



HAL
open science

Swarms of Mobile Robots: Towards Versatility with Safety

Pierre Courtieu, Lionel Rieg, Sébastien Tixeuil, Xavier Urbain

► **To cite this version:**

Pierre Courtieu, Lionel Rieg, Sébastien Tixeuil, Xavier Urbain. Swarms of Mobile Robots: Towards Versatility with Safety. Leibniz Transactions on Embedded Systems, 2022, Distributed Hybrid Systems, 8 (2), pp.02:1-02:36. 10.4230/LITES.8.2.2 . hal-03901898

HAL Id: hal-03901898

<https://hal.science/hal-03901898v1>

Submitted on 4 Dec 2024


HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Distributed under a Creative Commons Attribution 4.0 International License

Swarms of Mobile Robots: Towards Versatility with Safety

Pierre Courtieu ✉ 

Conservatoire des arts et métiers, Cédric EA 4629, Paris, France

Lionel Rieg ✉

VERIMAG, Grenoble INP – UGA, CNRS UMR 5104, Université Grenoble-Alpes, Saint Martin d’Hères, France

Sébastien Tixeuil ✉ 

Sorbonne University, CNRS, LIP6, Paris, France

Xavier Urbain ✉ 

Université de Lyon, Université Claude Bernard Lyon 1, CNRS, INSA Lyon, LIRIS, UMR 5205, F-69622 Villeurbanne, France

Abstract

We present PACTOLE, a formal framework to design and prove the correctness of protocols (or the impossibility of their existence) that target mobile robotic swarms. Unlike previous approaches, our methodology unifies in a single formalism the execution model, the problem specification, the protocol, and its proof of correctness. The PACTOLE framework makes use of the COQ proof assistant, and is specially targeted at protocol designers and problem

specifiers, so that a common unambiguous language is used from the very early stages of protocol development. We stress the underlying framework design principles to enable high expressivity and modularity, and provide concrete examples about how the PACTOLE framework can be used to tackle actual problems, some previously addressed by the Distributed Computing community, but also new problems, while being certified correct.

2012 ACM Subject Classification Theory of computation → Distributed computing models; Theory of computation → Self-organization; Theory of computation → Program reasoning; Theory of computation → Logic; Software and its engineering → Formal methods

Keywords and Phrases distributed algorithm, mobile autonomous robots, formal proof

Digital Object Identifier 10.4230/LITES.8.2.2

Supplementary Material *Software (Coq Formalization)*: <https://pactole.liris.cnrs.fr>

Funding This work was partially supported by CNRS PEPS DiDaSCaL and ANR project SAPPORO 2019-CE25-0005.

Acknowledgements The authors would like to thank the referees whose comments and suggestions helped improve the presentation of this work.

Received 2020-07-09 **Accepted** 2022-01-28 **Published** 2022-12-07

Editor Alessandro Abate, Uli Fahrenberg, and Martin Fränzle

Special Issue Special Issue on Distributed Hybrid Systems

1 Introduction: low cost and high expectations

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. In many cases, such groups of robots are deployed in adverse environments (e.g. space, deep sea, disaster areas). Thus, a group must be able to self-organize in the absence of any prior infrastructure and ensure dynamic coordination in spite of the presence of faulty robots as well as environmental changes. A faulty robot can stop its execution (crash) or start to behave in an



© Pierre Courtieu, Lionel Rieg, Sébastien Tixeuil, and Xavier Urbain;
licensed under Creative Commons Attribution 4.0 International (CC BY 4.0)
Leibniz Transactions on Embedded Systems, Vol. 8, Issue 2, Article No. 2, pp. 02:1–02:36

Leibniz Transactions on Embedded Systems
LITES Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

arbitrary way either due to some external factors (e.g. electromagnetic fields, attacks) or to some inaccurate information received by its own sensors.

Sending a single complex expensive robot for example to explore a dangerous area is often not the best solution, as some environments are likely to destroy robots in a matter of hours.¹ Instead, using a large number of cheap simple robots one can afford to lose, but that nonetheless are able to coordinate to *globally* solve a given task, is the underlying principle of swarm robotics.

The possibility to have a robot fail being almost certain, the operator has to use cheap robots to form the swarm, implying that very weak individual sensing and actuating capacities are to be privileged. Lowering the individual robots' abilities raises an important theoretical question: which *individual* capacities are necessary (and sufficient) to *collectively* solve a given task? Obviously, robots with more abilities than necessary to solve a task can still solve it. Conversely if a given task solvability requires a particular capacity, and that capacity is unavailable to the robots forming the swarm, no genius algorithm or protocol can come to the rescue.

Another consequence of failure likeliness is that no robot in the swarm should have a *particular* role to assume (e.g. a leader from which other robots wait for orders). Indeed, if the particular robot ceases functioning or starts behaving arbitrarily, the entire swarm fails. Instead, all robots are to be given the same role, and self-organization is to be used to take collective decisions regardless of the current situation.

On the other hand, proving task solvability requires to envision all situations, even the most unlikely ones. A classical setup only considers the system behaviour from a given well-formed initial state. Proofs written in the context of swarm robotics must consider all possible failure occurrences, be they at the individual robot level or induced by a catastrophic change in the environment.

Actual deployment of mobile robotic swarms mandates preliminary theoretical assessments, to ensure the swarm behaves according to its specification, and to assess its practical feasibility with respect to expected completion time and used resources. While the latter is typically quantified through simulations, the former requires a sound mathematical proof of correctness. Most of the literature makes use, for this purpose, of handwritten proofs. As recent findings show [1, 14, 32, 33], handwritten proofs are error-prone, and sometimes erroneous, which may compromise the safety and the correctness of actual deployments. Hence, a recent trend deals with computer-aided proving of important properties for mobile robotic swarms.

Following this trend, our focus in this paper is to propose a unified formal approach that permits us to express both the execution model and its variants, and the property specifications and their proof, relating all that we formally state to the usual model in the Distributed Computing Community. In more detail, our starting point is the model by Suzuki and Yamashita [65] (who was recently awarded the *Prize for Innovation in Distributed Computing*), extended by the many variants the Distributed Computing developed throughout the years [37], unified in a modular formal framework developed in COQ.

Our framework is meant to answer legitimate questions that arise when developing protocols for mobile robotic swarms:

1. Is algorithm A a solution to problem X in model variant Y ?
2. Is problem X solvable using model variant Y ?
3. Which problems are solvable using model variant Y ?
4. What is the weakest model variant that permit to solve problem X ?
5. Does the proof for algorithm A remain valid if we switch model variant Y for model variant Z ?
6. etc.

¹ <https://www.theguardian.com/world/2017/mar/09/fukushima-nuclear-cleanup-falters-six-years-after-tsunami>

The remainder of the paper is organized as follows. Section 2 reviews previous work related to the use of formal methods in the context of Distributed Computing, as well as an introductory example. The PACTOLE library for the COQ proof assistant is presented in Section 3. In Section 4, we review the structure of model variants used by the Distributed Computing community for mobile robotic swarms, while its formalization in our framework is presented in Section 5. Several case studies in Section 6 demonstrate the simplicity and universality of our approach. Finally, concluding remarks are presented in Section 8.

2 Formal approaches and their complementary uses

Motivations

Models of distributed computations are traditionally presented in natural language. But the algorithm, even when presented as pseudo-code [49], cannot be understood without the *precise setting* in which it is executed. *Implicit assumptions* [50] are known as folklore but as properties of distributed algorithms rely on very subtle hypotheses, in most cases a small barely noticeable shift in the statement of these assumptions may induce dramatic changes in an algorithm behaviour. Such shifts can still be found in the literature, and have led to erroneous results being published.

Formal methods tackle this difficulty, most notably with the use of tools providing a mathematical language that is non-ambiguous [62]. Given how challenging the task of establishing correct results in the area of distributed computing is, *the simple fact of using a common non-ambiguous format for definitions constitutes a significant asset.*

So far the most popular formal method in the Distributed Computing community has been *model checking*. Formal methods however encompass a wide variety of other methods. *Formal proof* for instance has little to do with model checking, and in most aspects it can be considered as its dual: formal provers can be applied to almost any domain of mathematics [55, 42, 41, 2, 67] but are poorly automated when dealing with higher order properties, whereas model checkers only apply on decidable (hence less expressive) logics but are highly automated. Model checkers produce counter-examples whereas theorem provers do not (at least systematically), etc.

Model Checkers

The power and elegance of model checking lie in building an *abstraction* of the property to prove, tailored so that its validity can be checked *exhaustively* by an automated tool. The correctness of the abstraction being in general proved on paper.

Model checking has been used with impressive success for distributed protocols, both in proving [14, 34, 35, 54, 51, 43, 28], and disproving [32, 33, 14] their correctness. In some cases, it was possible to go as far as program synthesis [16, 59, 36, 31] (that is, generating algorithms that are correct by design using a computer program). It may however be subject to combinatorial explosion, or become undecidable [4].

Consequently, model checking often deals with *instances* of a problem rather than with its full generality. Parameterized model checking sometimes allows for model checking all instances where an (infinite) parameter varies. For example, Sangnier et al. [64] makes use of Presburger formulae to express mobile robotic swarms operating on a discrete space (a ring of size n , meant to be arbitrary, and a parameter of the model). However, a key result of Sangnier et al. [64] in this context is that non-trivial properties (namely, liveness properties) are undecidable. Those recent findings command studying complementary techniques like formal proofs.

Formal Proofs

The formal proof approach consists in writing mathematical proofs in a fully explicit way, leaving absolutely no reasoning detail hidden or implicit. This is (obviously) a very tedious task, and it cannot really be applied without the help of mechanical *tools*, called *proof assistants*, that provide

- (1) a language for mathematical definitions and properties;
- (2) an interactive system assisting the user in writing all details of the proofs, thus ensuring their correctness *by construction*.

Since the proof system is not bound to be fully automatic, very expressive (undecidable) logics are allowed, making it possible to write virtually any mathematical definition, as witnessed by the wide range of mathematical results that have been proven using these tools [55, 42, 41, 2, 67, 48].

A drawback of being very expressive is an induced lack of automation. Despite the help of a variety of decision procedures for decidable sub-logics of the system, developing proofs in a proof assistant still requires a lot of expertise.

Given its characteristics, one can expect formal proof to be successfully applied in distributed computing, but in a way that is complementary to the model checking approach. Figure 1 gives hints on where a proof assistant and model checking are potentially usable in the everyday life of a researcher in distributed algorithms.

Primarily, it can be used as the *underlying non-ambiguous language* for all *definitions* involved in Distributed Computing: from high-level model specifications to low-level algorithms and all their properties. This is a specific complementary benefit of proof assistants. For instance in our setting it is possible to state and prove properties explicitly quantified over continuous spaces like \mathbb{R}^2 or the type of all protocols (functions over functions on \mathbb{R}^2):

```
∀ r: (robot → ℝ2) → ℝ2, ...
```

or over types populated with infinite objects like demons (infinite streams):

```
∀ d: Stream demonic_action, ...
```

This makes possible to state for instance that a given task is *impossible* to achieve [6, 23, 12] in some model, i.e. that *for all protocols* (even those that cannot be computed with usual operations) there exists an adversary (demon) that will make the protocol fail.

Even if this part is *much* more intricate and needs dedicated expertise, the proof assistant can also be used to *prove* these properties. It is notably more tedious than writing a pen and paper proof because of the required level of details, and it requires expertise in the assistant involved. Expecting an expert in any domain of computer science to become also an expert with a proof assistant it thus somewhat unrealistic at this time. It also misses the point as the fundamental first step consists in providing formal definitions, and not proofs.

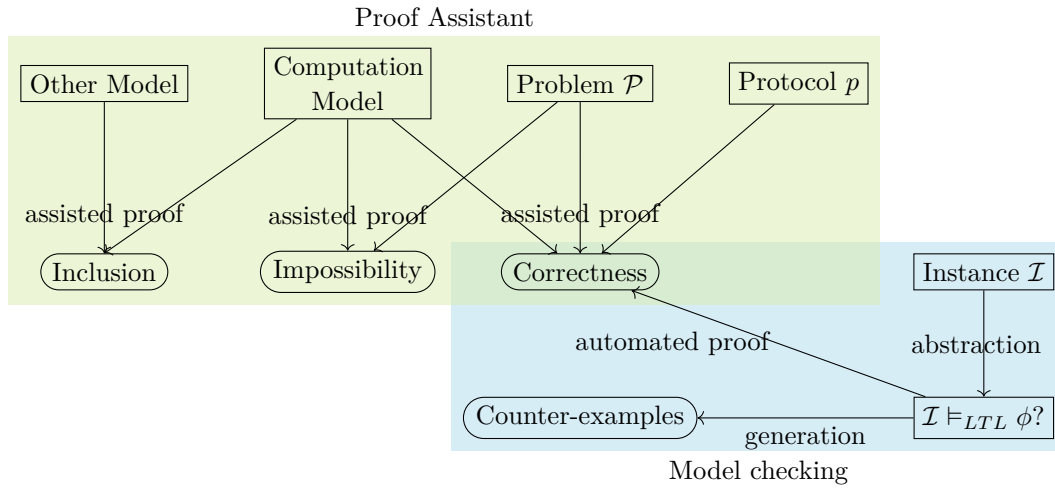
Finally the architecture of proof assistants allows for building large libraries of shared definitions for models, problems, protocols, and theorems, ensuring their mutual consistency, and *reusability*. In the long run this makes possible new and increasingly intricate but sound results.

To this goal, collaborations between experts of distributed algorithms and formal proof are needed and the PACTOLE library is an example of such collaboration, among others [3, 19].

3 The Pactole library for the Coq proof assistant

3.1 The Coq proof assistant

The proof assistant used for this work is COQ [5, 15]. It is based on type theory and its language for definitions and properties is a very rich typed λ -calculus: the calculus of inductive constructions [22, 15].



■ **Figure 1** Potential uses of a proof assistant: Definitions, statements of results and their proofs.

A particularity of proof assistants based on type theory is that the *definition* language is also used to express *proofs*. More precisely it makes use of the Curry-Howard correspondence: types are considered as properties, and a term of a given type is actually a *proof* of this property. Checking correctness of proofs therefore amounts to type-checking λ -terms. Since this is a rather simple task, a small kernel can be written, and proofs accepted by the proof assistant have strong guarantees of correctness.

The syntax of the COQ language is similar to that of a functional programming language *à la* ML. Function types are written in the usual Curryfied way: $A \rightarrow B \rightarrow C$ denotes the type of functions that take a parameter of type A and return a function from B to C , which can also be seen as the type of functions that take an A and a B , and return a C but can be partially applied to their first argument. Applying a function f to an argument a is simply denoted by juxtaposition: $f a$.

Function definitions and type synonyms are introduced with **Definition**. New data types may be defined either by their exhaustive list of constructors with **Inductive** (and the pattern matching of such a type is done by the **match ... with ... end** construction) or as a **Record** whose values are of the form $\{ | f1d1 := va11; f1d2 := va12 \dots | \}$, and fields can be accessed by the usual dot notation. For example: let x be the record $\{ | f1 := a ; f2 := b | \}$, the expression $x.(f1)$ has value a .

A particular construction allows for defining (sub-)types by intention: $\{ x : T \mid P x \}$ represents the type of any element x of type T *paired with* a proof that P holds on x .

Coinductive types (mainly infinite streams in our context) can be easily defined in COQ and coinductive values are introduced by **cofix**.

3.2 Pactole

Developed with and for the COQ proof assistant, PACTOLE is a library gathering definitions and proofs on a variety of models of robot swarms. It implements the generic seminal model by Suzuki and Yamashita [65] presented in details in Section 4.1.

Formally proven results are correct by construction and can therefore be highly trusted and reused. It is worth noting that it is still the responsibility of the experts of a certain domain to check *what* those results are the proof of. It is indeed critically important that the definitions

are scrutinized and validated by the community. The *proofs* themselves, while sometimes worth sketching, need not to be human-checked.² In that respect, a focus in PACTOLE is on *the ease to write and read specifications*.

Designed for robots and in particular agents that are *mobile*, PACTOLE provides a wide range of definitions and proofs, from very high level notions to concrete protocols and their properties:

- Definitions of models, proofs of relations between models (inclusion, equivalence, etc.);
- problem definitions (gathering, exploration, etc.);
- protocol definitions and proofs of correctness;
- proofs of impossibility.

All these notions are inter-dependent. One of the benefits of proposing the widest range of notions of the domain is that they share the same underlying definitions. They are therefore consistent with each other *by construction*.

For example, when dealing with impossibility results for some problem P , and protocols solving P under particular assumptions, sharing the definition of P ensure that these results are correctly linked together.

As another example, when two models m_1 and m_2 are proven equivalent (any execution possible in one is also possible in the other), then any proof made using the definition of m_1 can be transferred to m_2 *without any risk* of a shift in the definitions.

In this article

All the results we present in this paper have been fully formalized and proved in COQ and PACTOLE. For the sake of clarity a few definitions given in the following have been slightly stripped of some technical details. The actual formal development is publicly available at <https://pactole.liris.cnrs.fr>.

3.3 A tour of formal proof for robotic swarms

A formal semantics of a dynamic system (processor, virtual machine, robot swarm, physical system, etc) is a mathematical object that mimics perfectly the aspects of the behaviour of the system under consideration. All possible behaviours of the system must be possible in the model, and any impossible behaviour of the system must also be impossible in the model. Some things may be left out of scope of the model, usually on the basis of being irrelevant to the particular problem under study. For instance, we may ignore thermal radiation from the sun slowly heating up robots, as in most cases this does not result in any noticeable behavioural change. Or, while modelling a processor, we may choose not to model its performance counters and their associated instructions.

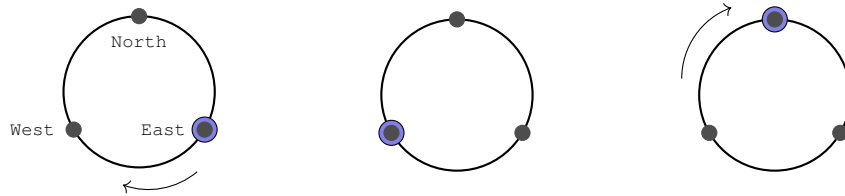
The mathematical object modelling the system is generally a function³ taking as input the state of the environment of the system and returning as output the *evolution* (the new state) of the system in this environment. In the following we give a series of examples of increasing complexity showing how we model different systems in robotic swarms.

3.3.1 A simple example

Suppose we want to model a single robot evolving on a ring with only three nodes in the following way: when reaching a node in the ring, the robot selects the next node clockwise and starts moving toward this new target.

² The actual work of reviewing in the context of formal proof is discussed by Bauer and Mahboubi [13, 56].

³ It may be a relation instead, for instance if the system is non-deterministic.



If we suppose that the robot cannot be interrupted during its move from one node to another, it is sufficient to model the topology with three positions North, East and West. A configuration is then given by a function returning the position of the robot.

Inductive Position : Type := North | East | West.

Definition Configuration := robot → Position.

It may seem silly to model a configuration with a single robot as a function but this representation generalizes to an arbitrary number of robots, so we choose to use it from the start.

The evolution of the system can then be formalized as a function round that takes the configuration and returns the new configuration after one round.

```
Definition round (c : Configuration) : Configuration :=
  fun id:robot =>
    match c id with      (* take position of id and compute the next one. *)
    | North => East
    | East  => West
    | West  => North
  end.
```

We can now define an execution as an infinite stream of configurations obtained by successive applications of round,⁴ and it is quite straightforward to prove for instance that in any such execution starting from a valid configuration (one position occupied by a robot) from any moment every node is occupied infinitely many times.

Lemma all_pos_occupied_eventually : \forall (c : Configuration) (p : Position),
Stream.eventually (**fun** str => (hd str) Robot1 = p) (execute c).

Proof.

```
intros c p.
(* by cases on positions and configurations *)
destruct p;destruct (c Robot1) eqn:heq;
  try (constructor 1;now auto);
  try (constructor 2; constructor 1;
    unfold execute, round; simpl;
    rewrite heq; reflexivity);
  try (constructor 2; constructor 2; constructor 1;
    unfold execute, round; simpl;
    rewrite heq; reflexivity).
```

Qed.

Lemma all_pos_occupied_forever : \forall (c : Configuration) (p : Position),
Stream.forever (Stream.eventually (**fun** str => (hd str) Robot1 = p))
(execute c).

⁴ See Figure 5 for the coinductive definition of execute in Coq.


```

Proof.
  cofix HI.
  constructor.
  - apply all_pos_occupied_eventually.
  - simpl.
    apply HI.
Qed.

```

The important statement here is that *what is true for this function is also true for the computation model it represents*.⁵ In other words we have reduced the problem of proving properties of the model to the problem of proving properties of a well defined function; a task theorem provers are perfectly suited for.⁶

3.3.2 The local computations

There is however a problem with this formalization. In the `round` function above, something important is left implicit: we formalized the decision of the robot *without defining the actual embedded algorithm operation*. Instead, we have only formalized a *centralized* protocol. This is a serious gap between the model we want to represent (autonomous robots) and our formalization. Distributed algorithms have very subtle behaviours, in particular because the code is executed on different devices “viewing” the global system from different perspectives. Any attempt to model distributed systems that jumps directly to a centralized vision like this would miss the important, and most difficult part, of distributed systems. In order to represent faithfully the distributed nature of our model we need to separate the computations done locally by each robot from the global behaviour of the system they yield.

To illustrate this, we need to define what the perception of the robot is. In this section we suppose that the robot sees the ring but cannot detect the “real” identity of a node. The robot sees the whole ring and knows on which node it stands, though it has no knowledge of whether it is actually North, East or West. The robot enjoys *chirality*: it can distinguish the node on its left (clockwise) from its right (counterclockwise). Let us rework our example to represent the distributed protocol. In the following we distinguish between two notions:

- the *global* configuration: what is really happening in the system, and where; this viewpoint is called the *global frame of reference* or the *demon’s frame of reference*;
- the (local) observation: the global configuration *as seen by the robot*. In our case the observation is composed of three nodes `Me`, `Left` and `Right` named after their positions relative to the observing robot. An observation is a function giving the position of all robots relative to the observing robot. This viewpoint is called the *local frame of reference* or the *robots’s frame of reference*.

```

Inductive RelativePosition : Type := Me | Left | Right.
Definition Observation : Type := robot → RelativePosition.

```

The protocol takes as input an observation and produces a *decision*: where to go next, expressed in its own frame of reference.

We reformulate the `round` function with an explicit call to the protocol on the observation by the robot. More precisely `round`

⁵ The fact that the represented model itself is the model accepted by the community needs a validation by experts.

⁶ Actually our claim is that humans also take benefit in agreeing that the function is accepted as *the actual true definition* of the model, once approved by the community.

- (1) establishes the robot’s observation,
- (2) passes it as a parameter to the protocol and takes back the returned decision (expressed in the robot’s own frame of reference), and then
- (3) determines where the robot moves in the global frame of reference.

Note that the protocol we consider in our example is very simple: since the robot sees itself at `Me`, its target is always the `Left` node in its own frame of reference.

Denoting by `relative_config` the “localized” version of the configuration where nodes are described relatively to an observing robot (eg. in the first figure of Section 3.3.1 where the robot is at `East`, `East` becomes `Me`, `West` becomes `Left`, `North` becomes `Right`) and `globalise_pos` the converse function mapping local names to global ones, we obtain the simple code:

```
(* Always go left, since this is clockwise. *)
Definition protocol (o : Observation) : RelativePosition := Left.
Definition round (c : Configuration) :=
  fun id : robot =>
    let pos_id := c id in (* Where is id? *)
    let obs := relative_config c pos_id in (* id sees c rotated with id on Me *)
    let destination := protocol obs in (* Call protocol on observation *)
    globalise_pos pos_r destination. (* Rotate back to global reference *)
```

It is now *provable* that this protocol behaves globally like the centralized version described in Section 3.3.1.

```
Lemma equiv_centralized :
  ∀ c : Configuration, execute c ≡ Centralized.execute c.
Proof.
  cofix HI.
  intros c ; constructor ; [ simpl ; reflexivity | apply HI ].
Qed.
```

This kind of proof may be quite difficult on realistic protocols. It generally (although not in this case) relies on the fact that the algorithm consists in operations that are invariant relative to the frame of reference.

Note that we can consider another distributed algorithm in the same model just by changing the protocol operations.

3.3.3 Weakening the sensing capabilities of robots

In this section, we change the computation model and the `round` function to account for more realistic sensors. In this new model, the robot’s compass may be subject to arbitrary recalibration at each round and change its *chirality*, i.e. its `Left` node may correspond to its clockwise or counterclockwise neighbour. Clearly the previous algorithm in this model behaves completely differently and does not satisfy the same properties, because when choosing always `Left` the robot may actually go counterclockwise.

The chirality reversal of the robot is not controllable and may change at each round, each leading to a different possible execution step. To account for this variability of execution we reformulate the model: `round` now takes a new parameter: a `flip` function that selects the chirality of the robot at the current round. The protocol is now called on the possibly flipped observation to simulate the new calibration of the sensors.

More generally, the uncontrollable part of the environment, that is the *alea* the protocol must be robust to, should be defined as a parameter of `round`, so that we can reason *for all* possible values.

02:10 Swarms of Mobile Robots: Towards Versatility with Safety

Denoting by `mirror_pos/_obs` the reversing of `Left` and `Right` we obtain the code:

```
(* flip id = false → robot id has clockwise orientation,
    true → counterclockwise orientation *)
Definition round (c : Configuration) (flip : robot → bool) :=
fun id : robot ⇒
  let pos_id := c id in
  let c_local := relative_config c pos_id in
  let obs := if flip id then mirror_obs c_local else c_local in (* flip? *)
  let dest := protocol obs in (* Call protocol on observation *)
  let dest_swap := if flip id then mirror_pos dest else dest in (*unflip?*)
  globalise_pos pos_id dest_swap. (* Rotate back to global reference *)
```

With this version of the formal model it is now possible to prove that, for example, there exist executions that never reach, say, node `East`. It suffices to use an infinite sequence of `flip` functions alternating the chirality of the robot, so that it would go alternatively to `North` and `West`.

```
Lemma exist_never_reaching_East: ∃ c (d : Stream.t (robot → bool)),
  Stream.forever (fun strm ⇒ hd strm Robot1 ≠ East) (execute d c) .
Proof.
  ∃ (fun id ⇒ match id with _ ⇒ North end). (* initial position *)
  ∃ (alternate (fun x ⇒ true) (fun x ⇒ false)). (* alternating demon *)
cofix HI. constructor.
- simpl. discriminate.
- simpl. constructor.
  + simpl. discriminate.
  + apply HI.
Qed.
```

3.3.4 Modeling Concurrency

Until now, our example involved one robot only. To model several distributed agents acting at the same time we need to determine the level of synchronicity of the agents. In the version above (Sections 3.3.2 and 3.3.3) we can see that `round` always applies the protocol at each round: the output position of *any* robot is obtained by calling the protocol on its observation. In other words, if multiple robots are present it *activates* all robots at each round. This model where all robots are always active at the same rate is called *fully synchronous* and is presented in more details together with others in Section 4.4.

We can relax this constraint to obtain a more loosely synchronized model: for some reason robots may not be all activated at each round. Similarly to chirality flips in the previous section, the subset of robots activated at each round is not controllable. Thus, in the same fashion that the `flip` parameter allows for quantifying on the uncontrollable variability of the sensors, a new parameter is added to account for the variability of scheduling. To avoid multiplying parameters we group uncontrollable parameters into a single record argument. This argument is called *demonic action* in the following in reference to the view of the environment as an adversary trying to make the protocol's task fail.

```
Record Demonic_action : Type := {
  activate : robot → bool; (* activated at his round? *)
  chirality : robot → bool; (* inverted chirality at this round? *)
}.
Definition round (da : Demonic_action)(c : Configuration) :=
```

```

fun id : robot  $\Rightarrow$ 
  if negb (da.(activate) id) then c id  (* if not activated, don't move *)
else                                     (* else move according to protocol and chirality *)
  let pos_id := c id in
  let c_local := relative_config c pos_id in
  let obs := if da.(chirality) id then mirror_obs c_local else c_local in
  let dest := protocol obs in
  let dest_swap := if da.(chirality) id then mirror_pos dest else dest in
  globalise_pos pos_id dest_swap.

```

In this version of the model, most provable properties depend upon a hypothesis on the *fairness* of the scheduler, i.e. constraints on the successive values of `activate` in the demonic action. See Section 4.4.

3.3.5 Other refinements

In the PACTOLE library, we model all the variants of the model described above and a few others.

The demonic action encompasses all external effects altering the operation of our robots. For example, the ground may be slippery and although robots try to reach a different node, some of them may simply not move. In this case, instead of reaching their target, the new locations of the robots depend on another Boolean function provided by the demonic action, say `reach`, expressing whether robots stay on their current location or reach their targets. Another example: a form of asynchronicity can be modelled by considering that robots may be activated while other have not yet reached their destination, etc.

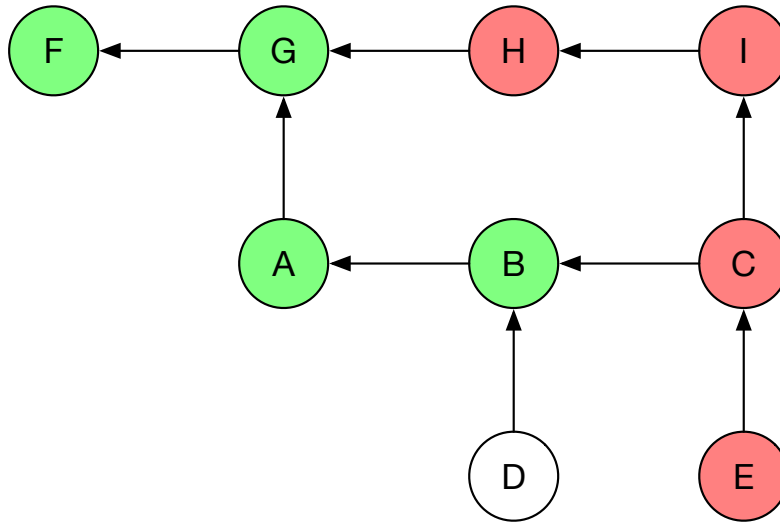
One does not have to keep extending the definitions of demonic action and `round` for each additional environmental effect: there is a general version that can encompass a wide range of external forces and is explained in Section 5.1. In order to get to this point, we must first present the seminal model by Suzuki and Yamashita [65] and the corresponding lattice of models.

4 A lattice of models

Research in Distributed Computing has traditionally considered three complementary approaches: **complexity-driven** when a particular problem can be solved in a particular model, it becomes interesting to reduce the complexity of the solutions. Various metrics can be considered, such as memory, time, number and size of exchanged messages, size of a causal chain of events, etc. **model-driven** when a particular model is designed for distributed computations (usually mimicking actual networks or systems), it becomes interesting to characterize the set of problems that are solvable in this model.

problem-driven when a particular problem is considered important in a general setting, it becomes interesting to characterize the models that enable solutions of the problem, and the models that make the problem impossible to solve.

The domain of mobile robotic swarms is mostly problem-driven. The focus of past efforts have thus consisted in characterizing which hypotheses are necessary and sufficient to solve a particular problem. Since the various hypotheses considered are sometimes unrelated, it becomes difficult to compare two different models with different sets of hypotheses. However, some particular hypotheses can be ordered, inducing a partial ordering among models. This partial ordering is important for two reasons: (i) if a model X is “weaker” than another model Y (in the sense that fewer computations are possible in model X than in model Y), then a solution to a problem considering model Y is also a solution to the problem considering model X , and (ii) if a given



■ **Figure 2** A lattice of models, with proofs on models B and H carrying to other models in the lattice.

problem admits no solution in model X , then it also admits no solution in model Y . Since the ordering between models is only partial, it is possible that two distinct models are both necessary and sufficient for solving a particular problem, albeit being unrelated with respect to the partial order.

Figure 2 depicts a possible partial order of models, where $X \leftarrow Y$ denotes the fact that X is a weaker model than Y . For a given problem, one was able to prove that a solution exists assuming model B , but that no solution exists assuming model H . Hence, from the partial hierarchy of models, it is possible to deduce that the problem is also solvable in models A , G , and F (that are weaker than B), and impossible to solve in models I , C , and E (that are stronger than H). From the current results, it remains unknown whether the problem is solvable assuming model D .

Hypotheses about the model span across various dimensions. The main ones are:

- synchronization** relates to the fact that mobile robots have independent control flow, and may thus execute their protocol at different paces;
- memory** relates to the fact that robots may make use of persistent memory, and may want to store various kinds of data (*e.g.* bits or Euclidean positions);
- sensors** relates to the fact that robots may have limited sensing capabilities, or limited ability to receive messages from other robots;
- actuators** relates to the fact that robots may have unreliable motion actuators;
- faults** relates to the fact that robot may follow their prescribed protocol or deviate from it [29].

The rest of the section presents the main relevant hypotheses that have been considered since the paper of Suzuki and Yamashita, and their induced order.

4.1 The Suzuki and Yamashita model

The seminal paper for studying robotic swarms from a Distributed Computing perspective is due to Suzuki and Yamashita [65]. They introduce, in this paper, a mathematical model for studying geometric pattern formation by swarms of possibly oblivious robots. The motivation for studying oblivious robots (that is, robots that do not retain history of past actions) is resilience to faults. For example, if a robot crashes, after rebooting it should not trust the content of its memory, either because it might be corrupted, or because it refers to an outdated view of the system. Ignoring past actions forces the design of algorithms that are simply more versatile.

In the Suzuki and Yamashita initial model, robots are represented as dimensionless points evolving in a bidimensional Euclidean space (that is, \mathbb{R}^2), and can accumulate on the same location. They operate in *LOOK-COMPUTE-MOVE cycles*. In each cycle, a robot “Looks” at its surroundings and obtains (in its own coordinate system) a snapshot containing some information about the locations of all robots. Based on this visual information, the robot “Computes” a destination location (still in its own coordinate system), and then “Moves” towards the computed location. When the robots are oblivious, the computed destination in each cycle only depends on the snapshot obtained in the current cycle (and not on the past history of actions). The visual snapshots obtained by the robots are not necessarily consistently oriented in any manner. Then, an *execution* of a distributed algorithm by a robotic swarm consists in having every robot repeatedly execute its LOOK-COMPUTE-MOVE cycle. In general, executions are *infinite* (even if robots do not move after a while, they still look, compute and decide not to move) and *fair* (every robot executes an infinite number of LOOK-COMPUTE-MOVE cycles).

Although this mathematical model is perfectly precise, it allows a great number of variants (developed over a period of 20 years by different research teams [37]), according to the various dimensions described above, namely: sensors, memory, actuators, synchronization, and faults, which we investigate now. This flurry of subtly different models makes reasoning very error-prone as it can easily happen that one designs a protocol in a model, and derives its proof in a slightly different one, without noticing the difference. A summary of all model variants explored in this section is depicted on Figure 4. In order to remain readable, we present all the dimensions separately, so that the overall lattice must be understood as the Cartesian product of all these smaller lattices.

4.2 Sensors

Robots perceive their surroundings through sensors, whose abilities have strong impact on task solvability. The most commonly considered types of sensors vary along several capabilities, described below. Obviously, one can think of other kinds of sensors not described here. For instance, in a completely opaque environment, one may imagine that the only available information is by direct contact through bumpers.

4.2.1 Range

The most obvious parameter of sensors is their range, that denotes how far a robot can sense another robot’s location:

full visibility robots are able to sense every other robot’s location, regardless of distance;

limited visibility there exists $\lambda > 0$ such that robots are able to sense every other robot’s location if their distance to the observing robot is less than λ , and are unable to sense the locations of other robots [38];

k -random there exists $\lambda > 0$ such that robots are able to sense every other robot’s location if their distance to the observing robot is less than λ , and up to k robots at distance more than λ , chosen uniformly at random, cannot be sensed (the other “distant” robots can be sensed) [45];

k -enemy there exists $\lambda > 0$ such that robots are able to sense every other robot’s location if their distance to the observing robot is less than λ , and up to k robots at distance more than λ , chosen by an adversary, cannot be sensed (the other “distant” robots can be sensed) [45].

Note that in general, robots are not aware of λ . Obviously, a protocol assuming limited visibility is strictly more powerful than one that requires full visibility.

4.2.2 Multiplicity detection

Multiplicity refers to the ability of robots to distinguish (to some extent) the number of robots sharing a given location. There are three variants about the accuracy, ordered by decreasing strength:

no multiplicity detection sensors can only distinguish occupied and unoccupied location, but any estimation about the number of robots present remains unknown;

weak multiplicity detection sensors can distinguish between a single robot or more than one robots at a location, but not their precise number [47];

strong multiplicity detection sensors can accurately count the number of robot at a location.

Another axis for variants is related to the range of the multiplicity detection:

local multiplicity detection indicates that weak or strong multiplicity information is only observable for the position of the observing robot (that is, a robot can only obtain multiplicity information about its own location) [46];

global multiplicity detection indicates that weak or strong multiplicity information can be obtained for all observed positions (that is, a robot can obtain multiplicity information about all locations in its viewing range).

Overall, we thus have five variants for multiplicity: no multiplicity, weak local multiplicity, weak global multiplicity, strong local multiplicity, and strong global multiplicity. Obviously, an algorithm assuming no multiplicity detection is more powerful than one requiring any of the other assumptions. However, some assumptions are uncomparable, *e.g.* weak global multiplicity and strong local multiplicity.

4.2.3 Orientation

This refers to the ability of robots to share some common notion of direction or orientation. Again, there are many variants:

common direction the robots have the same North-South and/or East-West axes, but the direction along these axes may be inverted (this is also called two-axes direction) [40];

common orientation in addition to having the same two axes direction, robots may also share orientation on either one axis (*e.g.* North only) or two axes (*e.g.* North and West);

common chirality robots have the same notion of left and right [39].

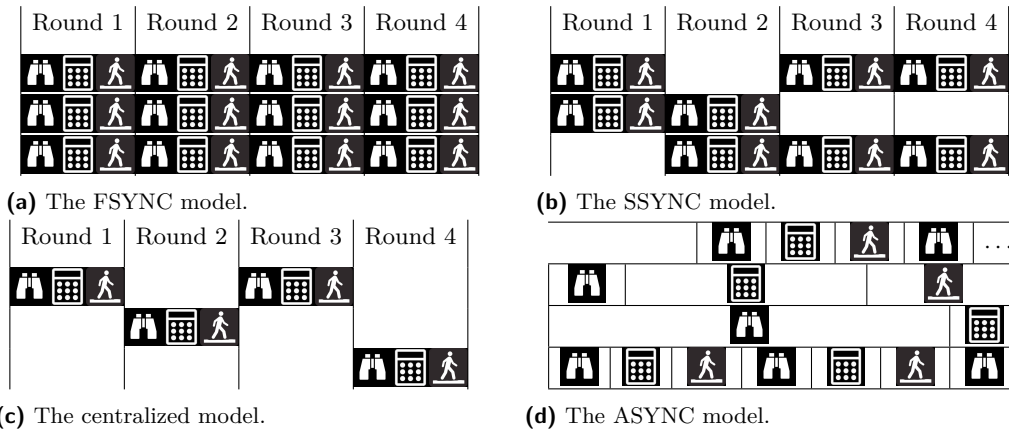
Notice that it is entirely possible to have common chirality without sharing a common direction. When having common direction, orientation on one axis, and chirality, robots are said to have *full compass*. Note that orientation on one axis and chirality amounts to having orientation on two axes.

4.3 Memory/Communication

The benefit of using oblivious robots is that they easily recover from crashes and memory corruption. Nevertheless, several extensions with memory have been proposed, ordered from strongest to weakest:

oblivious only volatile memory is available. Memory is reset at the beginning of each LOOK-COMPUTE-MOVE cycle. Robots thus have no memory of past actions. Practically, robots only use their current snapshot in the compute phase.

finite memory robots may have persistent memory between LOOK-COMPUTE-MOVE cycles. Several variants of this model called the *luminous* model [26, 66, 44, 27, 25] have been investigated (see below).



■ **Figure 3** Synchronization hypotheses in models.

infinite memory robots may make use of an infinite amount of memory. It allows robots to remember a full observation, as the position of robots may have to be encoded as actual real numbers (in the ego-centred observation).

Since robots are assumed to be anonymous, there is no way of performing point-to-point communication with a particular neighbour. Therefore, communication is handled through broadcast, which is described as the robots having *lights* whose color may be adjusted during the compute phase. In addition to the number of available colours for a robot light (hence the amount of states of information transmitted), there are three kinds of lights:

internal lights are only visible by the emitting robot itself, thus actually represent finite memory (the robot communicates with itself);

external lights are only visible by other robots but not the emitting root, thus they represent communication without memory [60];

full lights combine internal and external lights: they are visible by all robots.

4.4 Synchronicity and fairness

The considered model is based on discrete logical time, that is, on a sequence of events, an event being any change in the state of any robot.

The possible interleavings of those events define the synchronicity level of an execution.

If the LOOK-COMPUTE-MOVE cycles are considered atomic, that is no event can occur during a cycle, the model is said to be *semi-synchronous* (SSYNC): a subset of the robots enter (and finish) their cycle and each phase within it simultaneously while the others are idle, hence the notion of *round*. In the constrained version of SSYNC where no robot is idle, that is where *all* robots are activated simultaneously, the execution is said to be *fully-synchronous* (FSYNC). In the case where the cycles are not atomic and may overlap, the execution is said to be *asynchronous* (ASYNC) [40]. Clearly, ASYNC is the strongest model and FSYNC is the weakest.

A fourth synchronicity model exists: the *centralized* one, where only a single robot moves every round. It is a particular case of the SSYNC model (thus it is weaker) but it is incomparable to the FSYNC one.

Figure 3 illustrates these synchronicity hypotheses.

These synchronicity hypotheses between the LOOK-COMPUTE-MOVE cycles of robots are of paramount importance for proofs. Many proofs made in weak synchronization models were claimed to hold also under stronger synchronization models but turned out to be incorrect. This is actually the main source of errors in the literature.

02:16 Swarms of Mobile Robots: Towards Versatility with Safety

In the FSYNC, SSYNC, and centralized models, the actual duration of each phase does not matter since no observation occurs while a robot is moving, which justifies using discrete logical time. On the opposite, the ASYNC model represents the complete lack of synchronization between robots, and duration is important here, as a robot may observe others while they are moving.

Fairness

In all models except FSYNC where all robots are active at all times, the subset of active robots is chosen by the environment. In all generality, nothing prevents the environment, a.k.a the *demon*, from starving some or all robots. Obviously, most tasks are infeasible if some robots never get opportunities to act. Thus, there are fairness constraints on demons: a demon is said *fair* if every robot gets activated infinitely often. This is equivalent to saying that at any point of the execution, every robot is eventually activated.

Although fairness is usually enough for most protocols, it does not give any guarantee on the relative rates of robot activations: a robot may be activated arbitrarily more often than another. To remedy this situation, one can use the stronger fairness condition of *k-fairness*:⁷ every robot is activated at least once for every k activations of any other robot.

4.5 Rigid/Flexible Movement

The atomicity of cycles does *not* imply that the computed destination is actually reached by a robot before the start of its new cycle: the robot may be interrupted during its move by the environment.

An execution where all robots always reach the destination returned by the protocol is said to be *rigid*. Conversely, if robots can start a new cycle before they completed their scheduled journey, the execution is *flexible*. In such a case, so as to avoid Zeno-like counter-examples, it is assumed that robots travel at least some minimal non-null distance δ towards the *expected destination* before being subject to restart. In particular, a journey shorter than δ is always completed. This minimal uninterruptable distance is unknown to robots;⁸ they may however take into account that such a minimum exists.

4.6 Faults

In an adversarial environment, faults must be considered, either because malicious agents are present or because one of our own agent has been corrupted.

The most common fault hypotheses made in models are (from strongest to weakest):

byzantine faults some robots do not follow the protocol and are controlled by an adversary;

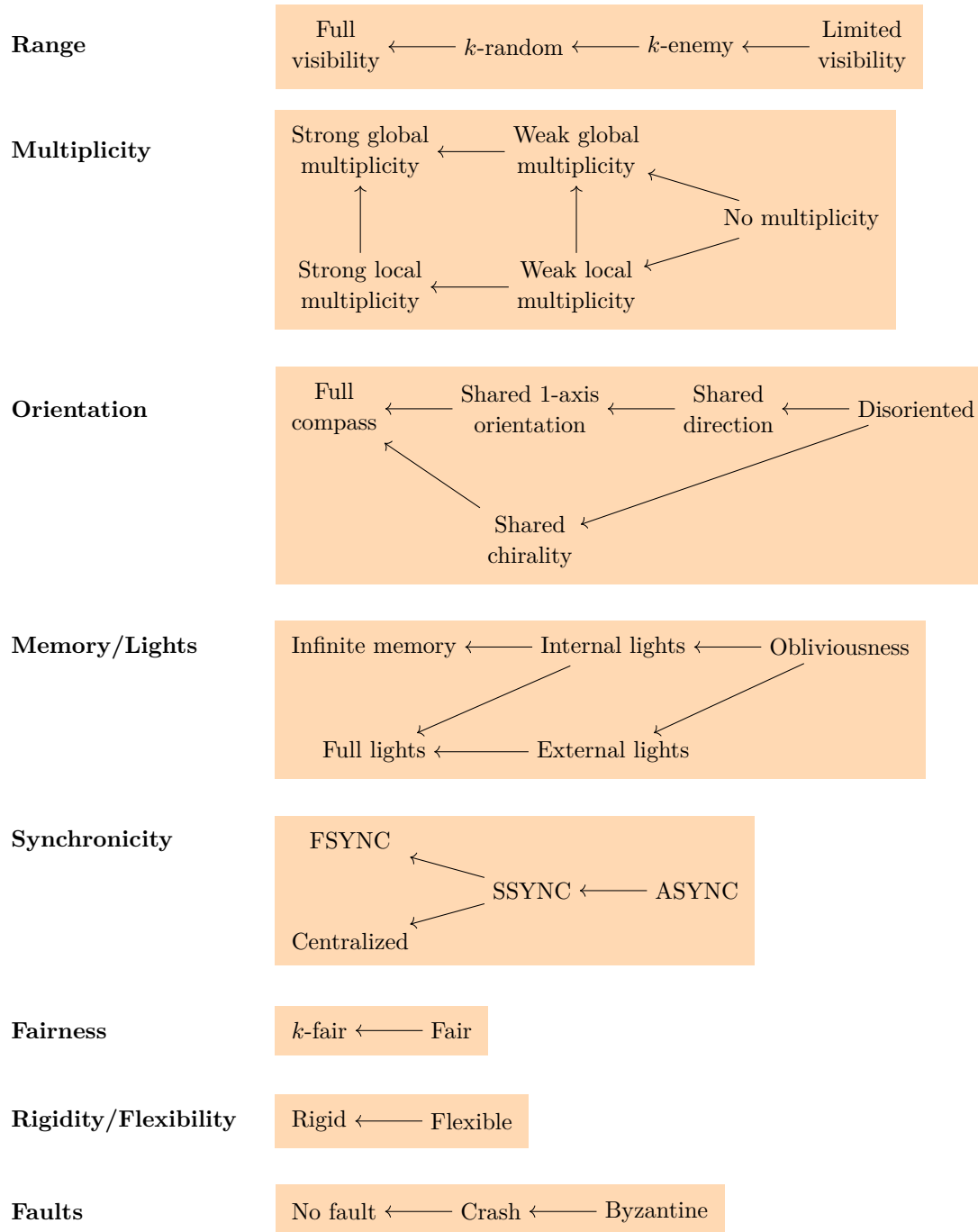
crash some robots may crash and stop acting forever;

no fault correct robots follow the protocol forever.

Notice that the faults described here are permanent, that is, they span the entire execution once they occur.

⁷ Early literature introduced *k-bounded*: between two activations of a robot, there are at most k activations of any other robot. This is not equivalent to *k-fairness* since it is vacuously satisfied if a robot is never activated: there are no two activations we should count activations of other robots between. Furthermore, one can prove that *k-bounded* and *fair* is equivalent to *k-fair*, making *k-fairness* the useful notion.

⁸ It would however make little sense to base an algorithm on such an absolute δ as it is possible that robots do not share frames of reference.



■ **Figure 4** The model lattice for robot swarms, as the Cartesian product of smaller lattices.

5 The formalization of the Suzuki, Yamashita model

The formalization of a computation model in a proof assistant consists of a body of mathematical definitions linked together. The main statement defines the set of *correct computations* in this model. This definition must be inspected very carefully to ensure it complies exactly with what specialists have in mind. It is however not rare that on the occasion of a formalization one realizes that different specialists have slightly different models in mind. Formalization is thus the occasion of clarifying things, either by proving equivalence of models or by establishing more subtle correspondences between them.

The computational model introduced by Suzuki and Yamashita states basically that *robots move in space according to their observation of the environment*. In order to complete a formal description of this model, we hence have to provide a COQ encoding of the relevant space, and of course a way to characterize robots and their sensors (that is the way the environment is perceived).

Implementing robot capabilities and instantiating the universe the robots move in must be generic, abstract, and (relatively) user-friendly. This is crucial, as this is where *the developer* clarifies assumptions and removes ambiguities and *the expert* validates definitions. It is a *sine qua non* for a broad acceptance of a formal framework.

We must also formalize the LOOK-COMPUTE-MOVE cycles and their possible interleavings: i.e. the core of the model and its synchronicity level. While the latter is still a parameter chosen by the user, the core itself is agnostic to assumptions about the environment. It is modelled in the framework by a function, `round`, that the developer never has to look at, except maybe for reassurance that it actually encodes Suzuki and Yamashita's robotic swarms model.

We hence shall describe first how `round` simulates the evolution of the system, with completely abstract parameters. Then we explore how various flavours of Suzuki and Yamashita's robots can or cannot solve fundamental tasks, and how to instantiate those variants within our formal framework.

5.1 Structure of the model, abstractions

So as to keep the core of the model as generic as possible, we provide the description of the environment as parameters. This way, those specifications and assumptions are kept abstract in the core, that is the actual description of how the system evolves.

The environment may be defined with several straightforward settings: the *space* where robots are moving, the level of *synchronicity*, the characteristics and capabilities of *robots and their sensors* (for example their accuracy or range), etc. Figure 6 and Section 5.4 show in details the structure of all parameters that must be instantiated.

Following Figure 6 we consider being given:

- a topology with its usual operations,
- the definition of a robot's *state* that includes its location (the position in space it is at), and a way of accessing it for all robots,
- a notion of *observation* that only considers what is allowed by the relevant variant, and finally
- an embedded algorithm : the protocol.

Usual operations regarding the space and its topology typically include a decidable equality on locations and change of frame operators.

A *state* describes, typically as a record, the internal state of robots, including in certain cases of synchronicity (namely ASYNC) their computed destination. Its access is ensured through the *configuration*: a function that takes a robot identifier as argument, and returns its state.

The configuration cannot generally be used as an observation of the robots as it may include private information about internal states, and thus may display *what should not be observed* by other robots. It is the case for example if local sensors have a limited range, or cannot get robots' ids, or even cannot detect multiplicity (i.e., the exact number of robots inhabiting a location in space), etc. Forbidden/private information is thus pruned from the configuration to get an *observation*. That observation is the one and only allowed input for a robot to compute its next destination; examples of its instantiation are given in section 5.3.2.

The protocol, that is the embedded algorithm that returns a path to a destination ⁹, based on an observation, is shared by all robots. It consists of the actual function mapping observations to (path-containing) states, and some properties (though irrelevant to `round`) ensuring, for example, that two equivalent observations produce equivalent paths and destinations, or on a graph that there is an actual arc towards the targeted node.

While the PACTOLE library can express all of the synchronicity hypotheses, namely ASYNC, SSYNC, centralized, and FSYNC, we shall focus in the following on the description of SSYNC executions (of which FSYNC executions are a particular case).

Modelling flexible executions simply amounts to allowing for a restart of any robot at any ratio of its trajectory, providing the effectively travelled distance is at least the minimal one δ . The new state is returned accordingly.

5.2 The function `round`

We describe the evolution of the system, following Suzuki and Yamashita's model, with a function: `round`.

5.2.1 Inputs

We want to design a function that, from a configuration, and given an embedded algorithm, returns the next configuration. Two obvious parameters for `round` are thus:

1. the configuration, and
2. the shared algorithm driving the robots: the protocol.

However, what happens in a step of execution depends also on some choice made by the demon, akin to Maxwell's: which robots are activated, what the new frames are, what distances are effectively travelled. . . We consider those choices as the results of a *demonic action*, which is given as an extra argument to `round`:

3. a demonic action.

We may find in a demonic action:

- the indication that a robot of a given id is activated, that is a function `activate` returning a Boolean when given an id as argument, in the special case of an FSYNC execution, its result is always `true`.
- a function for conversion of frames of references, say `change_frame`,¹⁰
- the actual function returning relevant choices on the entry of any robot's id. That function is a parameter of the model, it is hereafter referred to as `choose_update`.

⁹ Observe that simply assuming robots move toward the destination along a straight line precludes the use of our framework for e.g. proving the correctness of existing algorithms that make use of non-linear paths, such as parametric paths used by Defago et al. [30]. Hence, to preserve generality of the framework, we assume protocols return a path.

¹⁰ As it is constrained by the robot under consideration (recall that frames are self-centred) this function takes also a robot as argument.

02:20 Swarms of Mobile Robots: Towards Versatility with Safety

Depending on assumptions, robots may also undergo Byzantine failures. As the movements of Byzantine robots is, by definition, not determined by the algorithm, demonic actions must in that case include:

- a function that chooses the next destination of each Byzantine robot, hereafter referred to as `relocate_byz`.
- Finally, one needs a set of properties ensuring that the choices are coherent, for example that robots do not go past their computed destination, do not follow non-existing paths, etc.

The infinite sequence of demonic actions characterizes the choices for the whole execution, and constitutes the *demon* of the execution.

Finally, we have to be able to express flexible executions. Recall that in those, robots may be interrupted/restarted *before* they reach the end of their planned journey, but *after* they travelled at least an absolute distance δ . While the value of δ is unknown to robots, it is used in enforcing that the execution fulfils the aforementioned constraints. It has thus to be provided to `round`:

4. a minimal travel length δ .

It is worth noticing that the formal development allows for comparison of demons; it provides in particular proofs that demons with rigid movements are equivalent to demons with flexible movements and movement ratio 1, in the sense that any execution for one can be also obtained for the other.

We shall now describe the evolution of the system, following Suzuki and Yamashita's model, by devising our function `round`.

5.2.2 Operation

For the sake of simplicity, we focus on SSYNC flexible executions, of which FSYNC flexible executions are the particular case where *all* robots are selected to be activated by every demonic action.

In the remainder of this section, we shall denote the four formal parameters for `round` as follows :

- δ obviously represents δ ,
- `r` represents the protocol,
- `da` represents the used demonic action,
- `c` represents the configuration.

The result of `round` is a configuration, that is a *function*. The body of `round δ r da c` is hence a functional object taking an identifier, say `id`, as unique parameter, and returning its (new) state. We just have to describe this new state.

The first step is to figure out if the robot is activated, that is susceptible to undergo any change. This is a decision of the demon, and as so is stated in the demonic action. In an SSYNC context, `da(activate) id` can either return `false`,¹¹ in which case the new state is the previous one: `c id`; or return `true` in which case choices and changes may apply for robot `id`: usually some change of frame function together with a travel ratio.

Note that this information is irrelevant if `id` is Byzantine. Should it be the case, its new state would be arbitrarily chosen by the demon, using `da(relocate_byz)`.

Otherwise, if `id` refers to a correct robot, the new state depends on the protocol that requires an observation for `id`.

¹¹ Recall that this is never the case in an FSYNC execution.

1. The configuration is thus expressed from `id`'s point of view using its new frame of reference provided/chosen by the demonic action `da`. One obtains `id`'s *observation* by pruning the now translated configuration from illegitimate information.
2. **The protocol may now be applied on that observation.**
 The results contains in particular a path to a destination location, but is this *local* target reached before a new cycle starts? That depends of a choice of the demon, and is constrained by the ratio provided in `da.choose_update` `id`.
 Any destination location, in the local targeted state, closer than δ is reached, otherwise the location attained along the path is the one determined by the chosen ratio, the *chosen* target.
3. The demon-chosen target state is computed from the local target state by applying in particular the travel ratio.
4. As the distance between the current location of the robot and the chosen target is to be compared to the *absolute* δ , coordinates have to be translated back to the demon's frame of reference. The actual update with the new state can now be obtained from the result of this comparison: either the new location corresponds to the target locally computed or to the (demon-) chosen one.

5.3 Model specialization

As the core of the model is set once and for all, we may consider the many variants of Suzuki and Yamashita's robots. A slight modification of the robots, in their sensor capabilities for example, can dramatically change the feasibility of any given task. We shall review some of those, and describe how to instantiate the formal model accordingly.

5.3.1 Space

In this paper, we mainly consider the Euclidean plane \mathbb{R}^2 [24, 11]. Nevertheless, the formal model is not tied to this choice and we can consider any other space such as the real 3D space (\mathbb{R}^3), the real line (\mathbb{R}) [6, 23], discrete ones such as a ring ($\mathbb{Z}/n\mathbb{Z}$) or an arbitrary graph [12], possibly with robots moving continuously on edges [7], or even more exotic ones.

Providing a space in the formal framework amounts to define a type of *locations* where the robots are, and the necessary operations to compute: distances, the actual operations of the protocol, and *changes of frames of references*.

The computations by the protocol usually involve basic arithmetics; this is for example the case in Courtieu et al. [24] or Balabonski et al. [11] where all the necessary machinery to compute barycentres is provided with the instantiation of \mathbb{R}^2 .

Conversions into different frames of references can be as simple as rotations and homothetic transformations in a Euclidean space. They are however subtler when the considered space is a graph, and may then involve permutations of the node names.

To allow for a comfortable use of graphs in PACTOLE, Balabonski et al. [12] define a (light-weight¹²) template to be instantiated as desired by the user. The relevant interface is designed to link up with the general signature for spaces; it provides also a specialised version for rings.

¹²This template is not intended to be as powerful as a specialised development on graphs, like for instance the LoCo library for local computation on graphs [19].

```

Definition round  $\delta$  r da c :=
  (** for a given robot, we compute the new configuration *)
  fun id  $\Rightarrow$ 
    let state := c id in                (* state: id's state as seen by demon *)
    if da.(activate) id                  (** Is the robot activated? *)
    then match id with                  (** Byzantine or correct? *)
      | Byz b  $\Rightarrow$  da.(relocate_byz) b    (* Demon relocates Byzantine *)
      | Good g  $\Rightarrow$                        (* Config. expressed in the frame of g: PHASE 1 *)
        let frame_conv := da.(change_frame) c id in
        let local_config := map_config frame_conv c in
        let obs := obs_from_config local_config id in
                                                    (* APPLY r ON OBSERVATION: PHASE 2 *)
        let local_target_state := r obs in
                                                    (* Demon chooses a point on the path to the target *)
        let chosen_target_state := da.(choose_update)      (* PHASE 3 *)
                                                    id
                                                    local_config
                                                    local_target_state in
          (update
             $\delta$ 
            local_config
            id
            frame_conv -1
            loc_target_state
            chosen_target_state)
        end
    else state.                          (** Inactive robots stay unchanged *)

  (** [execute r d config] returns an (infinite stream) execution from an
  initial global configuration config, a demon d and a protocol r
  running on each good robot. Each configuration being the result of
  round applied to the previous configuration (and the corresponding
  demonic_action). *)
Definition execute r : demon  $\rightarrow$  configuration  $\rightarrow$  execution :=
  cofix execute d c :=
    Stream.cons c (execute (Stream.tl d) (round  $\delta$  r (Stream.hd d) c)).

```

■ **Figure 5** The round function, core of the formal simulator, and the execute function that produces an infinite execution from it.

5.3.2 Sensors

One of the key concepts in Suzuki and Yamashita’s model is the one of observation, that is the way to get some snapshot about the environment. The sensor capabilities can indeed change dramatically what is achievable in any given model. A very simple example of that is perception with a *limited* range: if robots are far enough apart to the point of not being aware of the others, there is no hope of cooperation of any kind!

Even seemingly minor differences may change the impossibility frontier: Gathering is impossible in general [23], but becomes possible with either a common compass [40] or detection of multiplicity [24] (that is, when robots can count the number of robots on a location instead of just detecting that the location is inhabited).¹³

Sensor capabilities are modelled through the way the configuration is perceived, that is transformed into an actual *observation*. The fact that states of robots may include some internal states, and thus may display information that should not be observed by other robots, prevents them to be used directly as an observation.

Depending on the variant one is interested in, assumptions may indeed require that local sensors cannot tell robots apart (anonymity), or detect whether they are correct or Byzantine, or are endorsed with multiplicity detection, etc.

These restrictions can be defined and encapsulated in the notion of *observation*, which characterizes what a robot’s sensors can perceive of the global system.

To obtain the observation, that is the only input to the protocol, all forbidden information must be pruned from the configuration. To that goal, a function `obs_from_config` is devised to return an observation when given the actual configuration. This function takes also the internal state of the observer, as the actual perception may depend on characteristics of *its own sensors*.¹⁴

Sensor capabilities are thus characterized through:

- the datatype for observation, and
- the operation of `obs_from_config`.

5.3.2.1 Anonymity, multiplicity

Modelling capabilities through the type definition is straightforward as the type has to describe only the public information.

Let us say one wants to model anonymous robots, equipped with sensors that can “see” the whole universe, and detect the number of robots inhabiting any location (strong multiplicity). A convenient datatype for the observation in that case may be simply a *multiset* of the inhabited locations, the detection of multiplicity (number of robots in a place) being directly expressed by multiplicities of elements (number of times a location appears as the location of a robot).

This is not tied to the underlying space: in the case of a discrete graph where no node naming or origin is shared between robots, one would have a multiset of nodes, with one node marked as the location of the observing robot.

In the case the anonymous robots cannot distinguish between different inhabitants, detecting only that the location is inhabited, the observation datatype may then be just a *set* of inhabited locations. Again, this would work just as well on a graph rather than the Euclidean plane.

For non-anonymous robots, observations may be sets of pairs consisting of a robot identifier and the location where this robot is.

¹³ Constraints on the starting configuration may be necessary, like the forbidding of a bivalent configuration.

¹⁴ Of course, that does implies presence of (parts of) the internal state in the result of `obs_from_config`, which depends on what is allowed by the variant.

5.3.2.2 Accuracy, range limits

It is possible to represent bounded accuracy of sensors, or limited vision, in the sense that the whole space is not perceivable by a single robot. Those are variants that are taken care of in the function rather than in the datatype definition directly.

Bounded accuracy can be obtained by rounding values in the configuration to the adequate level before entering them in the observation. It is for example possible to introduce noise, or even map actual locations to an underlying discrete version of the space, where robots perform their computations.

Limited perception is obtained by deleting from a “total” observation the robots that are out of range for the observer under consideration. Note the use of the internal state, given as a parameter, in that case where location and range of detection are at play to determine the observable ball.

5.3.2.3 Colours, memory

In the *robots with lights* variant, robots are equipped with coloured lights that they can turn on and off. Colours can have a dramatic impact on possibility results, as they introduce a form of communication and, when self-perceived, a certain kind of memory.

Let us consider that the observation for a variant without colours is, say, a multiset of n -tuples (location, id, etc.). Adding colours to that variant basically consists in adding the colour information to that observation, that is considering a multiset of $n + 1$ -tuples including the colour. An additional tuple that represents the observing robot itself must be added if it can detect its own colour. It must be absent from the observation otherwise, as it introduces some memorized information about the robot’s previous state.

Should robots be endowed with unbounded memory, being able to remember all previous observations for example, adding the list of previous observations as an internal state in the definition of the robots suffices. The relevant information from this list would then be included in the observation by `obs_from_config`.

5.4 Formal Parameters of the model

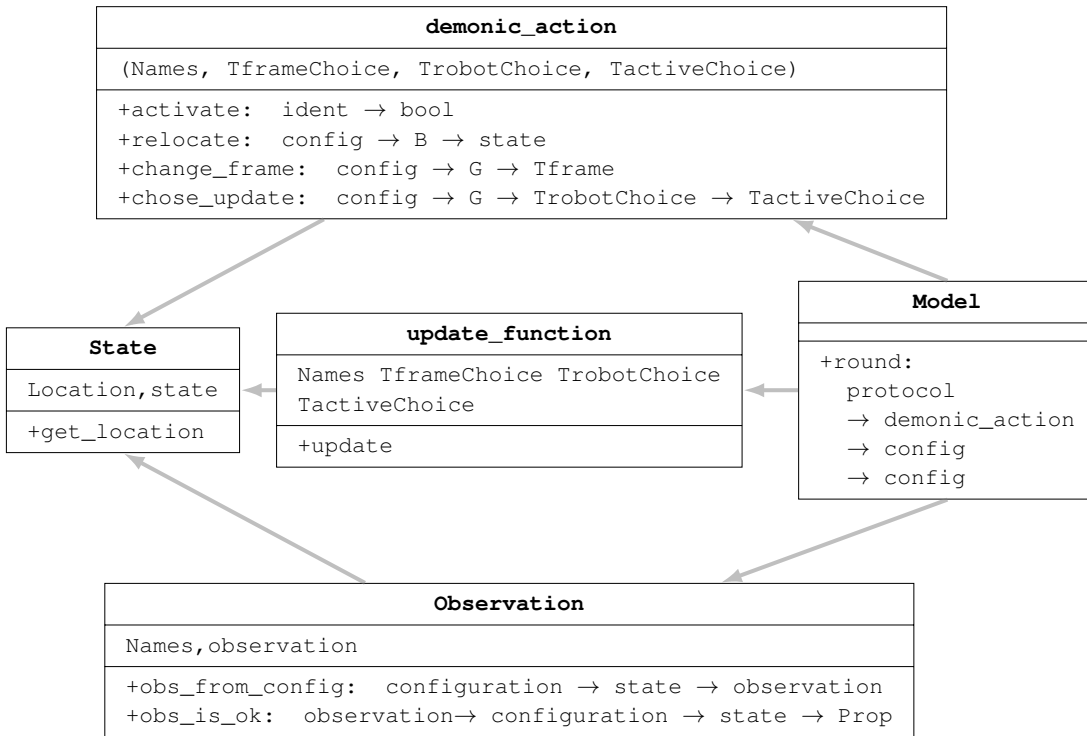
Figure 6 shows all the parameters needed to instantiate a particular model. The main parameters are:

- the location and more generally the **state** of the robot;
- the **observation** of the robot;
- the characterization of a **demonic action**;
- the way the robots move (**update function**), given the decisions of the robot and of the demon.

These parameters are themselves parameterized by types (Names, etc).

6 Examples

Examples provided in this section contain only a small subset of certified results obtained with the PACTOLE framework. For completeness, we summarize published results based on PACTOLE in Table 1.



■ **Figure 6** Parameters needed for a model.

6.1 Gathering

Robotic swarms are mostly a problem-driven domain, and as such focus on a few paradigmatic problems, some being fundamental, as for instance *Gathering*. This problem has been extensively studied, in particular by Principe [63], and in a formal setting by Balabonski et al. [10].

In its commonly shared definition, solving *Gathering* consists in having, within finite time, all correct (non-Byzantine) robots to stand on the same location, unknown beforehand, and to stay there indefinitely.

To describe *Gathering* formally, one has to define static (depending only on the configuration) and dynamic (depending on the demon) properties characterizing:

1. a configuration with all correct robots located at the same position, say p ,
2. an execution with all correct robots staying at the same position indefinitely,
3. an execution consisting of a finite number of evolution steps (the actual gathering movements), followed by what is an execution fulfilling the description of item 2.

The first item is easily modelled by a definition `gathered_at` stating that, given a configuration c and a location p , for any robot identifier, if that robot is correct, then its location is p in c .

Definition `gathered_at (p : location) (c : configuration) : Prop :=`

$$\forall \text{id, good id} \rightarrow \text{get_location (c id)} = p.$$

The second item characterizes an execution e with a location p : at each step in e , that is for each configuration c in e ,¹⁵ `gathered_at p c` holds. Let us call this property of p and e : *Gather* $p e$.

¹⁵ Recall that executions are streams of configurations, so that a property P on the head of a stream e must be projected by `Stream.instant P e`.

■ **Table 1** Certified results based on PACTOLE

Problem	Type of result	Setting	References	LoC
	Core	All		1 000
Framework	Spaces	\mathbb{R} , \mathbb{R}^2 , rings, grids, graphs		5 176
	Observation	multiset, sets		626
Gathering	Impossibility	\mathbb{R} , \mathbb{R}^2 , SSYNC	[23]	1 109
	Correctness	\mathbb{R} , \mathbb{R}^2 , not bivalent, SSYNC	[24]	2 307
	Correctness	\mathbb{R}^2 , FSYNC, flexible	[11]	609
Convergence	Impossibility	\mathbb{R} , 1/3 Byzantine	[6]	578
	Correctness	\mathbb{R} , FSYNC		170
Exploration with stop	Impossibility	Ring, FSYNC, $n k$	[12]	474
	Necessary condition	Ring, FSYNC		203
Life-line	Correctness	\mathbb{R}^2 , FSYNC	[8, 9]	1592
	Model equivalence	Graph, ASYNC	[7]	1187
	Model equivalence	ASYNC, flexible/rigid		275

```

Definition Gather (p : location) (e : execution) : Prop :=
  Stream.forever (Stream.instant (gathered_at p)) e.

```

The third item states a location p *exists* such that the Gather p status is reached in finite time for an execution e . `WillGather e` is directly an inductive property over streams, that is holding from a point in the stream accessible/reachable in finite time.

```

Definition WillGather (e : execution) : Prop :=
  Stream.eventually (fun ex => ∃ p, Gather p ex) e.

```

A protocol r achieves Gathering for a demon d if from *any* starting configuration c , all correct robots are eventually gathered forever in the execution obtained from r and d , that is :

```

Definition FullSolGathering r d : Prop :=
  ∀ c, WillGather (execute r d c).

```

Expressing now that a given protocol r is a solution to Gathering is simply stating that *for every* demon d , `FullSolGathering r d` holds.¹⁶

Conversely, expressing that Gathering is unsolvable (under certain assumptions) is simply stating that for any protocol r , it does *not* hold that r solves Gathering *for every* demon.

6.1.1 A model where gathering is proven impossible

In this section we give an example of a model where gathering is proved impossible. This is a well known fact [65, 63], of which a generalized version has been formally proved [23].

Instantiating the model

Let us have an arbitrary even number of robots, say n , of which none is Byzantine, moving on the Euclidean plane (note the use of **Variable** and **Hypothesis** for parameters left abstract).

¹⁶This can be constrained to demons *fulfilling the assumptions of the considered variant*: synchronicity, fairness, etc.).

```

Variable n : nat.
Hypothesis even_nG : Nat.Even n.
Definition MyRobots : Names := Robots n 0.
Definition Loc : Location := make_Location R2.

```

Movements are rigid (that is: robots always reach their destination before the next round). No demon interference is applied on robot's choice, and all operations deal with locations only:

```

Definition rchoice : Trobotchoice := location.
Definition state : State location := OnlyLocation.
Definition dchoice : Tactivechoice := unit.
Instance UpdFun : update_function := RigidUpdate.

```

Robots have multiplicity detection but cannot distinguish one robot from another. To model this limitation in sensing capabilities, we define the observation of a robot as a multiset of locations: the multiplicity of a location p gives the number of robots present at p , but robots are not identifiable. We give here the instantiation of the `Observation` model parameter. It is a record containing:

- (1) the logical definition of what the observation of a configuration must be: `obs_is_ok`, which is what is used in future proofs;
- (2) the definition of the function `obs_from_config` that computes the observation, from the configuration, and that is used in `round` (see figure 5)
- (3) the proof of correctness of `obs_from_config` with reference to the characteristic property `obs_is_ok`.

```

Definition multiset_observation : Observation := { |
  observation := multiset location;
  (* Characteristic property of the observation of a config. *)
  obs_is_ok obs (c:config) st :=
    ∀ loc, multiplicity obs loc
      = countA_occ loc (map c MyRobots);
  (* Function computing the observation from config *)
  obs_from_config c st := make_multiset (map c MyRobots) ;
  (* Proof that obs_from_config satisfies characteristic property *)
  obs_from_config_spec c st: obs_is_ok (obs_from_config c st) c st :=
    ...
| }.

```

Note that in this example, the assumptions on the sensors are completely global, unrelated to the internal state of the observer. Thus, `obs_from_config` does not use this state, even though it receives it as a parameter.

Stating the result

We call a position *invalid* if all robots are on two towers of the same height, that is: evenly distributed on exactly two distinct locations. Such a position is known as *bivalent*.

```

Definition invalid (config : configuration) :=
  ∃ pt1 pt2 : location, pt1 ≠ pt2
  ∧ multiplicity pt1 (obs_from_config config origin) = nG / 2
  ∧ multiplicity pt2 (obs_from_config config origin) = nG / 2.

```

The final lemma states that for any protocol, if one starts in an `invalid` configuration then there exists a demon that makes the protocol fail, i.e. that prevents the system to reach a gathered position.

```
Theorem noGathering : ∀ r : protocol, ∀ c : configuration,
  invalid c →
    ∃ d, SSYNC d ∧ Fair d ∧ ¬ WillGather (execute r d c).
```

A remark on the dual property

Adding as a condition on the initial configuration that it is *not* invalid (not bivalent) suffices to get a universal algorithm solving Gathering. Developed in PACTOLE, the solution given by Courtieu et al. [24] uses the *exact same specifications* of the model and the environment, thus eliminating any risk of shift, and closing the problem under those assumptions: Fair-SSYNC Gathering of oblivious rigid anonymous robots in \mathbb{R}^2 is impossible, unless the initial configuration is *not* bivalent, in which case a protocol is proven correct.

6.1.2 A model where gathering is proven possible

It is however possible to achieve gathering when different capabilities for sensors are assumed. For the sake of the example, we assume now, as in the works of Balabonski et al. [11], that sensors cannot detect multiplicity. It should be stressed here that the formal definition of the problem is *strictly* the same as before, preventing any shift or bias in its definition.

Instantiating the model

Let us have an arbitrary number n (more than 1) of robots (0 Byzantine) moving on the Euclidean plane.

```
Variable n : nat.
Hypothesis H_nG : n >= 2.
Definition MyRobots : Names := Robots n 0.
Definition Loc : Location := make_Location R2.
```

Movements are flexible (i.e. robots may not reach their destination before the next round). Let us have an arbitrary δ representing the minimal distance (in the global reference) a robot moves between two rounds, unless its destination is attained. The only choice made by the demon after a robot's decision is the *ratio* of the path toward its target the robot actually reaches.

```
Variable δ : R.
Definition ratio : Tactivechoice := {x : R | 0 ≤ x ≤ 1}.
Definition rchoice : Trobotchoice := path location.
Definition FlexChoice : update_choice := Flexible.OnlyFlexible.
Instance UpdFun : update_function rchoice location ratio := FlexibleUpdate δ.
```

As robots cannot detect multiplicity, observations may be reduced to a *set* of (inhabited) positions, as remarked in section 5.3.2.1.

```
Definition set_observation : Observation := { |
  observation := set location;
  obs_is_ok s c pt := ∀ l, In l s ↔ In l (map c MyRobots);
  obs_from_config c pt := make_set (map c MyRobots);
  obs_from_config_spec c st : obs_is_ok (obs_from_config c st) c st :=
    ...
  | }.
```

Stating the result

The main theorem of Balabonski et al. [11] states that the protocol `ffgatherR2`, given below, solves the gathering for any fully synchronous demon and any starting configuration.

```
Definition ffgatherR2_pgm (s : observation) : path :=
  paths_in_R2 (isobarycenter (elements s)).
```

```
Theorem FSGathering_in_R2 :
  ∀ d, δ > 0 → FSYNC d → FullSolGathering ffgatherR2 d.
```

And with strong detection of multiplicity?

The framework is generic enough to provide also a formal certification for a result by Cohen and Peleg [20, 21] when robot sensors are this time endowed with detection of multiplicity. The only noticeable difference in the two approaches is the definition of the observation: a multiset for Cohen and Peleg's, and a set for Balabonski et al.'s. The proof argument is similar, and only a few technical steps to take into account the new type for observation are required [11].

6.2 Exploration

An interesting benchmarking problem when dealing with robots on graphs is the one of *exploration*, in particular *with stop*. Exploration with stop (also known as Terminating Exploration) requires to ensure that:

1. all nodes are visited by a robot at some point during the execution, and
2. all robots eventually stop moving once all nodes have been visited.

There are thus 2 properties to formalize: for the space to be explored, and for the system to stop evolving.

For a node, say v , being *eventually* visited (inhabited) by a robot is simply an inductive (that is finitely reachable) property on the execution, say e : at some accessible point along e , the configuration returns a state displaying v as the current location.¹⁷ This is a basic property about streams.

```
Definition Will_be_visited v e :=
  Stream.eventually (Stream.instant (is_visited v)) e.
```

The second property, the halting, is built in three steps:

- firstly, one defines what is it for an execution to have two consecutive identical configurations, namely that it *stalls*. It is simply a call to the equivalence relation on configurations

```
Definition Stall e :=
  Config.eq (Stream.hd e) (Stream.hd (Stream.tl e)).
```

- secondly, the stall has to hold indefinitely

```
Definition Stopped e := Stream.forever Stall e.
```

- and finally the point where the execution is `Stopped` is reached within a finite number of steps; this is property `Will_stop`.

¹⁷Recall that, an execution being a stream of configurations, a property P on the head of a stream e is projected by `Stream.instant P e`.

```
Definition Will_stop e := Stream.eventually Stopped e.
```

Let $\text{execute } r d c$ be the execution obtained by running a protocol r with a demon d from an initial configuration c . Protocol r achieves Exploration with stop for a demon d if from *every* initial configuration c , $\text{Will_be_visited } v (\text{execute } r d c)$ holds for *every* node v (the exploration part), AND this execution stops, that is $\text{Will_stop } (\text{execute } r d c)$ holds.

```
Definition FullSolExplorationStop r d :=
  ∀ c, (∀ v, Will_be_visited v (execute r d c))
  ∧ Will_stop (execute r d c).
```

Similarly to what has been done for Gathering, expressing that a given protocol r is a solution to Exploration with stop is simply stating that for *every* demon d , $\text{FullSolExplorationStop } r d$ holds.¹⁸

Conversely, expressing that the problem is unsolvable (under certain assumptions) is simply stating that for any protocol r , it does *not* hold that r is a solution.

6.2.1 A model where Exploration with stop is proven impossible

We want to prove that Exploration with stop on a ring is not possible if the number of robots divides the size of the ring. We proceed along the lines of Balabonski et al. [12].

Instantiating the model

A ring is a special case of finite graph, already defined by the function `Ring` taking as input the size, which must be an integer greater than 1.

```
Variable ring_size : nat.
Hypothesis ring_size_spec : 1 < ring_size.
Instance Ring_space : FiniteGraph := Ring ring_size ring_size_spec.
```

Let us have an arbitrary number kG of robots (of which none is Byzantine) moving on our ring. We assume that kG divides the size `ring_size` of the ring and remove two corner cases: $kG = 1$ and $kG = \text{ring_size}$.

```
Variable kG : nat.
Instance Robots : Names := Robots kG 0.
Hypothesis kdn : ring_size mod kG = 0.
Hypothesis k_bounds : 1 < kG < ring_size.
```

As in the first example, robots contain no more information than their locations, robots' observations are the multiset of locations, and the demon does not interfere with the robot's choice, that is, movements are rigid.

```
Definition state : State location := OnlyLocation.
Instance RobotObs : Observation := multiset_observation.
Definition dchoice : Tactivechoice := unit.
```

¹⁸That is where constraints on the demon should appear, of the form *for all demons verifying such given property*, $\text{FullSolExplorationStop } r d$ holds, as detailed in Section 6.2.1.

The local frame is the one described in the introductory examples of Section 3.3, which amounts to a translation along the ring (relative locations) and potentially a symmetry (for chirality change).

The difference with these earlier examples is that robots do not choose a new node of the ring to move to, they only pick a direction. The update function is still rigid, and it simply applies the function `move_along` that returns the node reached by following the chosen direction from the robot's current location.

```
Inductive direction := Forward | Backward | SelfLoop.
Definition rchoice : Trobotchoice := direction.
Instance UpdFun : update_function := move_along.
```

Stating the result

The final theorem states that for any protocol r , there is a FSYNC demon d against which r does not solve exploration with stop, that is, there exists a configuration c such that the execution following r and d starting from c either does not terminate or does not explore the ring.

```
Theorem no_exploration :
   $\forall r : \text{protocol}, \exists d : \text{demon}, \text{FSYNC } d \wedge \neg \text{FullSolExplorationStop } r \ d.$ 
```

7 Related work

Numerous formalizations and verification tools have been designed and used to account for correctness in distributed computing. The formal treatment of mobile robotic swarms nevertheless requires specific tooling (w.r.t. “classical” distributed computing), as there is no direct transformation of “classical” shared memory or message passing models into models suitable to study mobile robotic entities. A naive transformation from the shared memory model would be to consider that the “values” shared by the robots are their positions, that observing other robots’s positions is similar to reading a shared variable, and that moving to a new position is similar to writing a new value in a shared variable. However, some aspects prevent formal solutions for the shared memory model to also apply to the mobile robots model.

First, the notion of “local observation” is quite specific to mobile robots: a robot has only access to a degraded view of its neighbourhood (according to its visual sensors), and the view is obtained in its local (i.e. ego centred) coordinate system. So, the same robot may appear at two different positions at the exact same time in the execution (by two different robots that have different coordinate systems), and, in the same execution, the same robot may appear or not appear at a given observing robot (depending on the location and sensing abilities of the observing robot). Second, in the more interesting ASYNC model, a robot can observe other robots while they move, resulting in getting any intermediate position the observed robot may reach during its movement. This is a strictly weaker setting than the classical “atomic” and “regular” shared memory registers popularized by Lamport [52, 53] (where reads may only return the “before-write” or the “after-write” value), and yet strictly stronger than the classical “safe” register [52, 53] (where *any* value in the domain could be returned by a read) if we make the reasonable assumption that robots cannot cover an infinite surface in a single move. Last: the position of a robot may belong to a continuous (dense) set, while a shared register is typically a discrete value.

This last concern makes model checking mobile robot algorithms quite difficult. If one wants a direct translation of the Suzuki and Yamashita model, one has to consider discrete (i.e. graphs) locations [14, 33, 35]. While this approach is suitable for checking problem instances, it cannot scale as when the number of locations becomes a parameter, interesting properties become

undecidable [64], even if the graph representing locations is as simple as a ring. Hence, the current hope on the model-checking side is to consider more abstract models, in the spirit of the recent approach by Defago et al. [28]. However, more abstract models yield two significant issues. First, models are likely to target a single problem rather than being generic, with no hint earned as how to handle other problems in the same setting. Second, one has to prove that properties obtained mechanically by model-checking for the abstract model echo in interesting properties for the original model (in the approach of Defago et al. [28], this part is handwritten, hindering a fully mechanized checkability of the approach).

From the methodology perspective, Pactole is close to Ivy [61, 58] and LoE [57]: they are based on a master model designed in a proof assistant and a methodology is designed to prove specific protocols in these model.

8 Conclusion

The mathematical description of mobile robotic swarms by Suzuki and Yamashita proved to be a fertile ground¹⁹ of research, with many model variants developed throughout the years, and many applicative domains envisioned by research teams all over the world (see the book edited by Flocchini et al. [37] for a recent survey).

When robotic swarm protocols are developed for critical applications, with lives at stake, reasoning about the model requires the foundations of a formal framework and methodology aimed at mobile robot protocol designers, to enable the certification of tentative robot protocols for any property related to their spatio-temporal behaviour that is useful in practice, or to demonstrate the impossibility of such designs.

A first step in that direction has been proposed with the PACTOLE framework, which allows so far working formally with:

- *Euclidean spaces* with geometrical constructs (barycentres, smallest enclosing circles, etc., together with their relevant properties) ; *Graphs*, either discrete or with continuous movement along the edges,
 - Rigid/flexible moves, SSYNC/FSYNC/ASync, various properties on demons (flavours of fairness), equivalences between demons, and means for the theoretical study of their lattice,
 - Common notions of observation depending on the capabilities of robots,
- including many cases studies, e.g. exploration, gathering, convergence...

PACTOLE is publicly available to the community at <https://pactole.liris.cnrs.fr>.

Three main axes of development are worth considering.

The first one addresses the issue of probabilistic behaviours. Probabilistic arguments are indeed commonly used in recent results regarding mobile robotic swarms, *e.g.* to break symmetries in configurations [18, 17, 68]. Probabilistic behaviours may also occur outside protocols, in the environment. Including probabilistic behaviours on the autonomous robot side (that is, robots are able to take actions based on some probabilistic source) and on the environment side (that is, the scheduling decisions are based on some probability distribution) in the PACTOLE framework is thus an important task.

The second one is to ensure that certified developments remain accessible to a non-specialist of formal proof, and to ease the proof burden as much as possible. This amounts to building manageable libraries with clear independent modules, definitions that can be shared and proof steps that can be reused. Fundamental results must be gathered into a formal library, with each relevant notion properly specified in the formal model, and each reusable property receiving a corresponding formal proof.

¹⁹ Masafumi Yamashita was awarded the 2016 Prize for Innovation in Distributed Computing for this seminal work.

Automation is expected wherever possible in this building process. Certifying properties of an algorithm in a proof assistant amounts to developing mechanical techniques matching the proof structures brought to the fore. Developing automation in that context is two-fold. On the one hand it involves high-level tactics to relieve the user from tedious and repetitive proof steps. On the other hand it must help in exhibiting certain properties and obtaining as a result a formal proof of it. In doing this, inputs from model checking approaches are precious due to their capacity to exhibit counter-examples, thus helping the developer, and to discharge automatically some base cases prior to induction steps.

Finally, it is fundamental to provide the prototype of an environment for proof and development of trustworthy distributed protocols for mobile robots, by linking together the results obtained with reference to the formal model, libraries, proof automation and management techniques, etc. Such an environment should allow the user to specify the algorithm, and to certify its properties, using generated verification conditions and proof constructs with the help of decision procedures. To reach this goal, one key challenge is to devise an annotation language rich enough to specify properties in the scope of our studies, but still convenient to use by designers of robot protocols. To ensure protocol validation, it must also integrate proof mechanisms allowing both assisted or automated certification, thus defining a complete certification chain that should be easy enough to use, even for a non specialist.

References

- 1 Jordan Adamek, Mikhail Nesterenko, and Sébastien Tixeuil. Evaluating and optimizing stabilizing dining philosophers. In *11th European Dependable Computing Conference, EDCC 2015, Paris, France, September 7-11, 2015*, pages 233–244. IEEE, 2015.
- 2 José Bacelar Almeida, Manuel Barbosa, Endre Bangerter, Gilles Barthe, Stephan Krenn, and Santiago Zanella Béguelin. Full Proof Cryptography: Verifiable Compilation of Efficient Zero-Knowledge Protocols. In *ACM Conference on Computer and Communications Security*, pages 488–500, 2012.
- 3 Karine Altisen, Pierre Corbineau, and Stéphane Devismes. A framework for certified self-stabilization. In Elvira Albert and Ivan 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.
- 4 Krzysztof R. Apt and Dexter C. Kozen. Limits for automatic verification of finite-state concurrent systems. *Information Processing Letters*, 22(6):307–309, 1986.
- 5 The Coq Proof Assistant. <https://coq.inria.fr/>.
- 6 Cédric Auger, Zohir Bouzid, Pierre Courtieu, Sébastien Tixeuil, and Xavier Urbain. Certified Impossibility Results for Byzantine-Tolerant Mobile Robots. In Teruo Higashino, Yoshiaki Katayama, Toshimitsu Masuzawa, Maria Potop-Butucaru, and Masafumi 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, November 2013. Springer-Verlag.
- 7 Thibaut Balabonski, Pierre Courtieu, Robin Pelle, Lionel Rieg, Sébastien Tixeuil, and Xavier Urbain. Continuous vs. discrete asynchronous moves: A certified approach for mobile robots. In Mohamed Faouzi Atig and Alexander A. Schwarzmann, editors, *Networked Systems - 7th International Conference, (NETYS 2019), Revised Selected Papers*, volume 11704 of *Lecture Notes in Computer Science*, pages 93–109, Marrakech, Morocco, June 2019. Springer-Verlag.
- 8 Thibaut Balabonski, Pierre Courtieu, Robin Pelle, Lionel Rieg, Sébastien Tixeuil, and Xavier Urbain. Brief Announcement: Computer Aided Formal Design of Swarm Robotics Algorithms. In Colette Johnen and Stefan Schmid, editors, *Stabilization, Safety, and Security of Distributed Systems - 23th International Symposium, (SSS 2021)*, Virtual conference, November 2021.
- 9 Thibaut Balabonski, Pierre Courtieu, Robin Pelle, Lionel Rieg, Sébastien Tixeuil, and Xavier Urbain. Computer aided formal design of swarm robotics algorithms. *CoRR*, abs/2101.06966, 2021.
- 10 Thibaut Balabonski, Pierre Courtieu, Lionel Rieg, Sébastien Tixeuil, and Xavier Urbain. Certified gathering of oblivious mobile robots: Survey of recent results and open problems. In Laure Petrucci, Cristina Seceleanu, and Ana Cavalcanti, editors, *Critical Systems: Formal Methods and Automated Verification - Joint 22nd International Workshop on Formal Methods for Industrial Critical Systems - and - 17th International Workshop on Automated Verification of Critical Systems, (FMICS-AVoCS 2017)*, volume 10471 of *Lecture Notes in Computer Science*, pages 165–181, Turin, Italy, September 2017. Springer-Verlag.
- 11 Thibaut Balabonski, Amélie Delga, Lionel Rieg, Sébastien Tixeuil, and Xavier Urbain. Synchronous gathering without multiplicity detection: A

- certified algorithm. *Theory of Computing Systems*, pages 200–218, 2019. <https://doi.org/10.1007/s00224-017-9828-z>.
- 12 Thibaut Balabonski, Robin Pelle, Lionel Rieg, and Sébastien Tixeuil. A foundational framework for certified impossibility results with mobile robots on graphs. In Paolo Bellavista and Vijay 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.
 - 13 Andrej Bauer. How to review formalized mathematics. <http://math.andrej.com/2013/08/19/how-to-review-formalized-mathematics/>, August 2013.
 - 14 Béatrice Bérard, Pascal Lafourcade, Laure Millet, Maria Potop-Butucaru, Yann Thierry-Mieg, and Sébastien Tixeuil. Formal verification of mobile robot protocols. *Distributed Computing*, 29(6):459–487, 2016.
 - 15 Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer-Verlag, 2004.
 - 16 François Bonnet, Xavier Défago, Franck Petit, Maria Potop-Butucaru, and Sébastien 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.
 - 17 Quentin Bramas and Sébastien Tixeuil. Brief Announcement: Probabilistic Asynchronous Arbitrary Pattern Formation. In George Giakkoupis, editor, *35th ACM Symposium on Principles of Distributed Computing (PODC 2016)*, pages 443–445, Chicago, IL, USA, July 2016. ACM.
 - 18 Quentin Bramas and Sébastien Tixeuil. The Random Bit Complexity of Mobile Robots Scattering. *International Journal of Foundations of Computer Science*, 28(2):111–134, 2017.
 - 19 Pierre Castéran, Vincent Filou, and Mohamed Mosbah. Certifying Distributed Algorithms by Embedding Local Computation Systems in the Coq Proof Assistant. In Adel Bouhoula and Tetsuo Ida, editors, *Symbolic Computation in Software Science (SCSS'09)*, 2009.
 - 20 Reuven Cohen and David Peleg. Robot Convergence via Center-of-Gravity Algorithms. In Rastislav Kralovic and Ondrej Sýkora, editors, *Structural Information and Communication Complexity - 11th International Colloquium (SIROCCO 2004)*, volume 3104 of *Lecture Notes in Computer Science*, pages 79–88, Smolenice Castle, Slovakia, June 2004. Springer-Verlag.
 - 21 Reuven Cohen and David Peleg. Convergence Properties of the Gravitational Algorithm in Asynchronous Robot Systems. *SIAM Journal of Computing*, 34(6):1516–1528, 2005.
 - 22 Thierry Coquand and Christine Paulin-Mohring. Inductively Defined Types. In Per Martin-Löf and Grigori Mints, editors, *International Conference on Computer Logic (Colog'88)*, volume 417 of *Lecture Notes in Computer Science*, pages 50–66. Springer-Verlag, 1990.
 - 23 Pierre Courtieu, Lionel Rieg, Sébastien Tixeuil, and Xavier Urbain. Impossibility of Gathering, a Certification. *Information Processing Letters*, 115:447–452, 2015.
 - 24 Pierre Courtieu, Lionel Rieg, Sébastien Tixeuil, and Xavier Urbain. Certified universal gathering algorithm in \mathbb{R}^2 for oblivious mobile robots. In Cyril Gavoille and David Ilcinkas, editors, *Distributed Computing - 30th International Symposium, (DISC 2016)*, volume 9888 of *Lecture Notes in Computer Science*, pages 187–200, Paris, France, September 2016. Springer-Verlag.
 - 25 Shantanu Das, Paola Flocchini, Giuseppe Prencipe, and Nicola Santoro. Forming sequences of patterns with luminous robots. *IEEE Access*, 8:90577–90597, 2020.
 - 26 Shantanu Das, Paola Flocchini, Giuseppe Prencipe, Nicola Santoro, and Masafumi Yamashita. Autonomous mobile robots with lights. *Theor. Comput. Sci.*, 609:171–184, 2016.
 - 27 Xavier Défago, Adam Heriban, Sébastien Tixeuil, and Koichi Wada. Brief announcement: Model checking rendezvous algorithms for robots with lights in euclidean space. In Jukka Suomela, editor, *33rd International Symposium on Distributed Computing, DISC 2019, October 14-18, 2019, Budapest, Hungary*, volume 146 of *LIPICs*, pages 41:1–41:3. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.
 - 28 Xavier Défago, Adam Heriban, Sébastien Tixeuil, and Koichi 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.
 - 29 Xavier Défago, Maria Potop-Butucaru, and Sébastien Tixeuil. Fault-tolerant mobile robots. In Paola Flocchini, Giuseppe Prencipe, and Nicola Santoro, editors, *Distributed Computing by Mobile Entities, Current Research in Moving and Computing*, volume 11340 of *Lecture Notes in Computer Science*, pages 234–251. Springer, 2019.
 - 30 Xavier Défago and Samia Souissi. Non-uniform circle formation algorithm for oblivious mobile robots with convergence toward uniformity. *Theor. Comput. Sci.*, 396(1-3):97–112, 2008.
 - 31 Carole Delporte-Gallet, Hugues Fauconnier, Yan Jurski, François Laroussinie, and Arnaud Sangnier. Towards synthesis of distributed algorithms with SMT solvers. In Mohamed Faouzi Atig and Alexander 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 200–216. Springer, 2019.
 - 32 Ha Thi Thu Doan, François Bonnet, and Kazuhiro Ogata. Model checking of a mobile robots perpetual exploration algorithm. In Shaoying Liu, Zhenhua Duan, Cong Tian, and Fumiko 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.

- 33 Ha Thi Thu Doan, François Bonnet, and Kazuhiro Ogata. Model checking of robot gathering. In James Aspnes and Pascal Felber, editors, *Principles of Distributed Systems - 21th International Conference (OPODIS 2017)*, Leibniz International Proceedings in Informatics (LIPIcs), Lisbon, Portugal, December 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.
- 34 Ha Thi Thu Doan, Kazuhiro Ogata, and François Bonnet. Specifying a distributed snapshot algorithm as a meta-program and model checking it at meta-level. In Kisung Lee and Ling Liu, editors, *37th IEEE International Conference on Distributed Computing Systems, ICDCS 2017, Atlanta, GA, USA, June 5-8, 2017*, pages 1586–1596. IEEE Computer Society, 2017.
- 35 Ha Thi Thu Doan, Adrián Riesco, and Kazuhiro Ogata. An environment for specifying and model checking mobile ring robot algorithms. In Mohsen Ghaffari, Mikhail Nesterenko, Sébastien Tixeuil, Sara Tucci, and Yukiko Yamauchi, editors, *Stabilization, Safety, and Security of Distributed Systems - 21st International Symposium, SSS 2019, Pisa, Italy, October 22-25, 2019, Proceedings*, volume 11914 of *Lecture Notes in Computer Science*, pages 111–126. Springer, 2019.
- 36 Fathiyeh Faghih, Borzoo Bonakdarpour, Sébastien Tixeuil, and Sandeep S. Kulkarni. Automated synthesis of distributed self-stabilizing protocols. *Logical Methods in Computer Science*, 14(1), 2018.
- 37 Paola Flocchini, Giuseppe Prencipe, and Nicola Santoro, editors. *Distributed Computing by Mobile Entities*, volume 11340 of *Lecture Notes in Computer Science, Theoretical Computer Science and General Issues*. Springer Nature, 2019.
- 38 Paola Flocchini, Giuseppe Prencipe, Nicola Santoro, and Peter Widmayer. Gathering of asynchronous oblivious robots with limited visibility. In Afonso Ferreira and Horst Reichel, editors, *STACS 2001, 18th Annual Symposium on Theoretical Aspects of Computer Science, Dresden, Germany, February 15-17, 2001, Proceedings*, volume 2010 of *Lecture Notes in Computer Science*, pages 247–258. Springer, 2001.
- 39 Paola Flocchini, Giuseppe Prencipe, Nicola Santoro, and Peter Widmayer. Pattern formation by anonymous robots without chirality. In Francesc Comellas, Josep Fàbrega, and Pierre Fraigniaud, editors, *SIROCCO 8, Proceedings of the 8th International Colloquium on Structural Information and Communication Complexity, Vall de Núria, Girona-Barcelona, Catalonia, Spain, 27-29 June, 2001*, volume 8 of *Proceedings in Informatics*, pages 147–162. Carleton Scientific, 2001.
- 40 Paola Flocchini, Giuseppe Prencipe, Nicola Santoro, and Peter Widmayer. Arbitrary pattern formation by asynchronous, anonymous, oblivious robots. *Theoretical Computer Science*, 407(1-3):412–447, 2008.
- 41 Georges Gonthier. Formal Proof—The Four-Color Theorem. *Notices of the AMS*, 55(11):1382–1393, December 2008.
- 42 Georges Gonthier. Engineering Mathematics: the Odd Order Theorem Proof. In Roberto Giacobazzi and Radhia Cousot, editors, *POPL*, pages 1–2. ACM, 2013.
- 43 Rachid Guerraoui, Thomas A. Henzinger, Barbara Jobstmann, and Vasu Singh. Model checking transactional memories. In Rajiv Gupta and Saman P. Amarasinghe, editors, *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pages 372–382. ACM, 2008.
- 44 Adam Heriban, Xavier Défago, and Sébastien Tixeuil. Optimally gathering two robots. In Paolo Bellavista and Vijay K. Garg, editors, *Proceedings of the 19th International Conference on Distributed Computing and Networking, ICDCN 2018, Varanasi, India, January 4-7, 2018*, pages 3:1–3:10. ACM, 2018.
- 45 Adam Heriban and Sébastien Tixeuil. Mobile robots with uncertain visibility sensors. In Keren Censor-Hillel and Michele Flammini, editors, *Structural Information and Communication Complexity - 26th International Colloquium, (SIROCCO 2019)*, volume 11639 of *Lecture Notes in Computer Science*, pages 349–352, L'Aquila, Italy, July 2019. Springer-Verlag.
- 46 Taisuke Izumi, Tomoko Izumi, Sayaka Kamei, and Fukuhito Ooshita. Randomized gathering of mobile robots with local-multiplicity detection. In Rachid Guerraoui and Franck Petit, editors, *Stabilization, Safety, and Security of Distributed Systems, 11th International Symposium, SSS 2009, Lyon, France, November 3-6, 2009, Proceedings*, volume 5873 of *Lecture Notes in Computer Science*, pages 384–398. Springer, 2009.
- 47 Tomoko Izumi, Taisuke Izumi, Sayaka Kamei, and Fukuhito Ooshita. Mobile robots gathering algorithm with local weak multiplicity in rings. In Boaz Patt-Shamir and Tınaz Ekim, editors, *Structural Information and Communication Complexity, 17th International Colloquium, SIROCCO 2010, Sirince, Turkey, June 7-11, 2010, Proceedings*, volume 6058 of *Lecture Notes in Computer Science*, pages 101–113. Springer, 2010.
- 48 Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an operating system kernel. *Communications of the ACM*, 53(6):107–115, 2010.
- 49 Igor Konnov, Helmut Veith, and Josef Widder. Who is afraid of model checking distributed algorithms? Unpublished to: CAV Workshop (EC)², July 2012.
- 50 Philipp Küfner, Uwe Nestmann, and Christina Rickmann. Formal Verification of Distributed Algorithms - From Pseudo Code to Checked Proofs. In Jos C. M. Baeten, Thomas Ball, and Frank S. de Boer, editors, *IFIP TCS*, volume 7604 of *Lecture Notes in Computer Science*, pages 209–224, Amsterdam, The Netherlands, September 2012. Springer-Verlag.
- 51 Sandeep S. Kulkarni, Borzoo Bonakdarpour, and Ali Ebnenasir. Mechanical verification of automatic synthesis of fault-tolerant programs. In Sandro Etalle, editor, *Logic Based Program Synthesis and Transformation, 14th International Symposium, LOPSTR 2004, Verona, Italy, August 26-28, 2004, Revised Selected Papers*, volume 3573 of *Lecture*

- Notes in Computer Science*, pages 36–52. Springer, 2004.
- 52 Leslie Lamport. On interprocess communication. part I: basic formalism. *Distributed Comput.*, 1(2):77–85, 1986.
 - 53 Leslie Lamport. On interprocess communication. part II: algorithms. *Distributed Comput.*, 1(2):86–101, 1986.
 - 54 Leslie Lamport and Stephan Merz. Specifying and verifying fault-tolerant systems. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863 of *Lecture Notes in Computer Science*, pages 41–76. Springer-Verlag, 1994.
 - 55 Xavier Leroy. A Formally Verified Compiler Back-End. *Journal of Automated Reasoning*, 43(4):363–446, 2009.
 - 56 Assia Mahboubi. Checking machine-checked proofs. <https://project.inria.fr/coqexchange/checking-machine-checked-proofs/>, July 2017.
 - 57 Vincent Rahli Mark Bickford, Robert L. Constable. Logic of events, a framework to reason about distributed systems. In *2012 Languages for Distributed Algorithms Workshop*, Philadelphia, PA, 2012.
 - 58 Kenneth L. McMillan and Oded Padon. Ivy: A multi-modal verification tool for distributed algorithms. In Shuvendu K. Lahiri and Chao Wang, editors, *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II*, volume 12225 of *Lecture Notes in Computer Science*, pages 190–202. Springer, 2020.
 - 59 Laure Millet, Maria Potop-Butucaru, Nathalie Sznajder, and Sébastien Tixeuil. On the synthesis of mobile robots algorithms: The case of ring gathering. In Pascal Felber and Vijay 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, September 2014. Springer-Verlag.
 - 60 Takashi Okumura, Koichi Wada, and Xavier Défago. Optimal rendezvous l-algorithms for asynchronous mobile robots with external-lights. In Jiannong Cao, Faith Ellen, Luis Rodrigues, and Bernardo Ferreira, editors, *22nd International Conference on Principles of Distributed Systems, OPODIS 2018, December 17-19, 2018, Hong Kong, China*, volume 125 of *LIPICs*, pages 24:1–24:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.
 - 61 Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. Ivy: safety verification by interactive generalization. In Chandra Krintz and Emery Berger, editors, *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 614–630. ACM, 2016.
 - 62 Maria Potop-Butucaru, Nathalie Sznajder, Sébastien Tixeuil, and Xavier Urbain. Formal methods for mobile robots. In Paola Flocchini, Giuseppe Prencipe, and Nicola Santoro, editors, *Distributed Computing by Mobile Entities, Current Research in Moving and Computing*, volume 11340 of *Lecture Notes in Computer Science*, pages 278–313. Springer, 2019.
 - 63 Giuseppe Prencipe. Impossibility of gathering by a set of autonomous mobile robots. *Theoretical Computer Science*, 384(2-3):222–231, 2007.
 - 64 Arnaud Sangnier, Nathalie Sznajder, Maria Potop-Butucaru, and Sébastien Tixeuil. Parameterized verification of algorithms for oblivious robots on a ring. *Formal Methods Syst. Des.*, 56(1):55–89, 2020.
 - 65 Ichiro Suzuki and Masafumi Yamashita. Distributed Anonymous Mobile Robots: Formation of Geometric Patterns. *SIAM Journal of Computing*, 28(4):1347–1363, 1999.
 - 66 Giovanni Viglietta. Rendezvous of two robots with visible bits. In Paola Flocchini, Jie Gao, Evangelos Kranakis, and Friedhelm Meyer auf der Heide, editors, *Algorithms for Sensor Systems - 9th International Symposium on Algorithms and Experiments for Sensor Systems, Wireless Networks and Distributed Robotics, ALGOSENSORS 2013, Sophia Antipolis, France, September 5-6, 2013, Revised Selected Papers*, volume 8243 of *Lecture Notes in Computer Science*, pages 291–306. Springer-Verlag, 2013.
 - 67 Vladimir Voevodsky. An experimental library of formalized mathematics based on the univalent foundations. *Mathematical Structures in Computer Science*, 25(5):1278–1294, 2015.
 - 68 Yukiko Yamauchi and Masafumi Yamashita. Randomized Pattern Formation Algorithm for Asynchronous Oblivious Mobile Robots. In Fabian Kuhn, editor, *Distributed Computing - 28th International Symposium, (DISC 2014)*, volume 8784 of *Lecture Notes in Computer Science*, pages 137–151, Austin, USA, October 2014. Springer-Verlag.