose\_inactive}) \ \text{config} \ id). \\
\end{align*}
\]
The Core of Pactole: the Round Function

**Definition** \( \text{round} \ (r \ : \ \text{robogram}) \ (da \ : \ \text{demonic\_action}) \ (\text{config} \ : \ \text{configuration}) \) :=

\[
\text{fun \ id} \Rightarrow \begin{cases} 
(* \text{for a given robot, we compute the new state} *) \\
(* \text{first see whether the robot is activated} *) \\
\text{if} \ da.(\text{activate}) \ id \\
\text{then} \\
\text{match id with} \\
| \text{Byz} \ b \Rightarrow \ da.(\text{relocate\_byz}) \ \text{config} \ b \ (* \text{byzantine robots} *) \\
| \text{Good} \ g \Rightarrow \\
(* \text{change the frame of reference} *) \\
\text{let frame\_choice} := da.(\text{change\_frame}) \ \text{config} \ g \ \text{in} \\
\text{let new\_frame} := \text{frame\_choice\_bijection} \ \text{frame\_choice} \ \text{in} \\
\text{let local\_config} := \text{map\_config} \ (\text{lift new\_frame} \ (\ldots)) \ \text{config} \ \text{in} \\
\text{let local\_state} := \text{local\_config} \ (\text{Good} \ g) \ \text{in} \\
(* \text{compute the observation} *) \\
\text{let obs} := \text{obs\_from\_config} \ \text{local\_config} \ \text{local\_state} \ \text{in} \\
(* \text{apply} \ r \ \text{on observation} *) \\
\text{let decision} := r \ \text{obs} \ \text{in} \\
(* \text{the demon chooses how to perform the state update} *) \\
\text{let choice} := da.(\text{choose\_update}) \ \text{local\_config} \ g \ \text{decision} \ \text{in} \\
\end{cases}
\text{else inactive \ config \ id} \ (da.(\text{choose\_inactive}) \ \text{config} \ \text{id}).
\]
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
        then
            (* first see whether the robot is activated *)
            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
                else inactive config id (da.(choose_inactive) config id).
```
Definition \( \text{round} \) \((r : \text{robogram}) \ (da : \text{demonic\_action}) \ (\text{config} : \text{configuration}) \) : \text{configuration} :=
\[
\begin{align*}
\text{fun} & \ \text{id} \ \Rightarrow \ \\
& \begin{cases} \\
\text{if} \ \text{da.(activate) id} \ \text{then} \\
\text{match} \ \text{id} \ \text{with} \\
\text{| Byz} \ b & \Rightarrow \ \text{da.(relocate\_byz) config b} \ (\text{* byzantine robots *}) \\
\text{| Good} \ g & \Rightarrow \\
\text{match} \ \text{frame\_choice := da.(change\_frame) config g} \ \text{in} \\
\text{| new\_frame := frame\_choice\_bijection frame\_choice in} \\
\text{| local\_config := map\_config (lift new\_frame ( ... )) config in} \\
\text{| local\_state := local\_config (Good g) in} \ (\text{* compute the observation *}) \\
\text{| obs := obs\_from\_config local\_config local\_state in} \ (\text{* apply } r \ \text{on observation *}) \\
\text{| decision := r obs in} \ (\text{* the demon chooses how to perform the state update *}) \\
\text{| choice := da.(choose\_update) local\_config g decision in} \ (\text{* the actual update of the robot state is performed by the update function *}) \\
\text{| new\_local\_state := update local\_config g frame\_choice decision choice in} \ (\text{* return to the global frame of reference *}) \\
\end{cases} \\
\text{else inactive config id (da.(choose\_inactive) config id) .}
\end{align*}
\]
(** Update the state of good robots in the move phase *)

choose_update : configuration \rightarrow G \rightarrow location \rightarrow 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: 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 → location
to robogam : observation → 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 \texttt{robogam : observation $\rightarrow$ location}

to \texttt{robogam : observation $\rightarrow$ Trobot}

This way, we can model:

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

Don’t forget to also change

\begin{verbatim}
choose_update : configuration $\rightarrow$ G $\rightarrow$ Trobot $\rightarrow$ Tactive;
update : configuration $\rightarrow$ G $\rightarrow$ Trobot $\rightarrow$ Tactive $\rightarrow$ info;
\end{verbatim}
Main Ingredients:
- Robogram
- Demon (scheduler)
- Round
- Execution
- Properties and Proofs

observation $\rightarrow$ Trobot
Execution

Execution = infinite sequence of configurations
\[\rightsquigarrow\] a stream of configurations

**Definition** execution := Stream.t configuration.
Execution

Execution = infinite sequence of configurations
⇝ a stream of configurations

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

How to build streams?
Execution

Execution = infinite sequence of configurations
⇝ 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.
Main Ingredients:
- Robogram
- Demon (scheduler)
- Round
- Execution
- Properties and Proofs

Observation $\rightarrow$ Trobot

Stream.t configuration
We can use all of Coq (inductive/coinductive, higher-order, etc.)

⇒ Follow the mathematical definition
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:

- \( P : \text{configuration} \rightarrow \text{Prop} \)
- \( P : \text{execution} \rightarrow \text{Prop} \)

- \( \text{Stream.instant} \ P \):
- \( \text{Stream.next} \ P \):
- \( \text{Stream.forever} \ P \):
- \( \text{Stream.eventually} \ P \):
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:

- \( P : \text{configuration} \rightarrow \text{Prop} \)
- \( P : \text{execution} \rightarrow \text{Prop} \)
- \( \text{Stream.instant } P : \)
- \( \text{Stream.next } P : \)
- \( \text{Stream.forever } P : \)
- \( \text{Stream.eventually } P : \)

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.

<table>
<thead>
<tr>
<th align="left">Definition gathered_at (pt : loc) (config : configuration) :=</th>
</tr>
</thead>
<tbody>
<tr>
<td align="left">∀ g, get_location (config (Good g)) = pt.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th align="left">Definition Gather (pt : loc) (e : execution) : Prop :=</th>
</tr>
</thead>
<tbody>
<tr>
<td align="left">Stream.forever (Stream.instant (gathered_at pt)) e.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th align="left">Definition WillGather (pt : loc) (e : execution) : Prop :=</th>
</tr>
</thead>
<tbody>
<tr>
<td align="left">Stream.eventually (Gather pt) e.</td>
</tr>
</tbody>
</table>
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.

\[
\begin{align*}
\text{(All good robots are at the same location pt (exactly).)} & \quad \forall g, \text{get\_location}(\text{config (Good g)}) = pt. \\
\text{(At all rounds of the execution e, robots are gathered at pt. *)} & \quad \text{Gather (pt : loc) (e : execution) : Prop := Stream.forever (Stream.instant (gathered\_at pt)) e.} \\
\text{(The (infinite) execution e is eventually Gathered. *)} & \quad \text{WillGather (pt : loc) (e : execution) : Prop := Stream.eventually (Gather pt) e.}
\end{align*}
\]
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

<table>
<thead>
<tr>
<th><strong>Global View</strong> (demon)</th>
<th><strong>Local View</strong> (robots)</th>
</tr>
</thead>
<tbody>
<tr>
<td>absolute location</td>
<td>local frame of reference</td>
</tr>
<tr>
<td>robots: Byzantine or not (B / G)</td>
<td>indistinguishable robots</td>
</tr>
<tr>
<td>identifiers ident = G + B</td>
<td></td>
</tr>
<tr>
<td>configuration = ident → location</td>
<td>local configuration</td>
</tr>
<tr>
<td></td>
<td>observation (abstract type)</td>
</tr>
<tr>
<td></td>
<td>robogram</td>
</tr>
<tr>
<td></td>
<td>: observation → Trobot</td>
</tr>
<tr>
<td>round r d : config → config</td>
<td></td>
</tr>
<tr>
<td>execution = Stream.t configuration</td>
<td></td>
</tr>
</tbody>
</table>
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 : \( \forall d \text{ config}, \text{round } r \ d \text{ config} == ... \)
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** \( \text{round\_simplify} : \forall d \text{ config}, \text{round} r d \text{ config} == ... \)
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
We can easily formalize what a universal algorithm is:

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

**Lemma impossibility**: \( \forall r, \exists d, \forall config, \) invalid config \( \rightarrow \neg \text{good\_execution (execute r d config).} \)

**Lemma correctness r**: \( \forall d, \forall config, \) \( \neg \) invalid config \( \rightarrow \) good\_execution (execute r d config).
Conclusion About Pactole

⊕ Designed for mobile robot networks

⊕ Ease of use for specification

⊕ Broadly applicable

แขนกิจ บรรณาธิการ
Conclusion About Pactole

⊗ Designed for mobile robot networks
  ⊗ cycle built-in
  ⊗ Other features are possible

⊗ Ease of use for specification

⊗ Broadly applicable

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

⊕ Designed for mobile robot networks
   ➤ 🕒 📅 🧵 cycle built-in
   ➤ Other features are possible

⊕ Ease of use for specification
   ➤ Expressive logic
   ➤ Maths can be directly expressed
   ➤ Principled bottom-up approach

⊕ Broadly applicable

memory, battery, ...
Conclusion About Pactole

⊕ Designed for mobile robot networks
  ➤ cycle built-in
  ➤ Other features are possible

⊕ Ease of use for specification
  ➤ Expressive logic
  ➤ Maths can be directly expressed
  ➤ Principled bottom-up approach

⊕ Broadly applicable
  ➤ Highly parametric
  ➤ Very expressive
  ➤ Common base of definition (no more mismatches)

⇒ A junction point for several formal results

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

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

Motivations

Overview of the Model

The Pactole Formalism

Case Study: Gathering

Pactole in Practice
Gathering

<table>
<thead>
<tr>
<th>Objective</th>
</tr>
</thead>
<tbody>
<tr>
<td>Have all (non byzantine) robots reach the same location in finite time and then stay there.</td>
</tr>
</tbody>
</table>
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?

Theorem [Suzuki & Yamashita 99]

Gathering is impossible even for 2 robots only.
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:

---

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

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 (\(Q\) or \(R\))
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:

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:

- \( \text{even number of robots} \)
- Type of line (\( O \) or \( R \))
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 \(\mathbb{R}\))
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?
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 \geq 3

Definition \text{solGathering} (r : \text{robogram}) :=
\forall d : \text{demon}, \text{SSYNC} d \rightarrow \text{Fair} d \rightarrow
\forall \text{config}, \neg \text{bivalent conf} \rightarrow \exists pt, \text{WillGather} pt (\text{execute} r d \text{config}).
Gathering Algo

Careful to the bivalent config!

1. If there is a (unique) majority tower, go there.

2. If 3 equidistant towers, move extreme robots to the middle.
Gathering Algo

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

Careful to the bivalent config!
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.

Careful to the bivalent config!
Gathering Algo

Careful to the bivalent config!

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!

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!

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.
3 configurations to consider:
3 configurations to consider:

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

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

⟹ Just wait long enough
Why Does It Work? SSYNC Case

Idea: adapt FSYNC proofs

Issue: Movement are not done in a single step
→ 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

→ Just wait long enough

And formally?
Idea: adapt FSYNC proofs

Issue: Movement are not done in a single step
  \[\Rightarrow\] 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

\[\Rightarrow\] Just wait long enough

And formally?
Separate proofs for correctness and termination

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

- non bivalent + non gathered $\iff$ a robot should move
- OK by contrapositive
Correctness and Termination

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

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

Termination: there is a result

- lexicographic order on configurations
- movement $\Rightarrow$ smaller configuration
Main Steps of the Coq Formalization

1. Ability to detect majority towers  \( \text{observation} = \text{multisets} \)
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

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

Problem in the last step!
Switch « middle » for « center of the smallest enclosing circle ».
Switch « middle » for « center of the smallest enclosing circle ».
Switch « middle » for « center of the smallest enclosing circle ».
Switch "middle" for "center of the smallest enclosing circle".
Switch "middle" for "center of the smallest enclosing circle".
Switch « middle » for « center of the smallest enclosing circle ».
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.
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 \textbf{backtrack} on the two phases of the algorithm.
It is possible to backtrack on the two phases of the algorithm.
It is possible to backtrack on the two phases of the algorithm.
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
  ~⇒ big cases of the proof
Problem in 2D Solved

Key Ideas

- #robots on the circle decreases → big cases of the proof

n/a Majority

2. Diameter

3. Triangle
   - scalene, isosceles
   - equilateral

4+. Generic

Most cases = same as 1D
Key Ideas

- #robots on the circle decreases $\Rightarrow$ 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
Key Ideas

- #robots on the circle decreases $\leadsto$ 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 \implies 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
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

Motivations

Overview of the Model

The Pactole Formalism

Case Study: Gathering

Pactole in Practice
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/~rieogl/teaching/package.tgz

▶ Start compiling Pactole:
  
  Use make at the root of the package

  ≈ 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 **Instance**s
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

**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
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 \( n : \text{nat} \).
Hypothesis \( \text{n\_non\_0 : n \neq 0.} \)
Instance MyRobots : Names := Robots (2 * n) 0.
Instance XXX : State info := XXX.

Instance Info : State location := OnlyLocation _.
Instance XXX : State info := XXX.

Instance Info : State location := OnlyLocation _.

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

    ...  }.
**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;

  ... }

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 ;

  ... }.
Instance XXX : State info := XXX.

Instance Info : State location := OnlyLocation _.

Class State \{Loc : Location\} info := {
  get_location : info \rightarrow 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 \rightarrow location) \rightarrow Prop;
  lift : \forall f, \text{precondition } f \rightarrow info \rightarrow info;
  (** Properties (compatibility, ...) *)
  lift_id : \forall Pid, \text{@lift } id Pid == id;
  get_location_lift : \forall f (Pf : \text{precondition } f) state,
    \text{get}_{location} (\text{@lift } f Pf state) == f (\text{get}_{location} state);
  ...
}. 
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 *)
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 *)
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 ⇒ 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 *)
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 *)
Instance Update : RigidSetting .
Proof. split . now intros . Qed .
Another Example: Gathering/Definitions.v

Require Export Pactole.Setting.

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
(* All good robots are at the same location \[pt\] (exactly). *)
Definition gathered_at (pt : location) (config : configuration) :=
\( \forall g, \text{get} \_\text{location} (\text{config} (\text{Good } g)) = pt \).

(* 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) := \( \exists pt, \text{WillGather } pt \ e \).
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).
Definition visited pt config := 
  \exists 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 := 
  (\forall pt, will_be_visited pt e) \land will_stop e.
Definition is_solution (r : robogram) := 
  \forall d config, Fair d \rightarrow Explore_and_Stop (execute r d config).
**Perpetual exploration:**
each location is visited infinitely often.

**Definition**
\[
\text{perpetual\_exploration } e := \\
\forall pt, \text{Stream.forever } (\text{Stream.eventually } (\text{Stream.instant } (\text{visited } pt))) e.
\]

**Safety:** stay outside of a given set [danger] of states.

**Definition**
\[
\text{safe\_config } (\text{danger} : \text{location} \to \text{Prop}) \text{ config } := \\
\forall g, \neg \text{danger} (\text{config } (\text{Good } g)).
\]

**Definition**
\[
\text{safe } \text{danger } e := \\
\text{Stream.forever } (\text{Stream.instant } (\text{safe\_config } \text{danger})) e.
\]
Solution: Until + FSYNC + Centralized

Inductive until \((P \ Q : t \ A \rightarrow \text{Prop}) \ (s : t \ A) : \text{Prop} := \)
\[
| \text{NotYet} : P \ s \rightarrow \text{until} \ P \ Q \ (\text{tl} \ s) \rightarrow \text{until} \ P \ Q \ s \\
| \text{YesNow} : Q \ s \rightarrow \text{until} \ P \ Q \ s.
\]

Definition \(\text{weak}_\text{until} \ P \ Q \ s := \text{Stream.forever} \ P \ s \ \lor \ \text{until} \ P \ Q \ s.\)

Definition \(\text{FSYNC}_\text{da} \ da \ da : \text{Prop} := \)
\[\forall \ \text{config} \ g, \ \text{activate} \ da \ \text{config} \ g = \text{true}.\]

Definition \(\text{FullySynchronous} : \text{demon} \rightarrow \text{Prop} := \)
\[\text{Stream.forever} \ (\text{Stream.instant} \ \text{FSYNC}_\text{da}).\]

Definition \(\text{centralized}_\text{da} \ da := \)
\[\forall \ \text{id1 id2}, \ \text{activate} \ da \ \text{id1} = \text{true} \rightarrow \]
\[\text{activate} \ da \ \text{id2} = \text{true} \rightarrow \ \text{id1} = \text{id2}.\]

Definition \(\text{centralized} \ (d : \text{demon}) := \)
\[\text{Stream.forever} \ (\text{Stream.instant} \ \text{centralized}_\text{da}) \ d.\]
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 _ \Rightarrow True ;
    lift := fun f info \Rightarrow (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.
Motivations

Overview of the Model

The Pactole Formalism

Case Study: Gathering
  Impossibility of gathering on a line
  Gathering on a line
  Going to 2D

Pactole in Practice