Computer Aided Formal Design of Swarm Robotics Algorithms

Thibaut Balabonski, Pierre Courtieu, Robin Pelle, Lionel Rieg, Sébastien Tixeuil, Xavier Urbain

Computer Aided Formal Design of Swarm Robotics Algorithms *

Thibaut Balabonski[1], Pierre Courtieu[2], Robin Pelle[1], Lionel Rieg[3], Sébastien Tixeuil[4], and Xavier Urbain[5]

[1]Université Paris-Saclay, CNRS, LMF
[2]Cédric, Conservatoire des Artts et Métiers, Paris
[3]VERIMAG, UMR 5160, Grenoble INP, Univ. Grenoble Alpes
[4]Sorbonne University, CNRS, LIP6
[5]Université Claude Bernard Lyon 1, LIRIS UMR5205

Abstract

Previous works on formally studying mobile robotic swarms consider necessary and sufficient system hypotheses enabling to solve theoretical benchmark problems (geometric pattern formation, gathering, scattering, etc.).

We argue that formal methods can also help in the early stage of mobile robotic swarms protocol design, to obtain protocols that are correct-by-design, even for problems arising from real-world use cases, not previously studied theoretically.

Our position is supported by a concrete case study. Starting from a real-world case scenario, we jointly design the formal problem specification, a family of protocols that are able to solve the problem, and their corresponding proof of correctness, all expressed with the same formal framework. The concrete framework we use for our development is the PACTOLE library based on the COQ proof assistant.

1 Introduction

1.1 Context

Swarm robotics envisions groups of mobile robots self-organizing and cooperating toward the resolution of common objectives, such as patrolling, exploring and mapping disaster areas, constructing ad hoc mobile communication infrastructures to enable communication with rescue teams, etc. As several of those applications are life-critical, the correctness of the deployed protocols becomes of paramount importance. In turn, correctness reasoning about autonomous moving and computing entities that collaborate to achieve a global objective in a setting where unpredictable hazards may occur is complex and error prone. A first step into more formal reasoning is to use a sound *mathematical model*.

Suzuki & Yamashita [28] introduced such a mathematical model describing the behaviour of robots in this context. The model is targeted at swarms of very weak robots evolving in harsh environments. At its core, the model simply commands individual robots to repetitively *observe* their environment before *computing* a path of actions to pursue and acting on it, usually by *moving* to a specific location. Three different levels of synchronization have been commonly considered. The fully-synchronous (FSYNC) case [28] ensures each phase of each cycle is performed simultaneously by all robots. The semi-synchronous (SSYNC) case [28] considers that time is discretized into rounds, and that in each round an arbitrary yet non-empty subset of the robots are active. Finally, the asynchronous (ASYNC) case [19] allows arbitrary delays among the Look, Compute and Move phases, and the movement itself may take an arbitrary amount of time. The Look-Compute-Move model received a considerable amount of attention from the Distributed Computing community,[1] yielding a large variety of submodels induced by refined system assumptions. Such submodels were typically used to assess the solvability of a certain

*This work was partially supported by Project SAPPORO of the French National Research Agency (ANR) under the reference 2019-CE25-0005-1.

[1]Yamashita received the "Prize for Innovation in Distributed Computing" for his seminal work on this model.

task assuming certain system hypotheses. As such, the Distributed Computing literature about mobile robots so far can be seen as *computability-oriented*.

Alas, the various submodels make it extremely tedious to check whether a particular property of a robot protocol holds in a particular setting. Furthermore, these variants do not behave well regarding proof reusability: checking that a property holding in a given setting also holds in another setting that is not strictly contained in the former often amounts to developing a completely new proof, regardless of the proof arguments similarity. This constitutes a major issue when one investigates the correctness of new solutions or implementations of existing protocols to be used in more realistic execution models. This problem is specially acute because of the great diversity of subtly different models: one may be tempted to simply hand-wave their way around the issue by declaring that the proof in this model is "obviously" also valid in this very close model, even more so as even a careful examination may not always find the most subtle errors. Last but not least, protocols are typically written in an informal high level language: assessing whether they conform to a particular model setting is particularly cumbersome, and may lead to hard to find mismatches. As a result, sustained research efforts were made in the last decade to use *formal methods* in the context of mobile robotic swarms.

1.2 Related works

Formal methods encompass a long-lasting path of research that is meant to overcome errors of human origin. Perhaps the most well known instance in the Distributed Computing community is the *Temporal Logic of Actions* and its companion tools TLA/TLA+ [13, 24]. Though very expressive, TLA is designed for the shared memory and message passing contexts, thus not perfectly suited to studying mobile robotic swarms. *Model-checking* and its powerful automation proved useful to find bugs in existing literature [6, 17, 18], and to assess formally published algorithms [6, 16], in a simpler setting where robots evolve in a *discrete space* where the number of possible positions is finite. Automatic program synthesis (for the problem of perpetual exclusive exploration in a ring-shaped discrete space) is due to Bonnet *et al.* [8], and can be used to obtain automatically algorithms that are "correct-by-design". The approach was refined by Millet *et al.* [25] for the problem of gathering in a discrete ring network. However, those approaches are limited to instances with few robots. Generalizing them to an arbitrary number of robots with similar models is doubtful as Sangnier *et al.* [27] proved that safety and reachability problems are undecidable in the parameterized case. Another limitation of the above approaches is that they *only* consider cases where mobile robots *evolve in a discrete space* (*i.e.*, graph). This limitation is due to the model used, that closely matches the original execution model by Suzuki and Yamashita [28]. As a computer can only model a finite set of locations, a continuous 2D Euclidean space cannot be expressed in this model. Défago et al. [14] used a more abstract model to model-check rendez-vous algorithms in a continuous 2D Euclidean space, however, their model is highly specific to rendez-vous and thus is not as versatile as one could hope, hinting at a more general and systematic technique.

The approach on which we focus in this work is *formal proof*, that is proof development mechanically certified by a proof assistant. Mechanical proof assistants are proof management systems where a user can express data, programs, theorems and proofs. In sharp contrast with automated provers (like model-checkers), they are mostly interactive, and thus require some kind of expertise from their users. Sceptical proof assistants provide an additional guarantee by checking mechanically the soundness of a proof after it has been interactively developed. Formal proof allows for more genericity as this approach is not limited to *particular instances* of algorithms. During the last twenty years, the use of tool-assisted verification has extended to the validation of distributed processes, in contexts such as process algebras [7, 20], symmetric interconnection networks [21], message passing settings [23], and self-stabilization [1, 15], etc. The main approach for mechanized proof dedicated to swarms of mobile entities is so far the PACTOLE[2] framework. Initiated in 2010, The PACTOLE framework enabled the use of high-order logic to certify impossibility results, as well as soundness of protocol, for swarms of autonomous mobile robots. To certify results and to guarantee the soundness of theorems, the proof assistant it uses is COQ. Briefly, COQ is a Curry-Howard-based interactive proof assistant that enjoys a trustworthy kernel. Its base language is a very expressive λ-calculus, the *Calculus of Inductive Constructions* [10], where datatypes, objects, algorithms, theorems and proofs can be expressed in a unified way, as terms. The syntax is close to that of an ML-like programming language, and a proof development consists in trying to build, interactively and using tactics, a λ-term, the type of which corresponds to the theorem to be proven (Curry-Howard style). The small kernel of COQ simply type-checks λ-terms to ensure soundness. Most importantly : *a theorem or a lemma can only be saved/defined in the system if it comes with its type-checked proof.* Designed for mobile entities, and

[2] https://pactole.liris.cnrs.fr

making the most of COQ's assets, PACTOLE allows for working on a given protocol to establish and certify its correctness [4, 12], as well as for quantifying over all protocol so as to prove *impossibility* results [2, 5, 11], with an unspecified number of robots, possibly including a proportion of Byzantine faults, in continuous or discrete spaces. FSYNC/SSYNC and ASYNC modes are all supported, and the framework is expressive enough to state and certify formally results as theoretical as comparisons between demons or models [3].

1.3 Our Contribution

Taking some perspective over aforementioned works mixing formal methods and swarm robotics, one can only notice that the computability-centric approach of the Suzuki and Yamashita model yielded a concentration of efforts towards few benchmark problems that are theoretically interesting (one can get impossibility results or correctness certification) but of little practical relevance, such as perpetual or terminating exploration of a ring-shaped graph, and gathering or concentrating all robots at a particular location.

On the other side, relevant practical problems, such as constructing ad hoc mobile communication infrastructures to enable communication with rescue teams, remain untouched using a formal approach. Yet, their correctness is crucial, and possibly life-critical, so it should be assessed formally and mechanically verified. Overall, for those practical problems, the question is not really to characterize which system hypotheses enable problem solvability, but rather how to design a provably correct solution using hypotheses that correspond to real devices.

This paper is the first step in this direction. In more details, we start from a real-life application scenario to jointly design *(i)* its formal specification, *(ii)* a family of protocols that are able to solve the problem, and *(iii)* their corresponding proof of correctness, all expressed with the same formal framework, PACTOLE. In this process, we illustrate how formal methods and PACTOLE in particular could be used to derive protocols that are correct-by-design before they are deployed to actual devices.

The developments for COQ v8.12 described in this work are available at `https://pactole.liris.cnrs.fr/pub/connecti`

2 Search and Rescue, the life line maintainance problem

One of the most advertised applications of robotic swarms is the Search and Rescue situation: a devastated zone has to be explored to bring assistance to survivors. A particular scenario in that context is the "life line" one where, static communication infrastructures having been destroyed, a robotic swarm establishes a dynamic network between a moving search team and the rescue station, by sending mobile transmitters relays.

The same problematic is faced in the less dramatic context of the pursuit of a hornet with drones whose sensors and transmitters are of limited range [26].

In such situations :

- A mobile target follows an unpredictable path and must always be in sight.
- Mobile robots with limited sensors/transmitters track and follow the target.
- To maintain contact with the base station which might be lost due to range limits, some of the robots act as relays.

The link between the base and the target, the so-called life line, must never be broken. More precisely: the existence of such a life line should be ensured at each and every point in the execution.

As practical applications may be critical, with lives at stake, it is imperative for that invariant to hold, and one needs formal guarantees about it.

2.1 Space

Focusing on the search and rescue scenario, we have drones flying and monitoring a rescue team on the ground. Though the most natural space would be the 3D space, it is enough to consider that the mobile robots are moving on a continuous plane, at a fixed altitude. Thus, we can choose the space to be the 2D Euclidean plane.[3]

In this space, there is a point called the *base* from which robots are launched. The second defining object is the *companion* which moves on the plane and follows the rescue team, regardless of other robots; it has to be connected to the base, one way or another.

[3] This incidentally will allow us to use vertical movement as a way to remove robots from the protocol, see Section 4.2.3.

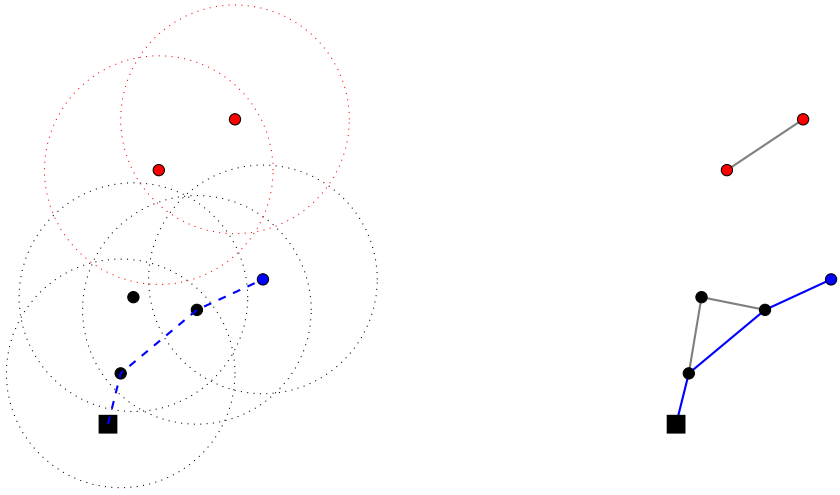


Figure 1: Let us consider in (a) a configuration of the system where ■ is the base, ● is the companion, and where visibility range are denoted as dotted circles. In this situation, the two robots ● cannot detect the others. Nonetheless there is a chain of visibility between the base and the companion, denoted as a dashed blue path. The corresponding visibility graph is in (b), with a suitable path we ask for in blue.

2.2 Basic robot characteristics and sensors

To avoid the problem being trivially solvable, we require the sensors and transmitters to have a limited range. Hence we grant robots the ability to see neighbouring robots up to some *finite range* D_{max}. Indeed, infinite range would make the whole protocol moot as a single robot located at the base would be able to see the target.

Note that we merge vision and transmission capabilities. As a matter of fact relays that do not see others would probably be lost, and the point of a relay is to have neighbours to transmit to.

There is no need to consider multiplicities, as the protocol has to avoid collision: two launched robots will never be at the same location (provided this property holds in the starting configuration).

Similarly, to avoid any trivial counter-example involving a straight line fleeing of the companion following the search team, we constrain the speed of drones to a certain D denoting the bound on the distance that can be travelled within one cycle. Also, we only consider executions where the number of available relays is sufficient.

2.3 Mode of synchronicity, movements, and initial configuration

We may assume the execution to be fully synchronous, with a short time span for each cycle. As pointed out in Section 1.3, our goal is *not* to find minimal synchrony assumptions that allow problem solvability, but rather make sensible and realistic assumptions that permit deployment with actual devices. When robots are homogeneous, they operate at a similar pace, and the fully synchronous model is a sensible choice. Furthermore, keeping the time span short makes the assumption close to what continuous time practitioners envision [9, 22].

For the same reason, we also consider movements to be rigid, that is robots do not start a new cycle before they have reached their computed destination. Since robots always select as a target another robot within the visibility radius, movements are always of limited length.

The fact that the rescue team departs from the base allows us to define a notion of *valid* initial configuration. Trying to solve the problem if the rescue team is already out of reach would make no sense in that context.

2.4 Informal specification of the problem

As it is critical to keep the rescue team connected to the base, we want a formal guarantee that there is always a chain of robots going from the base to the target, such that robots along this chain can relay communication.

Given a configuration of the system (see *e.g.* Figure 1), we say that two robots are *connected* if they are located at most D_{max} apart, that is within visibility range. We formalize this idea by a *visibility graph*, where nodes are

robots and edges relate connected robots. Thus, having a chain of connected robots going from the base to the companion can be expressed as *having a path from the base to the companion in the visibility graph*.

A second objective of our protocol is to avoid collision for obvious reasons, that is, to make sure that no two distinct robots are ever at the same location.

Overall, we end up with:

- A list of hypotheses characterizing our context and the environment, on space, sensors, synchronicity...
- An invariant that, provided there are enough relays to be sent, has to hold at each point of any execution starting from a valid configuration: there is a path in the graph of visibility and there is no collision.

3 A Brief Overview of the PACTOLE Framework

One of the aims of PACTOLE is to stay as simple and as close as possible to the definitions of the robotic swarm community.

Thus, a state of the overall system, a *configuration* is defined as the collection of the states of all robots, conveniently combined into a map from robot names to robot states:

```
Definition configuration := ident → state.
```

A robot *state* can be anything (to accurately describe reality) but must at least contains its *location*, accessible through a function `get_location : state → location`, where the type `location` denotes the space where robots evolve.

An *execution* is an infinite sequence of configurations:

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

Executions are usually built by executing a protocol (called a *robogram*) against an environment, represented as a *demon*, that is, an infinite sequence of decisions called *demonic actions*.

```
Definition demon := Stream.t demonic_action.
```

The robogram represents the Compute part of the Look-Compute-Move cycle. It takes the observation of the robot as input and outputs the action that the robot should perform.

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

This *observation* denotes a degraded version of the configuration centred on the observing robot, depending on its sensors. It is a parameter of the model and its computation from a (local) configuration is performed by an `obs_from_config` function, which hides the information unavailable to robots and takes as input the configuration and the state of the observing robot. This function is specified by a logical formula `obs_is_ok` relating any configuration to its possible observation from any robot state.

```
Parameter observation : Type.
Parameter obs_from_config :
  configuration → state → observation.
Parameter obs_is_ok :
  observation →  configuration → state → Prop.
Parameter obs_from_config_spec : ∀ config st,
  obs_is_ok (obs_from_config config st) config st.
```

To represent the fact that robots observe from a personal point of view, they have their own *frame of reference* that need not be consistent in time or with other robots (other orientation, other scale, other origin, etc.). This frame of reference allows to create a *local configuration* (by opposition to the point of view of the demon denoted as *global*) from which the observation is computed, it depends on the underlying space and it is picked by the demon.

In such an execution, the robogram corresponds to one Look-Compute-Move cycle and the demonic action to the reaction of the environment. Their interaction is described by a function `round` so that the resulting execution is simply repeatedly calling this function with the robogram, the demon and the starting configuration.

The `round` function is the heart of the model, implementing the Look-Compute-Move cycle and computing the configuration obtained after one round. Note that this function is the same for all variants, FSYNC/SSYNC/A-SYNC synchronization, all spaces, all sensors, etc. This is done in the following consecutive steps for each robot name `id`:

1. If the robot `id` is not activated, its state may undergo some change by the `inactive` function to represent an ongoing action or the effect of the environment.
2. If `id` is a byzantine robot, it is relocated by the demonic action `da`.
3. Use the local frame of reference provided by `da` to compute the local configuration.
4. Transform this local configuration into an observation.
5. Apply the robogram on this observation.
6. If moves are flexible, compute the new position of `id` using information given by `da`.
7. Convert the new position from the local frame to the global one.

The full `round` function is:

```
Definition round r da conf : configuration :=
  (* for a given robot, we compute the new configuration *)
 fun id ⇒
  let state := conf id in (* id's state read from conf *)
   (* first see whether the robot is activated *)
  if da.(activate) id
  then match id with
   | Byz b ⇒ (* byzantine robots *)
       da.(relocate_byz) conf b
   | Good g ⇒
     (* change the frame of reference *)
     let frame_choice := da.(change_frame) conf g in
     let new_frame :=
        frame_choice_bijection frame_choice in
     let local_config := map_config new_frame conf 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 the observation *)
     let local_robot_decision := r obs in
     (* the demon chooses how to update the state *)
     let choice :=
        da.(choose_update) local_config g
                           local_robot_decision in
     (* the actual update by the update function *)
     let new_local_state := update local_config g
       frame_choice local_robot_decision choice in
     (* return to the global frame of reference *)
     new_frame ⁻¹ new_local_state
     end
  else inactive conf id (da.(choose_inactive) conf id).
```

4 Co-Designing Specifications, Solutions, and Proofs within the Formal Framework

The first step is to instantiate in the formal framework the assumptions listed in Section 2, as well as the property that we want to be holding throughout a relevant execution.

Those specifications are "refined" until a set of constraints on robots and protocol candidates is formally proven sufficient. Ultimately, our family of protocols will be given by an abstract protocol parameterized by several functions that will be specified enough for the correctness proof to hold.

The final specifications are thus designed together with a solution and its proof.

For the sake of readability we use `cf x` as a shorthand for `cf (Good x)`, i.e. the position of a (non-Byzantine) robot named `x` in a configuration `cf`.[4]

4.1 Specifications

The parameters of the problem are the number n of robots, the visibility radius D_{max} of robots and the maximum distance D they can travel in a round.

```
Parameter n : nat.
Parameters D Dmax : R.
```

Using the number n of robots, we define canonical names to be able to refer to them:

```
Instance Robot_Names : Names := Robots n 0.
```

The `0` represents the absence of Byzantine failures.

The space is the 2D Euclidean space, already predefined in PACTOLE:

```
Instance Loc : Location := make_Location R2.
```

Initially, a robot's state contains only its location, and its observation consists of the set of inhabited locations within its vision range:

```
Instance first_State : State R2 := OnlyLocation.
Instance SetObs : Observation :=
  LimitedSetObervation.limited_set_observation Dmax.
```

The changes of frame of reference allowing to switch between global and local observation (and vice-versa) are similarities (that is, moving the origin around and changing the orientation, chirality, and scale):

```
Instance first_Frame : frame_choice (similarity location)
  := FrameChoiceSimilarity.
```

The FSYNC setting is reflected in an hypothesis made over demonic actions, by requiring property `FSYNC_da` to hold (as will be seen the final statement of the invariant).

Movements are defined as rigid.

```
Instance setting_is_rigid : RigidSetting.
```

As a consequence of our FSYNC and rigid setting with no Byzantine failure, some parameters of the PACTOLE framework are not used and can be set to arbitrary values, namely what happens to a robot whenever it is inactive, how the demon interferes with a robot movement, what happens to robot suffering Byzantine failures, etc. Essentially, one can skip the first two steps of the `round` function described in Section 3 and focus on the Look-Compute-Move cycle.

```
Context {inactive_choice_ila : inactive_choice bool}.
Instance demon_upd : update_choice unit := NoChoice.
```

We say of a robot participating to the task that it is *launched* (by the base station) and *alive*. (These notions will be formally motivated and introduced during the design of the protocol, respectively in Sections 4.2.2 and 4.2.3.) The main goal of any candidate protocol is to maintain the following properties at any moment during the execution:

1. There is no collision between robots, i.e. any two distinct alive robots not at base do not share the same location:

   ```
   Definition no_collision_conf (cf : config) :=
     ∀ g g', g ≠ g'
       → get_launched (cf g) = true
       → get_launched (cf g') = true
   ```

[4] For the sake of clarity, some notations may slightly differ between the actual code and this section, and some irrelevant technical overhead may have been pruned.

```
    → get_alive (cf g) = true
    → get_alive (cf g') = true
    → dist (get_loc (cf g)) (get_loc (cf g')) ≠ 0ℝ.
```

2. There is a sequence of connected robots from the base to the companion (numbered 0, always alive). In other words: for any robot alive (we shall see that there is always some robot alive on base) either it is the companion or it has a visible, active, launched neighbour with a smaller id.

```
Definition path_conf (cf:config) := ∀ g,
 get_alive (cf g) = true
→ get_ident (cf g) = 0
   ∨ ∃ g',
      dist (get_loc (cf g)) (get_loc (cf g')) ≤ Dmax
      ∧ get_alive (cf g') = true
      ∧ get_launched (cf g') = true
      ∧ get_ident (cf g') < get_ident (cf g).
```

Note that this definition implies the existence of a connection from the base to the companion only if there is a robot alive (launched or not) close to the base, i.e. the set of robots waiting to be launched never gets exhausted. The fact that this property always holds during the considered execution is stated as the premise `exists_at_based` below.

The preservation of this invariant is expressed as

```
Definition NoCollAndPath e :=
  Stream.forever (Stream.instant
                        (fun cf ⇒ no_collision_conf cf ∧ path_conf cf)) e.
```

Being assumed that the invariant holds in the initial configuration `config_init` The final lemma will state that the invariant holds forever along any execution of the (abstract and parametric) protocol `rbg_ila`, provided that the set of robots alive at base never gets empty.

```
(* There is a robot not yet launched *)
Definition exists_at_base cf := ∃ g, get_launched (cf g) = false.
(* This property holds forever *)
Definition exists_at_based e :=
  Stream.forever (Stream.instant (exists_at_base)) e.
(* Hypotheses on each demonic action: it is FSYNC
   and local frames are centred on the observing robot *)
Definition da_assumption da := change_frame_origin da ∧ FSYNC_da da.
(* The demon must forever satisfy these properties *)
Definition demon_ILA demon :=
  Stream.forever (Stream.instant da_assumption) demon.
(* Main lemma *)
Lemma validity_conf_init:
  ∀ demon, demon_ILA demon
    → exists_at_based (execute rbg_ila demon config_init)
    → NoCollAndPath (execute rbg_ila demon config_init).
```

We show in the remainder of this work that the aforementioned lemma can in fact be formally proven for any `rbg_ila` fulfilling some sufficient conditions that we discovered in the next section. A concrete suitable protocol is outlined in Section 4.4.

4.2 The March towards a Family of Solutions

We describe in the following how we refine this initial formal model according to the issues we ran into and the hypotheses (whether necessary or by design) we find helpful along the protocol and proof co-development. We update the formal instantiation accordingly. For the sake of clarity, we use the following notation: the successive re-definitions of the parameters are indexed with the refinement step (e.g. $\texttt{State}_2$ is second refinement of the initial default robot state $\texttt{State}_0$). We drop that numbering when we reach the final version of the parameters.

4.2.1 Robot path needs an orientation

Problem: When robots see neither the base nor the companion, for instance when they only see their two neighbours on the life line, they have no way of knowing in which direction the base lies and in which direction the companion lies.

This information is important since the behaviour of relay robots is not symmetric: they should follow the companion as the base is responsible for launching new relay robots whenever necessary. Therefore relay robots need a way to know in which direction lies the companion.

Solution: We provide visible identification and a strict ordering between those robot identifiers. To this goal we use a unique positive integer per robot, and assume that robots will be launched in ascending order. This way, the direction of the companion is given by smaller identifiers. The definition of a state is updated accordingly.

Note that the newly introduced *identifiers* are visible information and are not to be confused with `Names` of robots (Section 4.1), which are constructs internal to the formal framework.

```
Definition identifier := nat.
Definition info₁ := identifier.
Definition State₁ := AddInfo info₁ OnlyLocation.
```

The companion will be given number 0, and at any moment if a robot is launched then all other robots already launched have a smaller id.

4.2.2 Distinguishing robots waiting at base

Problem: What if a robot is located at the base but is already launched? Should this configuration be indistinguishable from the one where it is not yet launched? Obviously not, as collisions may occur in the former but not in the latter.

Solution: The state of a robot should contain a boolean indicating if the robot is "launched" or not. The state should be updated accordingly.

```
Definition launched := bool.
Definition info₂ := identifier*launched.
Definition State₂ := AddInfo info₂ OnlyLocation.
```

4.2.3 Robots may have to withdraw from the system

Problem: When the companion comes closer to the base, the connection line shrinks making the connecting robots possibly unable to stay out of collision risk. Therefore a useful feature of a candidate protocol would be the ability to withdraw from the line.

Solution: We assume available any arbitrary procedure for robot removal, and we add a boolean `alive` to the robot state, making sure that dead (that is, withdrawn) robots are not taken into consideration by alive robots.[5] For simplicity and separation of concern, we make dead robots not even observable by the other robots by adding new constraints to the specification of the observation.

```
Definition alive := bool.
Definition info₃ := identifier*alive*launched.
Definition State₃ := AddInfo info₃ OnlyLocation.
Definition obs_is_ok s config pt:
 get_alive pt = true
 ∧ (∀ l, In l s ⇔ ... ∧ get_alive l ≡ true).
```

[5] For instance, we may make them change altitude making sure they no longer represent a collision risk.

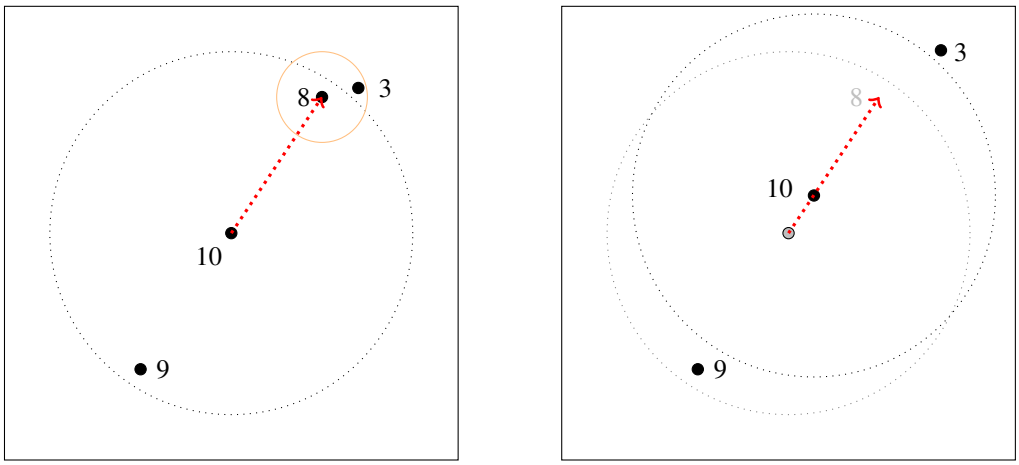


Figure 2: In situation (a), robot 8 will withdraw as it is too close to 3. Robot 10 can see 8 and 9, 3 is out of sight, and it may choose 8 as its target. Situation (b) shows what can happen in this case: 10 moves towards 8 (its previous position is shown in grey), which disappears. With 3 moving further, 10's connection with any suitable robot is lost.

4.2.4 Pruning the space of candidate solutions

Since robots are ordered, and the problem essentially revolves around following some robot of lower identifier, we choose to consider protocols where any given robot makes its decisions depending only on the robots of lower identifier. While this choice is not an answer to any particular problem, it reduces drastically the design space we explore. Once we make this assumption, several formerly open choices become closed. In particular, no robot of small identifier will ever take into account robots of higher identifier nor try to avoid collision with them. That means avoiding collisions and, if necessary, withdrawing, will always be the responsibility of the robots with higher identifiers.

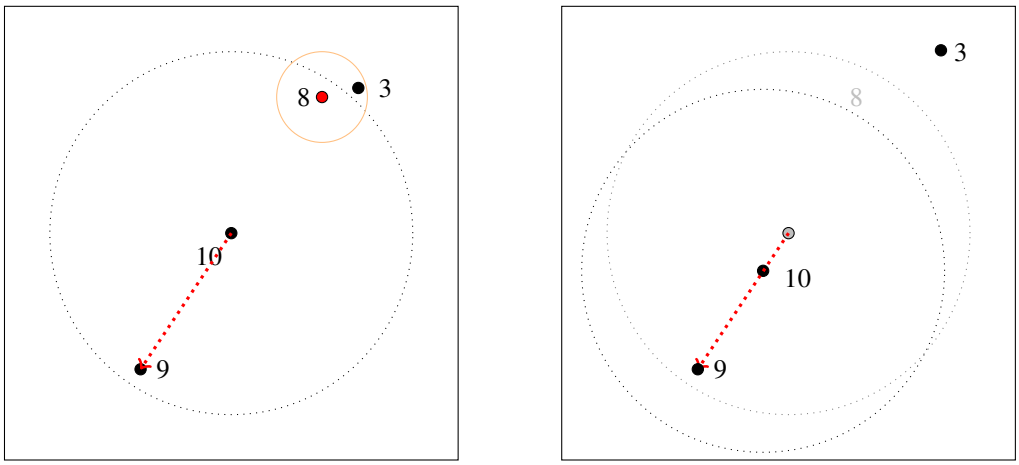
4.2.5 Robots need to warn neighbours before withdrawing

Problem: A robot too close to others shall disappear, but there are cases where a robot immediately behind it may move too far and break the link, precisely *because it tried to avoid that collision*, even if that collision is now impossible due to the robot removal. This situation is illustrated on figure 2. We decide to give robots the ability to warn neighbours about the possibility that it may withdraw in the next round.
Solution: We add a *light* to the robots capabilities. The light of a robot is either `on` or `off`. Any robot that sees another robot sees its light. Light `on` means the robot might be about to withdraw and should not be followed anymore, see Figure 3.

```
Definition light := bool
Definition info4 := identifiant*light*alive*launched.
Definition State4 := AddInfo info4 OnlyLocation.
```

4.2.6 A robot cannot predict the moves of other robots

Problem: Any robot must ensure it avoids collision with other robots and keep in range the neighbour it chooses to follow. Since robots do not know their neighbours views, they cannot predict the moves other robots are about to make. Thus any robot must choose its move so that collisions are avoided and connection with the followed neighbour is preserved for any possible simultaneous move of the other robots.
Solution: Since we assume no robot can travel a distance greater than a constant D in one cycle, we take all decisions with a D margin. This entails defining several zones around a robot using this distance D, summed up



(c) (d)

Figure 3: Situation (c) is similar to Figure 2 (a) as 8 will withdraw but now it *switches its light on* (here in red). As 10 knows that 8 is likely to disappear, it chooses 9 as target, thus keeping a connection with a robot of lower identifier (situation (d)).

in Figure 4.

- A neighbour at distance less than D from the range limit, that is a neighbour at a distance comprised between $D_{max} - D$ and D_{max} might become out of range after its move. If the followed neighbour is in this particular range, a move is necessary to ensure connection. This defines the Pursuit Zone. The distance of pursuit is then defined as $D_p = D_{max} - D$.
- A neighbour at distance less than D might provoke a collision. Since cycles are atomic we do not model how robots move inside a cycle and we can never ensure that robots with crossing trajectories will not collide. Thus the distance D defines a Collision Zone. When two robots are at a distance less than D, either they have to agree on moving apart, or one has to withdraw.
- A neighbour that is not in the Collision Zone, but is still at a distance less than $2D$ might enter the Collision Zone after its move. This range defines the Danger Zone, and letting another robot enter this zone means risking an imminent collision.

The range between distances $2D$ and $D_{max} - D$ gives no particular constraint: neighbours in this Relay Zone are neither too close nor too far.

4.2.7 Robot vision radius must be large enough w.r.t. to robots speed

Problem: A robot may not see far enough to connect without risk. An exemple of this situation is given on Figure 5 where too long a wait to avoid collision risks at launch time leads to a connection break.
Solution: We put a defensive safe limit for launching distance at $D_{max} - 4D$, small enough to avoid losing robots in the example. It must also be large enough to allow for safe launch, at least $3D$. A bound follows on D_{max} that must be strictly greater that $7D$.

4.3 A family of solutions

At this point, we have gathered a few conditions that *appear to solve* the problems we foresee, and should help in maintaining the invariant. As we formalized them in the PACTOLE framework, we may now *prove formally* that indeed any protocol fulfilling these conditions is a solution to our connection problem. We define that way a family of protocols satisfying our needs.

Let us first recap the description of this family of solutions before moving to the proof of its correctness.

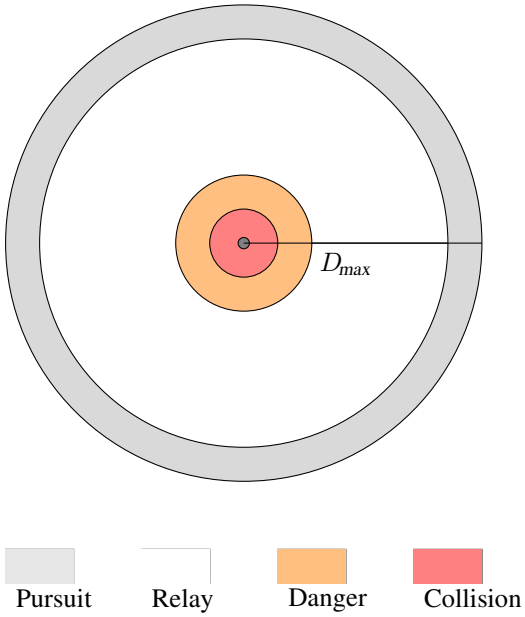


Figure 4: The visible surroundings of a robot can be divided into several zones that characterize the situations it may encounter: need to follow a robot (Pursuit zone, width D), high risk of immediate collision (Collision zone, radius D) or possibly at the next round (Danger zone, width D), and finally just transmission (Relay).

Description of the family of solutions

We describe the family of solutions to the problem by exhibiting a generic protocol `rbg_fnc` parameterized by three auxiliary functions.

```
Context {choose_target : obs_ILA → (R2*ILA)}.
Context {choose_new_pos: obs_ILA → location → location}.
Context {move_to: obs_ILA → location → bool }.

Definition rbg_fnc (s:obs_ILA) : R2*light :=
 (* Chose target and new position accordingly *)
 let target := choose_target s in
 let new_pos := choose_new_pos s (fst target) in
 match move_to s new_pos with                    (* Is this dangerous? *)
 | true ⇒ (new_pos,false)                      (* Safe: move + light off. *)
 | false ⇒ ((0,0), true)                     (* Danger: stay + light on.  *)
 end.
```

The role of `choose_target` is to select among the visible other robots the one that should be followed, i.e. the one we should maintain connection with on the next round. The role of `choose_new_pos` is to decide, given the target computed by `choose_target`, where the robot should move to ensure connection to it. Finally, the role of `move_to` is to decide whether it is dangerous to move to the selected new position. If that is the case, then the protocol decides not to move and warns neighbours that it may withdraw soon by turning on its light (and therefore should preferably not be selected as a target by another robot on the next round).

Our claim is that any three functions verifying the specifications described below will make the protocol achieve our goal. In the following section we give an example of a working instance, thus ensuring that this family is not empty.

The first hypothesis specifies the behaviour of the `choose_target` function.[6] It expresses that its output must:

- be in range, that is be within the input (an observation) to `choose_target`;

[6] In the actual code, each of these five properties is split into its own independent statement for greater flexibility.

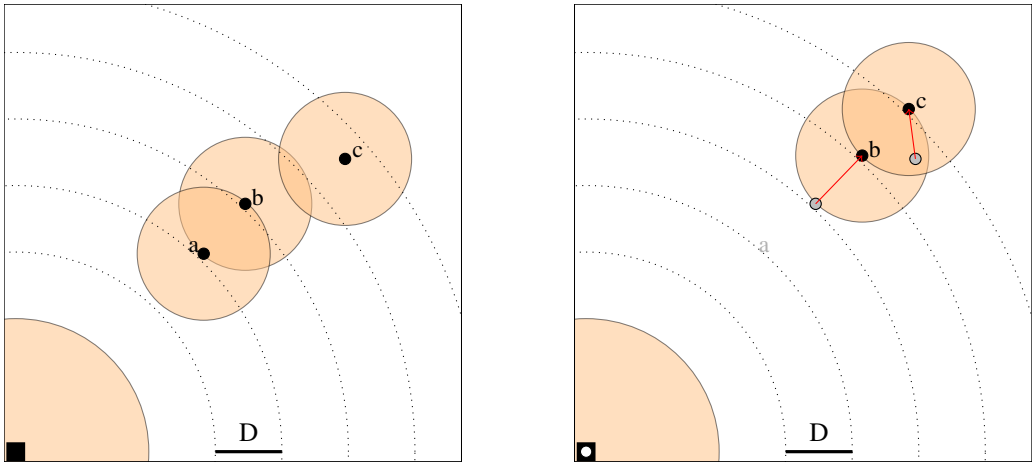


Figure 5: To avoid elimination at launch, the base waits for a robot a to be at more than $D_{max} - 3D$ before launching another robot. On the left, as the base decides to launch a new robot, robot a is too close to robot b and will withdraw. The next round is on the right: b is now too close to c and may withdraw, leaving only c, which moves further away. As the new robot is launched, robot c *should however be still visible from the base*. The new robot ○ was launched too late.

- be alive;
- have a smaller identifier that the observing robot;
- preferably have its light `off`, that is, if the target has its light on, then all robots within range also do;
- preferably select close robots, that is, if the target is in the pursuit zone and has its light on, then all robots within range are also in the pursuit zone.

```
Axiom choose_target_spec : ∀ obs_id local_config,
  let obs := obs_from_config local_config in
  let target := choose_target obs_id obs in
  (* the target must be in range *)
  target ∈ obs
  (* the target must be alive *)
  ∧ get_alive target = true
  (* the target must have a smaller id *)
  ∧ get_ident target < get_ident obs_id
  (* the target must preferably have its light off *)
  ∧ (get_light target = true
     → ∀ id ∈ obs, get_light id = true)
  (* the target must preferably be close *)
  ∧ (get_light target = true
     → dist (0,0) (get_loc target) > Dp
     → ∀ id ∈ obs, dist (0,0) (get_loc elt) > Dp).
```

The second property specifies `choose_new_pos`: the new location is reachable with the speed of the observing robot (it is at most D away from its current location) and not too far from its target (at most D_p away from it, thus out of the pursuit zone).

```
Axiom choose_new_pos_spec : ∀ obs target,
  let new := choose_new_pos obs target in
  dist new target ≤ Dp ∧ dist new (0,0) ≤ D.
```

Finally, the specification of `move_to` is split in two depending on whether movement is possible or not. If it is possible, the chosen location is at least $2D$ away from all other robots.

```
Axiom move_to_true_spec : ∀ obs choice,
  move_to obs choice = true
  → ∀ id, id ∈ obs → dist choice get_loc id > 2*D.
```

If movement is not possible, there is a robot with smaller identifier too close to the chosen location, that is, inside its danger zone (within a disc of radius $2D$).

```
Axiom move_to_false_spec : ∀ state local_config new_loc,
  let obs := obs_from_config local_config state in
  move_to obs new_loc = false
  → ∃ other, other ∈ obs
             ∧ get_ident other < get_ident state
             ∧ dist (get_loc other) new_loc ≤ 2*D.
```

Correctness proof of the family of solutions

The proof amounts to ensuring that the property `NoCollAndPath` given in Section 4.1 is an invariant of the execution. This property combines `no_collision_conf` (the absence of collision) and `path_conf` (which entails the existence of a path from the base to the companion). We only sketch the proofs to convey their insight, for further detail we direct the interested reader to the COQ files.

The absence of collision can be directly proven to be an invariant. Assume that two robots that do not start at the same location collide after their move. Let `id` and `id'` be the identifiers of the robots, with `id` $<$ `id'`. Since the robots are going to collide and none of them can travel more than D, they start at most $2D$ apart. Since `id'` observes a robot with lower identifier `id` in its danger zone, it does not move. Thus, to provoke a collision, `id` and `id'` should start at most D apart. In this case, `id` is actually in the collision zone of `id'`, so `id'` withdraws and there is no collision.

Proving that the existence of a path is invariant is more subtle and we need to introduce three other properties that give more insight into the behaviour of the protocol:

- `executed_means_light_on` expressing that if a robot withdraws at the next round, then it has its light `on` (it is aware that it is in a potentially dangerous situation);
- `executioner_means_light_off` expressing that if a robot withdraws at the next round, then the robot causing this removal has its light `off` (ensuring that it cannot disappear too);
- `exists_at_less_than_Dp` expressing that if all robots of lower identifier in range of an alive robot r have their light `on`, then one of them is not in the pursuit zone of r (at most D_p away from r).

These properties are proven by case analysis over two consecutive rounds, let us call them `cf`, `cf'`.

For the property `executed_means_light_on`, remark that if a robot r alive in `cf` withdraws in `cf'`, it is either because, in `cf'`, there is no robot in range or there is one too close (at most D away). By contradiction, let us assume that r withdraws in `cf'` while not deciding to turn its light `on` in `cf`. Since the light is `on` when `move_to` returns `false`, we know this function returns `true` in `cf`, and r thus performs the move chosen by `choose_new_pos` between `cf` and `cf'`. Thus r cannot lose contact with its target: by the specification of `choose_new_pos` r moved at a distance no greater than D_p to the `cf`-location of its target, that is at a distance no greater than $D_P + D = D_{max}$ to the `cf'`-location of the target. Moreover, r cannot withdraw due to another robot at a distance less than D in `cf'`: by the specification of `choose_new_pos` it moved at a distance more than $2D$ apart from the `cf`-location of any other robot of lower identifier, that is more than $2D - D = D$ apart from the `cf'`-location of any other robot of lower identifier.

For the property `executioner_means_light_off`, note that by the definition of the abstract protocol, a moving robot always has its light `off` (by a simple case analysis on `move_to`).

By the property `executed_means_light_on` a robot r alive in `cf` that withdraws in `cf'` has its light `on` and did not move between `cf` and `cf'`. If r stays alive in `cf` and withdraws in `cf'`, then some other robot r' is at a distance from r that is greater than D in `cf` and at most D in `cf'`. Since r does not move between the configurations, r' necessary does, and thus has its light `off`.

We show now that `exists_at_less_than_Dp` holds for `cf'`. Let us consider an alive robot r in `cf'` such that all robots in range have their light `on`. We want to prove that at least one of them is at most D_p away. Note that as r is alive in `cf'`, there was a robot in range in `cf` and r had a target r' in `cf`. Since r' has its light `on` in `cf'`,

it did not move between cf and cf′. If r' was out of the pursuit zone of r in cf (that is, at most D_p away from r), we can conclude because r either did not move or moved to a position not farther that D_p away from r'. If r' was inside the pursuit zone if r in cf, by the specification of choose_target, so were all robots in range of r. In particular, r could move towards r' as no robot was inside its danger zone and since r' did not move, r and r' are at most D_p away in cf′.

Finally, let us turn to the proof that the property path_conf is an invariant. Let r be a relay robot. We prove that it has a visible, active, launched neighbour with a smaller id. We consider several cases depending on the value of move_to and whether the target of r withdraws.

If move_to is true and r' has its light off, then by property executioner_means_light_off r' cannot withdraw, and cannot get out of range of r since r moves closer to r'.

If move_to is true and r' has its light on, the invariant exists_at_less_than_Dp and the specification of choose_target entail that all robots in range of r have their light on and that r' is out of the pursuit zone. Hence, r' cannot withdraw since all robot close enough to eliminate it have their light on thus cannot cause it to withdraw (by executioner_means_light_off).

If move_to is false, then by move_to_false_spec, there is a robot with smaller id in range of r.

Proof effort

The proof effort for this work is decomposed in setup, specifications and actual proofs.

It consists of **520** lines of *setup* to instantiate the context (definition of robots state, observation, etc.), **540** lines of Coq of *specifications* to describe the problem.

The *proof* part is much more verbose, the table below show how many lines of proof tactics were used for the main invariants.

Stability of main invariants

no_collision_conf	1000
executed_means_light_on	2300
executioner_means_light_off	300
exists_at_less_than_Dp	1300
path_conf	50
Auxiliary results	≈ 5400
Total for proofs	**10500**

4.4 Extracting a Sample Solution

In the previous sections we designed a family of solutions defined by a template robogram function. The template robogram is parameterized by three auxiliary functions, each one being restricted by some axioms. Hence we can obtain a concrete solution by providing for each of the auxiliary functions a concrete definition that is consistent with the corresponding axioms.

It is again possible to develop such a concrete solution in our formal framework. Indeed, the Pactole library already provides some instances of concrete algorithms proven correct for other problems [4, 12]. However, if the axioms are simple enough one can also consider that most of the complex and error-prone reasoning on the model has been taken care of, and that a traditional pen-and-paper check provides a satisfying level of certainty.

In this section we exhibit a concrete solution, for which we check the axioms by hand. Please note that the selected solution has no particular property, and is not supposed to be more efficient, or better in any sense than the other members of our family of solutions. The concrete solution is chosen as one of the most straightforward validations of the axioms. In other words, the solution we present is a naive solution (or at least, it is naive *once the axioms have been designed*). Such a solution provides a sanity check of our axioms: it witnesses that our family of solutions is not empty.

Concretisation of choose_target

We take the point of view of a robot with identifier n, in a configuration satisfying the invariants of the algorithm. Define V the subset of the observed robots that are

- alive
- at a distance at most D_{max}

- with an identifier strictly smaller than n

If there is at least one robot in V whose light is `off`, then select any robot in the subset $V_{\texttt{off}}$ of V containing the robots with lights `off`. For instance, select the robot in $V_{\texttt{off}}$ with minimal identifier. If however all robots in V have their lights on, then select any robot in the subset V_{Dp} of V containing the robots at a distance at most D_p. For instance, select the robot in V_{Dp} with minimal identifier. Otherwise select any robot in V, for instance the robot in V with minimal identifier.

We have to check that this function is defined and satisfies all the axioms. First note that, following the invariant `path_conf`, there is at least one robot that is alive, at a distance at most D_{max} of n and with an identifier strictly smaller than n. Hence the set V is not empty, and our function is defined. Second, the result of this function is a robot of V. As such it is visible, alive and and with an identifier strictly smaller than n (three first parts of axiom `choose_target_spec` are satisfied). Third, the result has its light `off` as soon as there is at least one visible robot with its light `off`: fourth part of `choose_target_spec` is also satisfied. Finally, when there are only robots with their lights on, the result is a robot at a distance at most D_p if there is one: the fifth and final part of axiom `choose_target_spec` is again satisfied. Hence our naive `choose_target` function satisfies all the relevant axioms.

Concretisation of `choose_new_pos`

We take the point of view of a robot with an already chosen target at distance at most D_{max}. One can select the potential move of the robot as follows:

- if the target robot is at a distance greater than D_p, then choose a move of length D toward the target;
- otherwise, choose a null move.

This tentative move aims at a position at distance at most D from the starting position, and at most D_p from the target. It thus complies with the axiom `choose_new_pos_spec`.

Concretisation of `move_to`

We take the point of view of a robot with a potential move already selected, with some observed configuration. Validation of the potential move can be performed as follows:

- if there is any other robot at most $2D$ away from the potential destination, then invalidate the potential move by returning `false`;
- otherwise, validate the potential move by returning `true`.

This function returns `true` only if there is no robot at a distance to the potential destination less or equal to $2D$: the axiom `move_to_Some_zone` is satisfied. Conversely, the function returns `false` only if there is an observable other robot at a distance to the potential destination less or equal to $2D$: the axiom `move_to_None` is also satisfied.

Finally, the three parameter functions `choose_target`, `choose_new_pos`, and `move_to` all satisfy their respective axioms: they define a proper member of our family of solutions, which is not empty.

5 Concluding remarks

In this paper, we demonstrated by example how formal methods, and the PACTOLE framework in particular, can help mobile robotic swarm protocol designers to formally specify, design, and prove their algorithms are correct, balancing expressivity to tackle practically relevant problems, and formality to preserve the mathematical soundness of software developments.

Of course, proving correct algorithms for new problems is only the first step. A natural second step is to ensure the *implementations* of the algorithms maintain the relevant invariants when actually deployed on real devices. We leave this path for future research.

References

[1] K. Altisen, P. Corbineau, and S. Devismes. A framework for certified self-stabilization. In E. Albert and I. Lanese, editors, *Formal Techniques for Distributed Objects, Components, and Systems - 36th IFIP WG 6.1 International Conference, FORTE 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings*, volume 9688 of *Lecture Notes in Computer Science*, pages 36–51. Springer-Verlag, 2016.

[2] C. Auger, Z. Bouzid, P. Courtieu, S. Tixeuil, and X. Urbain. Certified Impossibility Results for Byzantine-Tolerant Mobile Robots. In T. Higashino, Y. Katayama, T. Masuzawa, M. Potop-Butucaru, and M. Yamashita, editors, *Stabilization, Safety, and Security of Distributed Systems - 15th International Symposium (SSS 2013)*, volume 8255 of *Lecture Notes in Computer Science*, pages 178–186, Osaka, Japan, Nov. 2013. Springer-Verlag.

[3] T. Balabonski, P. Courtieu, R. Pelle, L. Rieg, S. Tixeuil, and X. Urbain. Continuous vs. discrete asynchronous moves: A certified approach for mobile robots. In M. F. Atig and A. A. Schwarzmann, editors, *Networked Systems - 7th International Conference, NETYS 2019, Marrakech, Morocco, June 19-21, 2019, Revised Selected Papers*, volume 11704 of *Lecture Notes in Computer Science*, pages 93–109. Springer-Verlag, 2019.

[4] T. Balabonski, A. Delga, L. Rieg, S. Tixeuil, and X. Urbain. Synchronous gathering without multiplicity detection: A certified algorithm. *Theory of Computing Systems*, 2019. https://doi.org/10.1007/s00224-017-9828-z.

[5] T. Balabonski, R. Pelle, L. Rieg, and S. Tixeuil. A foundational framework for certified impossibility results with mobile robots on graphs. In P. Bellavista and V. K. Garg, editors, *Proceedings of the 19th International Conference on Distributed Computing and Networking, ICDCN 2018, Varanasi, India, January 4-7, 2018*, pages 5:1–5:10. ACM, 2018.

[6] B. Bérard, P. Lafourcade, L. Millet, M. Potop-Butucaru, Y. Thierry-Mieg, and S. Tixeuil. Formal verification of mobile robot protocols. *Distributed Computing*, 29(6):459–487, 2016.

[7] M. Bezem, R. Bol, and J. F. Groote. Formalizing Process Algebraic Verifications in the Calculus of Constructions. *Formal Aspects of Computing*, 9:1–48, 1997.

[8] F. Bonnet, X. Défago, F. Petit, M. Potop-Butucaru, and S. Tixeuil. Discovering and assessing fine-grained metrics in robot networks protocols. In *33rd IEEE International Symposium on Reliable Distributed Systems Workshops, SRDS Workshops 2014, Nara, Japan, October 6-9, 2014*, pages 50–59. IEEE, 2014.

[9] J. Castenow, P. Kling, T. Knollmann, and F. M. auf der Heide. A discrete and continuous study of the max-chain-formation problem: Slow down to speed up. In C. Scheideler and M. Spear, editors, *SPAA '20: 32nd ACM Symposium on Parallelism in Algorithms and Architectures, Virtual Event, USA, July 15-17, 2020*, pages 515–517. ACM, 2020.

[10] T. Coquand and C. Paulin-Mohring. Inductively Defined Types. In P. Martin-Löf and G. Mints, editors, *International Conference on Computer Logic (Colog'88)*, volume 417 of *Lecture Notes in Computer Science*, pages 50–66. Springer-Verlag, 1990.

[11] P. Courtieu, L. Rieg, S. Tixeuil, and X. Urbain. Impossibility of Gathering, a Certification. *Information Processing Letters*, 115:447–452, 2015.

[12] P. Courtieu, L. Rieg, S. Tixeuil, and X. Urbain. Certified universal gathering algorithm in $\mathbb{R}^2$ for oblivious mobile robots. In C. Gavoille and D. Ilcinkas, editors, *Distributed Computing - 30th International Symposium, (DISC 2016)*, volume 9888 of *Lecture Notes in Computer Science*, Paris, France, Sept. 2016. Springer-Verlag.

[13] D. Cousineau, D. Doligez, L. Lamport, S. Merz, D. Ricketts, and H. Vanzetto. TLA + Proofs. In D. Giannakopoulou and D. Méry, editors, *FM*, volume 7436 of *Lecture Notes in Computer Science*, pages 147–154, Paris, France, Aug. 2012. Springer-Verlag.

[14] X. Défago, A. Heriban, S. Tixeuil, and K. Wada. Using model checking to formally verify rendezvous algorithms for robots with lights in euclidean space. In *International Symposium on Reliable Distributed Systems, SRDS 2020, Shanghai, China, September 21-24, 2020*, pages 113–122. IEEE, 2020.

[15] Y. Deng and J.-F. Monin. Verifying Self-stabilizing Population Protocols with Coq. In W.-N. Chin and S. Qin, editors, *Third IEEE International Symposium on Theoretical Aspects of Software Engineering (TASE 2009)*, pages 201–208, Tianjin, China, July 2009. IEEE Computer Society.

[16] S. Devismes, A. Lamani, F. Petit, P. Raymond, and S. Tixeuil. Optimal Grid Exploration by Asynchronous Oblivious Robots. In A. W. Richa and C. Scheideler, editors, *Stabilization, Safety, and Security of Distributed Systems - 14th International Symposium (SSS 2012)*, volume 7596 of *Lecture Notes in Computer Science*, pages 64–76, Toronto, Canada, Oct. 2012. Springer-Verlag.

[17] H. T. T. Doan, F. Bonnet, and K. Ogata. Model checking of a mobile robots perpetual exploration algorithm. In S. Liu, Z. Duan, C. Tian, and F. Nagoya, editors, *Structured Object-Oriented Formal Language and Method - 6th International Workshop, SOFL+MSVL 2016, Tokyo, Japan, November 15, 2016, Revised Selected Papers*, volume 10189 of *Lecture Notes in Computer Science*, pages 201–219, 2016.

[18] H. T. T. Doan, F. Bonnet, and K. Ogata. Model checking of robot gathering. In J. Aspnes, A. Bessani, P. Felber, and J. Leitão, editors, *21st International Conference on Principles of Distributed Systems, OPODIS 2017, Lisbon, Portugal, December 18-20, 2017*, volume 95 of *LIPIcs*, pages 12:1–12:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

[19] P. Flocchini, G. Prencipe, N. Santoro, and P. Widmayer. Gathering of asynchronous robots with limited visibility. *Theor. Comput. Sci.*, 337(1-3):147–168, 2005.

[20] W. Fokkink. *Modelling Distributed Systems*. EATCS Texts in Theoretical Computer Science. Springer-Verlag, 2007.

[21] N. Gaspar, L. Henrio, and E. Madelaine. Bringing coq into the world of gcm distributed applications. pages 643–662, 2014.

[22] P. Kling and F. M. auf der Heide. Continuous protocols for swarm robotics. In P. Flocchini, G. Prencipe, and N. Santoro, editors, *Distributed Computing by Mobile Entities, Current Research in Moving and Computing*, volume 11340 of *Lecture Notes in Computer Science*, pages 317–334. Springer, 2019.

[23] P. Küfner, U. Nestmann, and C. Rickmann. Formal Verification of Distributed Algorithms - From Pseudo Code to Checked Proofs. In J. C. M. Baeten, T. Ball, and F. S. de Boer, editors, *IFIP TCS*, volume 7604 of *Lecture Notes in Computer Science*, pages 209–224, Amsterdam, The Netherlands, Sept. 2012. Springer-Verlag.

[24] L. Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, May 1994.

[25] L. Millet, M. Potop-Butucaru, N. Sznajder, and S. Tixeuil. On the synthesis of mobile robots algorithms: The case of ring gathering. In P. Felber and V. K. Garg, editors, *Stabilization, Safety, and Security of Distributed Systems - 16th International Symposium, (SSS 2014)*, volume 8756 of *Lecture Notes in Computer Science*, pages 237–251, Paderborn, Germany, Sept. 2014. Springer-Verlag.

[26] L. Reynaud and I. G. Lassous. Design of a force-based controlled mobility on aerial vehicles for pest management. *Ad-Hoc Networks*, 53:41–52, 2016.

[27] A. Sangnier, N. Sznajder, M. Potop-Butucaru, and S. Tixeuil. Parameterized verification of algorithms for oblivious robots on a ring. *Formal Methods Syst. Des.*, 56(1):55–89, 2020.

[28] I. Suzuki and M. Yamashita. Distributed Anonymous Mobile Robots: Formation of Geometric Patterns. *SIAM Journal of Computing*, 28(4):1347–1363, 1999.