Line-Recovery by Programmable Particles

Giuseppe Antonio Di Luna[*], Paola Flocchini[*], Giuseppe Prencipe[†], Nicola Santoro[‡], Giovanni Viglietta[*]

July 24, 2017

Abstract

Shape formation has been recently studied in distributed systems of programmable particles. In this paper we consider the *shape recovery* problem of restoring the shape when f of the n particles have crashed. We focus on the basic *line* shape, used as a tool for the construction of more complex configurations.

We present a solution to the *line recovery* problem by the non-faulty anonymous particles; the solution works regardless of the initial distribution and number $f < n - 4$ of faults, of the local orientations of the non-faulty entities, and of the number of non-faulty entities activated in each round (i.e., semi-synchronous adversarial scheduler).

1 Introduction

The problems arising in distributed systems composed of autonomous mobile computational entities has been extensively studied, in particular the class of *pattern formation* problems requiring the entities to move in the space where they operate until, in finite time, they form a given pattern (modulo translation, rotation, scaling, and reflection), and terminate (e.g., [1, 3, 15–17, 22]).

Very recently, other types of distributed computational universes have been started to be examined (e.g., [14]), most significantly those arising in the large inter-disciplinary field of studies on *programmable matter* [20], that is matter that has the ability to change its physical properties and appearence (e.g., shape, color, etc.) based on user input or autonomous sensing. Programmable matter is typically viewed as a very large number of very small (possibly nano-level) computational particles that are programmed to collectively perform some global task by means of local interactions. Such particles could have applications in a variety of important situations: smart materials, autonomous monitoring and repair, minimal invasive surgery, etc.

Several theoretical models for programmable matter have been proposed, ranging from DNA self-assembly systems, (e.g., [18]) to metamorphic robots, (e.g., [21]), to nature-inspired synthetic insects and micro-organisms (e.g., [11, 13]). Among them, the *geometric Amoebot* model [2, 6–8, 10, 12] is of particular and immediate interest from the distributed computing viewpoint. In fact, in this model (introduced in [11]) programmable matter is viewed as a swarm of decentralized autonomous self-organizing entities (also called *particles*, operating on an hexagonal tessellation of the plane. These particles have simple computational capabilities (they are finite-state machines), strictly local interaction and communication capabilities (only with particles located in neighboring nodes of the hexagonal grid), and limited motorial capabilities (they can move only to empty neighboring nodes); furthermore, time is divided into round, and their activation at each round is controlled by an adversarial (but fair) scheduler; the scheduler is said to be sequential, fully synchronous, and arbitrary (or semi-synchronous) depending on

[*]School Electrical Engineering and Computer Science, University of Ottawa, Canada.
[†]Dipartimento di Informatica, University of Pisa, Italy
[‡]School of Computer Science, Carleton University, Canada.

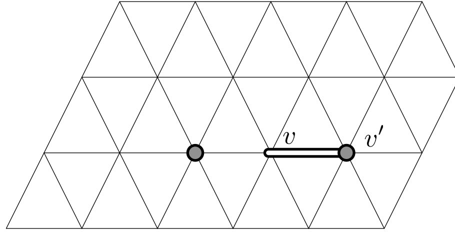


Figure 1: A fragment of a triangular grid with two particles(grey circles). One particle is expanded, with the head in v' and the tail in v; the other is contracted.

whether it activates at each round only one particles, all particles, or an arbitrary subset of them, respectively. A characteristic feature of the Amoebot model is that, at any round, a particle can be *contracted* (occupying one node) or *expanded* (occupying two adjacent nodes); it is through expansions and contractions that particles move on the grid. In this model, the research focus has been on applications such as *coating* [7,12], *gathering* [2], and *shape formation* [4,6,9–11]. The shape formation (or pattern formation) problem is prototypical for systems of self-organizing programmable particles, and particular attention has been given to special basic shapes like the *line* [4,9,11], that is used as a tool for the construction of more complex configurations.

A common feature of the existing studies on these programmable particles is that all the particles are assumed to be fully operational at all times; that is, faults have never been considered. In this paper we address the presence of faulty particles.

We consider a connected shape of n particles of which f are faulty, and the faults are *crashes*. We are interested in the problem of the non-faulty particles efficiently re-configuring themselves so to form the same shape without including any faulty particles. We call this problem *shape recovery*, and is a basic task of self-reconstruction/self-repair for a prescribed shape.

In this paper we study this problem when the shape the particles form is the *line*, hence the problem becomes that of *line recovery*. Solving this problem requires formulating a set of rules (the algorithm) that will allow the non-faulty entities to form the line within finite time, regardless of the initial distribution and number of faults and of the local orientations of the non-faulty entities. Unfortunately this task, as formulated, is actually *unsolvable*, even with a fully synchronous scheduler. In fact, there are initial configurations where unbreakable symmetries make it impossible to form a single line. We thus require that either one or two lines of equal size be formed, depending on the symmetry level of the initial configuration. This problem has been studied in [5] in a different computational setting, and solved in the case of a square grid assuming a fully synchronous scheduler.

In this paper we solve the *line recovery* problem in the Amoebot model under a semi-sysnchronous adversarial scheduler. We present a line recovery algorithm allowing $n - f > 4$ non-faulty particles without chirality to correctly form either a single line or two lines of equal size, regardless of the position of the faulty particles and of the number of non-faulty ones activated at each round.

2 Model

We consider the space to be an infinite unoriented anonymous triangular grid $G(V, E)$, where the nodes in V are all equal and edges are bidirectional (see Figure 1). In the system there is a set P of n particles, initially located at distinct positions in G. A subset $F \subset P$ of the particles, with $|F| = f$, is *faulty*: a faulty particle does not move or communicate with other particles. The other particles are said to be *correct*; the subset of correct particles is denoted by $C = P \setminus F$.

A particle p assigns to each incident edge a distinct port number; this numbering is local and

we do not assume that the particles agree on a common clockwise direction. Two particles that are neighbours in G form a *bond*. Each particle has a shared constant size memory associated to each of its local ports, that can be read and written also by a neighbour particle. Moreover, each particle has a constant size memory used to store its state.

To ease the writing, in our algorithm we will use a message passing terminology, in which, when a particle sends a message to a neighbour, it writes on the shared memory of the receiving neighbour. Symmetrically, if a particle receives a message, it will find it in its shared memory.

The system works in rounds, and particles are *activated* by an external semi synchronous schedule: at each round r the scheduler selects a subset of correct particles, $C_r \subseteq C$, and it activates them; at the same round, the particles in $C \setminus C_r$ are inactive. The scheduler is fair, in the sense that it has to activate each particle infinitely often. A particle p moves by a sequence of *expansions* and *contractions*: a contracted particle occupies a single node $v \in V$, while an expanded one occupies two neighbours node nodes. Initially, each particle occupies exactly a single node; i.e., all particles are contracted. During the execution of the algorithm, a correct contracted particle p that is in v might expand: after the expansion, p will occupy two nodes: v and the neighbour node w where it expanded to. We will say that node w is the *head* of the particle, and v is the *tail*. Particle p always knows which node is its head and which one is its tail. If a particle is expanded, it can contract back in either tail or head node (if a particle is contracted, node and tail are the same). We assume that a particle q that is a neighbour of p knows if p is contracted or expanded, and it knows if it is bonded with the tail or the head of p. Also, particles are endowed with a failure detector, that takes as input a local port number and returns true if its neighbour (if any) is in F.

Upon activation at round r, a particle p executes the following operations:

Look: It reads the shared memories of its local ports, the shared memories of the local ports of its neighbours (if any), and the output of the failure detector on each port.

Compute: Using these information and its local state, it performs some local computations. Then, it updates its internal memory and it possibly writes on the shared memories of its neighbours. As an outcome of the Compute operation, p can either decide to stay still or to Move.

Move: If in the Compute operation p decides to move, then it can either expand to an occupied neighbour location, if contracted at round r, or contracts towards its head or tail, if expanded. Moreover, it can perform a special operation called *handover*, in which it forces the movement of a correct neighbour particle q: if q is expanded and p is contracted, then p forces the contraction of q, by pushing q towards its head/tail occupying the tail/head of q; otherwise, if q is contracted and p is expanded, then p contracts towards its head/tail forcing the expansion of q in the tail/head of p.

Since the scheduler can activate more than one particles in the same round, it is crucial to specify what happens in case of conflicting operations executed by different particles:

(i) If two or more particles try to expand in the same node v, then only one succeeds, and the decision depends on the scheduler. The particle that fails to move, will be aware of this at the next activation, by realising that it is still contracted.

(ii) If two or more particles try to execute an handover with a particle p and p is moving, then only one will succeed (which one depends on the scheduler). The ones that fail to move will be aware of this at the next activation, by realizing that they have not moved.

(iii) If a particle p tries to execute an handover with a particle q, and q is moving, then the handover succeeds if and only if also q is executing the same handover operation with p. Otherwise, q moves, p fails the handover, and p will be aware of this at the next activation, by realizing that it has not moved.

Given a set of particles P, we say that they are on a *compact straight line* if they are on a *straight line* and the subgraph induced by their positions is connected. Initially, at round $r = 0$, all particles (both correct and faulty) are positioned on a compact straight line (the "initial

line"). In the following, we will assume that $n - f \geq 5$. The LineRecovery problem is solved at round r^* if, for any round $r \geq r^*$, the particles in C either form a straight line with an unique leader particle, or they form two straight lines of equal size each one with its own leader particle.

3 Line Recovery Algorithm

3.1 Overall Description

In the following, we will assume that the particles can exchange fixed-size messages (this can be easily simulated in our model). Also, if not otherwise specified, the variable *dir* of a particle p stores the movement's direction of p; that is, it stores the port number where p intends to move; when no ambiguity arises, we will use the expression "direction of p" to indicate the content of *dir*. Similarly, the content of variable *pre* stores the location of p in the previous round; again, when no ambiguity arises, we will use the expression "previous location of p" to denote the content of this variable. Moreover, we will say that p is *pointing at* a particle p' if p and p' are neighbors, and the direction *dir* of p is toward the location occupied by p'. Finally, when a particle p changes state from s, we will say that p *becomes* s.

Let L_0 be the line where the particles are placed at the beginning, and L_1, L_{-1} be the two lines adjacent to L_0. The overall idea to solve the problem is as follows: first, the particles, try to elect either one or two leaders. The election is achieved by having few selected particles move on L_1 and L_{-1}; also, during this movements, no gap of size larger than 3 is ever created: this is a crucial invariant to keep, to correctly understand whether a particle is on one of the two extremes of the line. The role of the leader(s) is to start the construction of the final straight line. If there is only one leader, it will perform a complete tour around L_0 (i.e., moving on L_1 and L_{-1}): during this tours it collects all correct particles, that will follow the leader(s), by attaching to the line they are building. If there are two leaders, then one must be placed on L_1 and the other on L_{-1}. Each one will build a line of correct particles, and then they will compare the length of such lines. If the lines are not equal the symmetry is broken and a single line of correct particles will be formed; otherwise, two equal sized lines will be formed.

In more details, the LineRecovery algorithm is divided in seven sub-algorithms: Fault Checking, Explorer Creation, Candidate Creation, Candidate Checking, Unique Leader and Opposite Sides.

The algorithm starts by checking whether there are no *faults*: in this case, all particles in C are already forming a compact line. This scenario is detected during the Fault Checking sub-algorithm, started by the particles who occupy the extreme positions of the starting configuration, i.e., by the two particles having only one neighbor, these particles get state *marker*. These two extreme particles send a special message inside the line: if the two messages meet, there are no *faults*.

Should there be *faults*, the second sub-algorithm (the Explorer Creation) is performed, started by all the *particles* who have a *faulty* neighbor. In this sub-algorithm some *particles* become *explorers* and move out of the line (either on L_1 or on L_{-1}). The selection of the *explorers* is made in such a way that their movement does not create "gaps" of more than two consecutive empty positions anywhere in the original line (this property is crucial to detect the end of the line in subsequent sub-algorithms).

3.2 Sub-Algorithms

Fault Checking. In the Fault Checking sub-algorithm (reported in Figures 2 and 3) the particles detect if there are no *faults*. If so, they elect either one or two *nofaulty.leaders*. In case two *nofaulty.leaders* are elected, two lines having exactly the same size will be formed.

In the following, we will use the following convention: in the pseudo-code, when there is an **If** statement that checks the presence of a particular message (i.e., Line 48), and the guard of

```
1: Upon Activation in State Init do:
2:     Set Line L_0=getLineDirectionFromActivatedInitAndFaultyNeighbours()
3:     if ∃p ∈ Ports|msg.switchtoslave ∈ p then
4:         Set State slave
5:     else if (∃!p ∈ Neighbours|(p.state = Init ∨ p.state = starting ∨ faulty(p)) then
6:         Set flag.linetail
7:         lineparity = 1
8:         send(p, msg.coin)
9:         Set State marker
10:    else
11:        Set State starting
12: End
13:
14: Upon Activation in State marker do:
15:    if ∃p ∈ Ports|msg.markertoleader ∈ p then
16:        Set State nofaulty.leaders
17:    else if ∃p ∈ Ports|msg.asktobecandidate ∈ p ∧ ¬flag.candidate then
18:        Set flag.candidate
19:        send(p, msg.candidate)
20:    else if ∃port ∈ Ports|msg.switchtoleader ∈ port then
21:        Set direction and flags from msg.switchtoleader
22:        Set State leader
23:    else if ∃p ∈ Neighbours|p = probe ∧ contracted(p) ∧ ∃p′ ∈ Neighbours|p′ = collector.counting then
24:        if ∄msg.seen ∈ p then
25:            send(p, msg.seen)
26:            send(p′, msg.other)
27:    else if ∃p ∈ Neighbours|p = collector.done ∧ contracted(p) ∧ ∃p′ ∈ Neighbours|p′ =
   collector.counting then
28:        send(p′, msg.winner)
29:    else if ∃p ∈ Neighbours|p = collector.done ∧ contracted(p) ∧ ∃msg.done ∈ opposite(p) then
30:        send(p′, msg.even)
31:        Set State follower
32: End
```

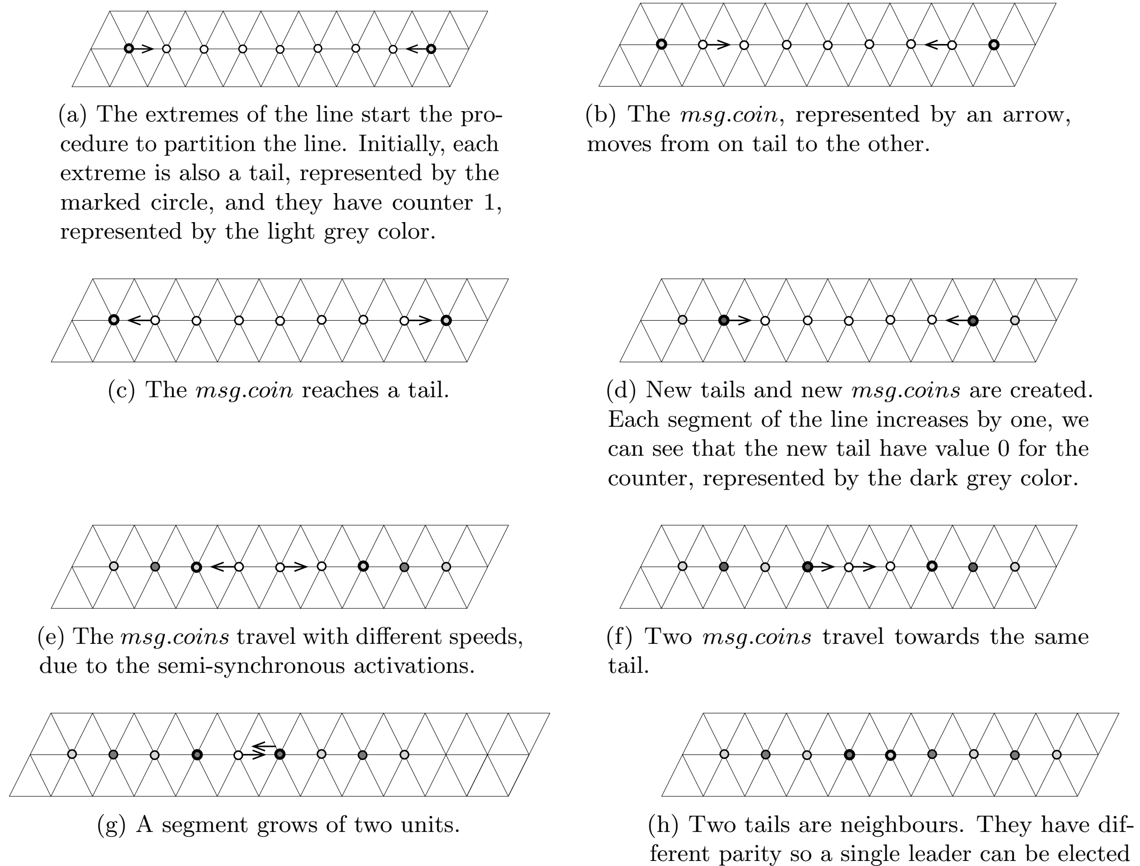
Figure 2: Algorithm for *Init*, *marker* and starting – Part One

32: **Upon Activation in State** starting **do:**
33: **if** $\exists p \in Ports|msg.switchtoslave \in p$ **then**
34: Set State *slave*
35: End Cycle
36: **else if** $\exists p \in Ports|msg.coin \in p$ **then**
37: **if** $\exists neighbour \in opposite(p) \wedge neighbour.linetail$ **then**
38: $send(neighbour, msg.newtail)$
39: $parent = neighbour$
40: Set $flag.linetail$
41: $lineparity = neighbour.lineparity + +$
42: $send(p, msg.coin)$
43: **if** $\exists neigh \in opposite(p) \wedge neigh \neq Init \wedge \nexists msg.coin \in neigh.port$ **then**
44: $send(neigh, msg.coin)$
45: **else**
46: Add $msg.coin$ to p
47: **else if** $flag.linetail$ **then**
48: **if** $\exists p \in Ports|msg.newtail \in p$ **then**
49: Unset $flag.linetail$
50: **if** $\exists neighbour \in Neighbours \wedge neighbour \neq parent \wedge neighbour.flag.linetail$ **then**
51: $p = opposite(neighbour)$
52: **if** $lineparity = 0 \vee lineparity = neighbour.lineparity$ **then**
53: $send(p, msg.markertoleader)$
54: **if** $\exists p \in Ports|msg.markertoleader \in p$ **then**
55: $p = opposite(neighbour)$
56: $send(p, msg.markertoleader)$
57: **if** $(\exists p \in L_0|faulty(p)) \wedge (\exists p' \in L_0|(p' \neq p \wedge p' \neq Init \wedge (\nexists port \in Ports|msg.notified \in port))$ **then**
58: $send(p', msg.notify)$
59: Set State pre.*explorer*
60: **if** $(\exists! port \in Ports|msg.notify \in port) \wedge (\nexists p \in L_0|p = Init)$ **then**
61: $send(opposite(port), msg.notified)$
62: Add $msg.notify$ to $port$
63: Set State notified
64: **End**

Figure 3: Algorithm for *Init*, *marker* and starting – Part Two

the **If** statement is **true**, then the message is immediately deleted from the memory (i.e., when Line 49 is executed on particle p, the *msg.newtail* of the statement of 48 is deleted from the memory of p).

This routine is started by the particles who occupy the extreme positions of the starting configuration, i.e., by the two particles having only one neighbor, called *marker*. The general idea behind this sub-algorithm is as follows: each *marker* generates a special message that travels towards the other *marker*; if the two messages meet, than there are no *faulty* particles. In particular, two *msg.coin* messages are generated, one from each *marker*, travelling in opposite directions; the other particles, when activated for the first time, switch from *Init* to starting state, and save the direction of the initial line L_0 in its local memory. Without loss of generality, let m_l and m_r be the leftmost and rightmost *marker*, respectively; also, let us denote by S_l and S_r the *segments* of m_l and m_r, respectively: at the beginning, S_l and S_r contain only m_l and m_r, respectively (i.e., each *marker* is both the start and the end of its own segment). When the *msg.coin* sent by m_l reaches the end S_r, S_r expands of one unit towards m_l, and the *msg.coin* is sent back to m_l; symmetrically, when the *msg.coin* sent by m_r reaches the end S_l, S_l expands of one unit towards m_r, and the *msg.coin* is sent back to m_r. During this expansion process, the end of each segment stores the parity of the length of the segment it belongs to. This process is iterated until, if there are no *faulty* particles, the ends of the two segments will be neighbors: when this occurs, if the two segments have the same size, two leaders will be elected; otherwise their sizes differ by at most one unit and a unique leader is elected, that is the *marker* of the segment with length of parity 0. An example run is in Figure 4.



(a) The extremes of the line start the procedure to partition the line. Initially, each extreme is also a tail, represented by the marked circle, and they have counter 1, represented by the light grey color.

(b) The *msg.coin*, represented by an arrow, moves from on tail to the other.

(c) The *msg.coin* reaches a tail.

(d) New tails and new *msg.coins* are created. Each segment of the line increases by one, we can see that the new tail have value 0 for the counter, represented by the dark grey color.

(e) The *msg.coins* travel with different speeds, due to the semi-synchronous activations.

(f) Two *msg.coins* travel towards the same tail.

(g) A segment grows of two units.

(h) Two tails are neighbours. They have different parity so a single leader can be elected

Figure 4: Run of the algorithm where the Fault Check subphase elects an unique leader.

Lemma 1. *By executing procedure* Fault Checking*, the size of the two segments differ by at most* 1 *unit.*

Proof. We first show that the procedure that expands the two tails forces the size of the two segments to differ by at most 1. In fact, the size of a segment is the number of nodes between a *marker* and its respective tail. The proof is by induction on the number of rounds. Let r be the first round at which a *marker* is created. At the end of round r, if the *marker* exists, each segment has size 1; otherwise the segments have both size 0. Therefore the claim is verified.

Let us suppose that the claim holds until round $r + k$. At round $r + k + 1$ each segment may increase its size of at most one unit; if at round $r + k$ the difference between the size of the two segments was 0, then the statement is verified. Otherwise, if the difference was 1, then we need to show that the bigger segment, say s, does not increase until the smaller one, say s', increases by one unit.

A segment, to increase its size, has to consume a message *msg.coin*. This message can only be produced by the tail of the other segment, and it is produced only when the segment grows by one unit. Notice that there is no *msg.coin* travelling from the tail of the smaller segment s' to s. Otherwise, the difference in size between the two segments would have been more than 1 at a round $r' < r + k$: s' would have grown of one unit, and still be smaller than s. □

Theorem 1. *If $f = 0$, then the procedure* Fault Checking *correctly solves the* LineRecovery *problem.*

Proof. It is immediate to see that, as long as the two tails are not neighbours, a segment will eventually grow. We distinguish two cases:

1. If the line has an even number of particles, then Fault Checking divides the initial line in two segments of equal size. By contradiction, let us assume that one of the two segments is greater than the other, and that two tails are touching. Being the line even, this can only be possible if one of the two segments is two units or more longer than the other one: by Lemma 1, this is not possible, and two *markers* are both elected as leader.

2. If the line has an odd number of particles, and the two tails are touching, we have that only one of the segment has an odd size. In fact, they cannot have both odd size, otherwise the initial line would have an even number of particles, having a contradiction. Therefore, the two tails have different parities, and by executing Fault Checking only one leader is elected.

In both cases, the theorem follows. □

Explorer Creation. The sub-algorithm Explorer Creation is used to bootstrap the other sub-algorithms: its execution is started from sub-algorithm Fault Checking if in the system there is at least one *faulty* particle. The main purpose of this sub-algorithm is to select, among the correct particles, at least three *explorers*, who will move out of line L_0 without creating empty "gaps" of more than two consecutive positions. This is done as follows.

```
1: Upon Activation in State pre.explorer do:
2:     if ∃port ∈ Ports|msg.switchtoslave ∈ port then
3:         Set State slave
4:         End Cycle
5:     l = getNeighbourOutside(L_0)
6:     direction = left
7:     move(l)
8:     Set State explorer
9: End
10:
11: Upon Activation in State notified do:
12:     if ∃port ∈ Ports|msg.switchtoslave ∈ port then
13:         Set State slave
14:         End Cycle
15:     cond_1 := ∄p ∈ Ports|msg.notified ∈ p
16:     cond_2 := ∄p ∈ opposite(port(msg.notify))|p = pre.explorer
17:     cond_3 := ∄msg.notify ∈ opposite(port(msg.notify)
18:     if (cond_1 ∧ cond_2 ∧ cond_3) then
19:         Set State pre.explorer
20: End
```

Figure 5: Algorithm for *pre.explorer* and *notified*

If a particle in *starting* state (from Fault Checking) has a *faulty* neighbour, then it becomes *pre.explorer*, and it notifies this decision to any non *faulty* neighbouring particle. If the neighbour is correct but it is still in the *Init* state, it waits (see Line 57 of Figure 3).

A *starting* particle that, upon activation, finds such a notification message, becomes *notified* (Line 60 of Figure 3). A *notified* particle that is not a *marker* becomes a *pre.explorer* if, on the opposite port to the one containing the notification, there is no neighbour that will become either *notified* or *pre.explorer*; otherwise, it stays in the *notified* state. Note that, when a particle changes state to *notified*, it sends a message to its other neighbour, to avoid that it also becomes a *pre.explorer*.

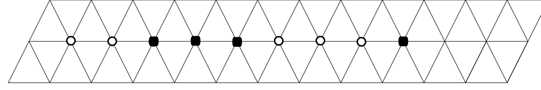
If a *pre.explorer* is activated, it becomes an *explorer*, it moves outside L_0 on L_j with $j \in \{1, -1\}$, and it picks as direction on L_j the left one, according to its chirality (see Lines 5-8 of Figure 5). The direction is stored in the local variable *direction*, that is always pointing to some location on line L_j on the left of the current one according to the handedness of the particle. Notice that the particle might fail to leave L_0, because of collisions with other particles: to handle this case, an *explorer* that finds itself on line L_0 tries to expand to go outside L_0.

Let a *sequence of particles* be a set of consecutive particles; also, let a *gap* be the maximum number of empty locations between two particles (either correct or *faulty*) on line L_0. An example run of Explorers creation is in Figure 6.

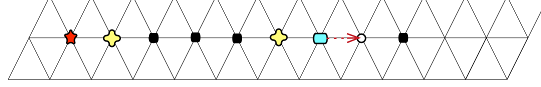
Lemma 2. *Sub-algorithm* Explorer Creation *never creates a gap of size* 3*. Moreover, for each sequence of correct particles of size greater or equal* 3*, at least two pre.*explorer *will be created.*

Proof. The proof considers runs of correct particles of different size. First, it is easy to see that for any run of size 2 we cannot create a gap of size 3.

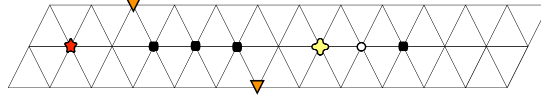
For a run of size 3, let p_1, p_2, p_3 be the placement of the three correct particles. First, notice that at least two *pre.explorers* will be created: either the two immediate neighbours of a *faulty* particle, or an immediate neighbour of a *faulty* and a *notified*. Without loss of generality, let r be the first round at which particle p_1 becomes a *pre.explorer*. Also, let $r' \geq r$ be the first round at which the particle p_2 wakes up after r. If also p_3 becomes a *pre.explorer* before round r', then p_2 receives two notify messages. Therefore, p_2 will not became *notified*; hence, it cannot become *pre.explorer*. Otherwise, if p_3 becomes *pre.explorer*



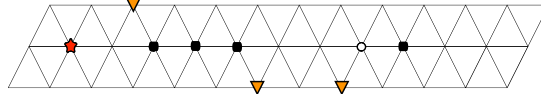
(a) Initial coinfiguration: we have a sequence of two correct particle, a sequence of 3 faulty particles, a sequence of 3 correct particles and a faulty particle at the end of the line.



(b) The left endpoint of the line becomes a *marker*, red star. The pre-*explorers* are the yellow crosses and they are neighbours of faulty particles. The notified particle is the cyan rectangle and it is sending a notified msg to the white particle, that is still in the *starting* state. This implies that the white particle will never become an *explorer*.



(c) The pre-*explorers* become *explorers* and the notified becomes a pre-*explorer*.



(d) All the *explorers* have been created.

Figure 6: Explorer Creation, example run.

at round r', p_2 receives an additional notify message at round r'; so, when p_2 is activated after becoming *notified*, the predicate that leads the particle to be a *pre.explorer* cannot be verified ($\exists! port \in Ports | msg.notify \in port$, see Line 18 of Figure 5), hence p_2 cannot become *pre.explorer*. Finally, by predicate $(\exists p' \in L_0 | (p' \neq p \wedge p' \neq Init \wedge (\nexists port \in Ports | msg.notified \in port))$, p_3 cannot become a *pre.explorer* after round r'.

For a run of size 4, let f, p_1, p_2, p_3, p_4, f be the placement of the particles, with f the faulty processes, and let us examine the round r at which p_2 becomes *notified*. Let us suppose, w.l.o.g., that r is the first round at which a particle becomes *notified*. If at r particle p_3 does not become *notified*, then notice that p_3 cannot become a *pre.explorer* either (by predicate $(\nexists port \in Ports | msg.notified \in port)$), so a gap of size 3 cannot be created. If at r also p_3 becomes *notified*, then, for the same predicate, both p_3 and p_2 cannot become *pre.explorer*. Also, notice that at least two *pre.explorers* will be created: the two immediate neighbours of a *faulty*. Notice, that if the run is f, p_1, p_2, p_3, p_4, with p_4 the last process of the initial line L_0, then both p_1, p_2 become *pre.explorer*; thus also in this case at least two *pre.explorers* are created.

In the general case of a run of size greater than 4, notice that only the immediate neighbour of a *faulty* particle and its neighbour can become *pre.explorer*. So, also in this case, a gap of size 3 cannot be created. It thus follows that also in this case at least 2 *pre.explorers* will be created, and the lemma follows. □

Observation 1. *If at round $r = 0$, there is a correct particle p at the end of the initial line, then this particle will become a* marker*.*

Proof. If this particle has a non-faulty neighbour on the line, then such neighbour will not move

until p is in the *Init* state. When p is activated, it will verify that it has only one neighbour in state *starting*, and it will thus become a *marker*. The same occurs if the neighbour on the line is *faulty*. □

Lemma 3. *There exists a round r in which there is either: (1) a* marker *and at least two* pre.explorers*, or (2) at least three* pre.explorers.

Proof. Let us first examine the case in which, at round $r = 0$ we have a run of at least 3 correct particles near the end of the line. By Lemma 2 and Observation 1, the lemma follows.

Let us now examine the case in which the run has at least 2 correct particles near the end of the line. By Observation 1, there is at least one *marker*, and the neighbour of the *marker* will become a *pre.explorer*.

By hypothesis, $n - f \geq 5$; therefore, if there is a run of 3 correct particles, by Lemma 2, at least two *pre.explorers* will be created, and the lemma holds. Otherwise, notice that if the run has only 2 correct particles, again at least one *pre.explorer* will be created, and the lemma holds as well. The lemma holds also if there are two different runs, each of size one.

Finally, let us examine the case in which, at round $r = 0$, there is at least 1 correct particle near the end of the line. By Observation 1, there is at least one *marker*. The remaining correct particles form either a run of 3 particles; or a run of two correct particles near the end of the line and another run of at least one correct particle inside the line; or two runs of size less than 3 inside the line. In all these cases the lemma holds.

The last case to consider is when there is no correct particle near the end of the line at round $r = 0$: in this case, if the correct particles are all in one run, we have 4 *pre.explorers*. Otherwise, if there is a run of size 3, then we still have at least 3 *pre.explorers*: two created in the run of size 3, and the third one among the remaining correct particles. If there are two runs of size 2, they will both create 2 *pre.explorers*. The lemma still holds if there are five runs of size one, or a run of size two and three runs of size one. □

Candidate Creation. This sub-algorithm, reported in Figure 7, is executed when $f > 0$ at round $r = 0$. Its main purpose is to elect at most two *explorers*; one of the two elected *explorers* becomes a *candidate*. In this sub-algorithm the *explorers* move along the line until they find either the end of the line, which is detected by seeing three consecutive empty locations, or a *marker*. If an *explorer* meets a particle in the *Init* state, it waits.

The first *explorer* that reaches an extreme of the line without *marker*, becomes a *marker* and stays there. If two *explorers* try to become *marker* on the same end of the line at the same time, only one will succeed (they will both try to move in the same location, see Lines 19-21 and 1-6 of Figure 7). An *explorer* communicates its direction of movement to other *explorer*s by writing an appropriate flag in the shared memory of the port where the head is going to expand. In the following, we will say that the *explorer* is pointing in some direction.

If two *explorers* meet and they have opposing directions, since they cannot pass through each other, they simply switch directions. When an *explorer* switches direction, it sends a message to the other explorer, to ensure that it will also switch direction. This message is also used to ensure that a particle does not switch direction twice with the same particle; that is, the *explorer* checks that the other is pointing at it, and that it has not a pending *msg.changedirection* in the shared memory of the corresponding port (see Lines 22,25 of Figure 7). If an *explorer* finds its next location occupied, it waits. Depending on the initial configuration, either one, two, or no *markers* are created during this procedure.

An *explorer* who reaches a *marker*, sends a message to the *marker* asking to become a *candidate*; if the *marker* accepts, then the *explorer* becomes a *candidate*. If two *explorers* reach the same *marker* and both ask to become a candidate, then the *marker* will answer affirmatively to only one of them (see Lines 27,30 of Figure 7, and Line 17 of Figure 2).

```
1: Upon Activation in State pre.marker do:
2:     if myself.expanded then
3:         contracToHead()
4:         Set State marker
5:     else
6:         Set State explorer
7: End
8:
9: Upon Activation in State explorer do:
10:    if ∃port ∈ Ports|msg.switchtoslave ∈ port then
11:        Set State slave
12:        End Cycle
13:    else if  myself ∈ L_0 then
14:        l = getNeighbourOutside(L_0)
15:        direction = left
16:        move(l)
17:    else if ∃p ∈ L_0|p = Init then
18:        End Cycle
19:    else if ∃l_1, l_2, l_3 ∈ L_0|empty(l_{1,2,3}) = true ∧ l_1, l_2, l_3 are contigous then
20:        pre.marker
21:        expand(l_3)
22:    else if ∃p ∈ getLocation(direction) ∧ p = explorer ∧ pointingAtMe(p) ∧ ∄mgs.changedirection ∈ p
   then
23:        send(p, mgs.changedirection)
24:        direction = opposite(direction)
25:    else if ∃p ∈ Ports|mgs.changedirection ∈ p ∧ direction = p then
26:        direction = opposite(direction)
27:    else if ∃p ∈ L_0 ∧ p = marker then
28:        send(p, msg.asktobecandidate)
29:        Set State slave
30:    else if ∃!p ∈ Port|msg.switchtocandidate ∈ p then
31:        Set direction and flags from msg.switchtocandidate
32:        Set State candidate
33:    else if ∃p ∈ Port|msg.switchtocandidate ∈ p then
34:        Set direction and flags from msg.switchtocandidate
35:        Set State leader
36:    else if ∃p ∈ Ports|msg.switchtoleader ∈ p then
37:        Set direction and flags from msg.switchtoleader
38:        Set State leader
39:    else if (∃!p ∈ Ports|msg.switchopposer ∈ p) then
40:        Set direction and flags from msg.switchopposer
41:        Set State opposer
42:    else if (∃p ∈ Ports|msg.switchopposer ∈ p) then
43:        Set direction and flags from msg.switchopposer
44:        Set State leader
45:    else if ∃p ∈ L_0 ∧ (p = opposer ∨ p = leader) then
46:        Set State slave
47:    else if empty(getLocation(direction)) then
48:        move(getLocation(direction))
49: End
```

Figure 7: Explorer Algorithm

```
1: Upon Activation in State slave do:
2:     cond_1 := ∃p ∈ Ports|msg.candidate ∈ p
3:     cond_2 := ∄p' ∈ Port|msg.switchtocandidate ∈ p'
4:     if cond_1 ∧ cond_2 then
5:         direction = right
6:         Set State candidate
7:     else if cond_1∧!cond_2 then
8:         Set direction and flags from msg.switchtocandidate
9:         Set State leader
10:    else if (∃p ∈ Ports|msg.switchcollector ∈ p then
11:        Set direction and flags from msg.switchcollector
12:        Set State collector
13:    else if ∃port ∈ Ports|msg.switchtoleader ∈ port then
14:        Set direction and flags from msg.switchtoleader
15:        Set State leader
16:    else if ∃port' ∈ Port|msg.switchtocandidate ∈ port' then
17:        Set State candidate
18: End
```

Figure 8: Candidate and Slave Algorithm – Part One

Lemma 4. *Starting from any initial configuration where $f > 0$, there exists a round r in which there is at least one* marker *p that signals the end of the line, and an* explorer *p' moving towards p.*

Proof. Let us first examine the case in which at round $r = 0$ both endpoints, p_1 and p_2 are in C. In this case, it is easy to see that they will both be activated resulting in having two *marker*s at the end of the line. Moreover, by Lemma 3, we have at least three pre.*explorers*; therefore, there will be one *explorer* moving towards one of the *marker*.

In case at round $r = 0$ only one of the endpoints, say p_1, is in C, it will be eventually activated becoming a *marker*. By Lemma 3, we have that at least 2 pre.*explorer* will be created: if at least one of them has the direction of p_1, then the lemma follows. Otherwise (they have the same direction), one of them will either move towards a *marker*, or towards the end of the line with a *faulty* particle, and there are three empty consecutive locations. Thus, the first *explorer*, will become *marker* p and the other will move towards p. Finally, by Lemma 3, it follows that, even if both p_1 and p_2 are not in C, there exists a round in which there are at least three pre.*explorers*: two of them will have the same directions; thus, the previous argument can be applied again, and the lemma follows. □

Candidate Checking. The purpose of the Candidate Checking sub-algorithm is to determinate whether one or two *candidates* have been created.

A newly elected *candidate* has to determine if it is the unique *candidate*; to do so, it switches direction trying to reach the other extremity of the line. While moving, it blocks every neighbour on line L_0, by sending a *msg.switchtoslave* message; this avoids that a new *explorer* is created on the portion of the line that has been visited by the *candidate* (Line 18 of Figure 9).

If the *candidate* meets an *explorer* coming from opposite direction, it "virtually" continues its walk by switching roles with the explorer: the *explorer* becomes *candidate* and switches direction, and the old *candidate* becomes a *slave* and stops. Similarly, if a *candidate* and a *slave* meet, they switch roles (Line 23 of Figure 9).

There are two possible outcomes of this procedure: either a unique *leader* is elected, or not. An unique *leader* can be elected in all the following cases:

- (C1) The *candidate* finds a *marker* that has not elected a candidate, flag $flag.candidate$ is unset; also, it sends to *marker* a message asking to be a candidate, and it receives the

17: **Upon Activation in State** *candidate* **do:**
18: **for all** $p \in L_0 | p \neq marker$ **do**
19: $send(p, msg.switchtoslave)$
20: $cond_1 := \exists p \in getLocation(direction)$
21: **if** $\exists p \in L_0 \wedge (p = candidate \vee p = leader)$ **then**
22: Set State *slave*
23: **else if** $cond_1 \wedge p \neq candidate \wedge contracted(p)$ **then**
24: $send(p, msg.switchtocandidate)$
25: Set State *slave*
26: **else if** $contracted(myself) \wedge cond_1 \wedge p = candidate$ **then**
27: l =location not in L_0 that is neighbour to $myself$ and p.
28: $expand(l)$
29: **else if** $cond_1 \wedge p = candidate \wedge head(p) \notin L_{1,-1}$ **then**
30: Set State *slave*
31: **else if** $head(myself) \notin L_{1,-1} \wedge !cond_1 \wedge p = candidate$ **then**
32: $contractToTail()$
33: Set State *leader*
34: **else**
35: $cond_2 := \exists p \in L_0 \wedge p = marker$
36: **if** $cond_2 \wedge p.flag.candidate$ **then**
37: $direction = oppositeDirection()$
38: Set State *collector*
39: **else if** $cond_2 \wedge \neg p.flag.candidate \wedge \nexists msg.asktobecandidate \in p$ **then**
40: $send(p, msg.asktobecandidate)$
41: **else if** $\exists p \in Port | msg.candidate \in p$ **then**
42: $send(p, msg.switchtoleader)$
43: $direction = p$
44: Set State *follower*
45: **else**
46: $cond_3 := \exists l_1, l_2, l_3 \in L_0$ s.t. $empty(l_{1,2,3}) = true$
47: $cond_4 := l_1, l_2, l_3$ are contigous
48: **if** $cond_3 \wedge cond_4 \wedge contracted(myself)$ **then**
49: $expand(l_3)$
50: **else if** $cond_3 \wedge cond_4 \wedge myhead \in l_3$ **then**
51: Set $flag.firstsideswitch$
52: $direction = getDirectionOppositeToTail()$
53: $contractToHead()$
54: Set State *leader*
55: **else if** $empty(getLocation(direction))$ **then**
56: $move(getLocation(direction))$
57: **End**

Figure 9: Candidate and Slave Algorithm – Part Two

affermative answer from the *marker* (Lines 48,39,41 of Figure 9). This also implies that the *candidate* is unique and no other *candidate* can be created.

- (C2) A *slave* receives message *msg.candidate* (from *marker* p'), and message *msg.switchtocandidate* (see Line 9 of Figure 8).
- (C3) An *explorer* receives on two distinct ports a *msg.switchtocandidate* request. This occurs when there are two *candidates* on the same side, and both tried to switch their role with the same explorer (Line 35 of Figure 7).
- (C4) A *candidate* reaches the other extremity, it finds three empty locations, and it expands occupying the last empty location (Line 50 of Figure 9). This implies that the *candidate* is unique and that no other *candidate* can be created.
- (C5) A *candidate* meets another *candidate* (Line 26 of Figure 9). In this case, each leader knows the position of L_0, and it can identify the unique location l that is neighbour to both and that is not on L_0. l is empty. Both *candidate* try to expand to l; the one that succeed, waits until the other *candidate* becomes a *slave*. When this happens it contracts and it becomes a *leader* (see Lines 26,29,31 of Figure 9).

In all the above cases, the sub-algorithm Unique Leader is executed. We cannot immediately elect a *leader* when the *candidate* reaches the other extremity, finding a *marker* that has elected a candidate (Line 36 of Figure 9). In this last case, the sub-algorithm Opposite Sides is executed. It can still happen that, during the execution of Opposite Sides, the symmetry between *candidates* is somehow broken: in this case, an unique *leader* is elected, as explained in detail in the section describing Opposite Sides.

Lemma 5. *There exists a round r in which there is at least one* candidate*.*

Proof. By Lemma 4, there exists a round when there is at least one *marker* p, and an *explorer* p' moving towards the *marker*. Let this *explorer* be in L_1. Notice, that the *explorer* can only be stopped by a *candidate*, turning it into a *slave*. However, if there exists a *candidate*, the lemma follows.

Therefore, let us assume that no *candidate* exists. Notice that, if p' is blocked by another *explorer* p'' that is pointing at him, then p'' receives a message *mgs.changedirection*. This message forces p'' to eventually get the same direction of p', thus pointing at p. If there is a another *explorer* blocking p'', we can iterate the same argument, until there is no one on L_1 between the last *explorer* p' and a neighbour location of p on L_1. Thus, after a finite number of activations p' will be a neighbour of p. When this occurs, either p' becomes a *candidate* by receiving *msg.candidate* from p; or there exists another *explorer* on L_{-1} that asked to become a *candidate*, it received *msg.candidate*, thus becoming a *candidate*. In all cases, the lemma follows. □

Unique Leader. The sub-algorithm Unique Leader, reported in Figure 11, is executed when an unique leader is elected: it main goal is to let the *leader* collect every particle and eventually form a single line.

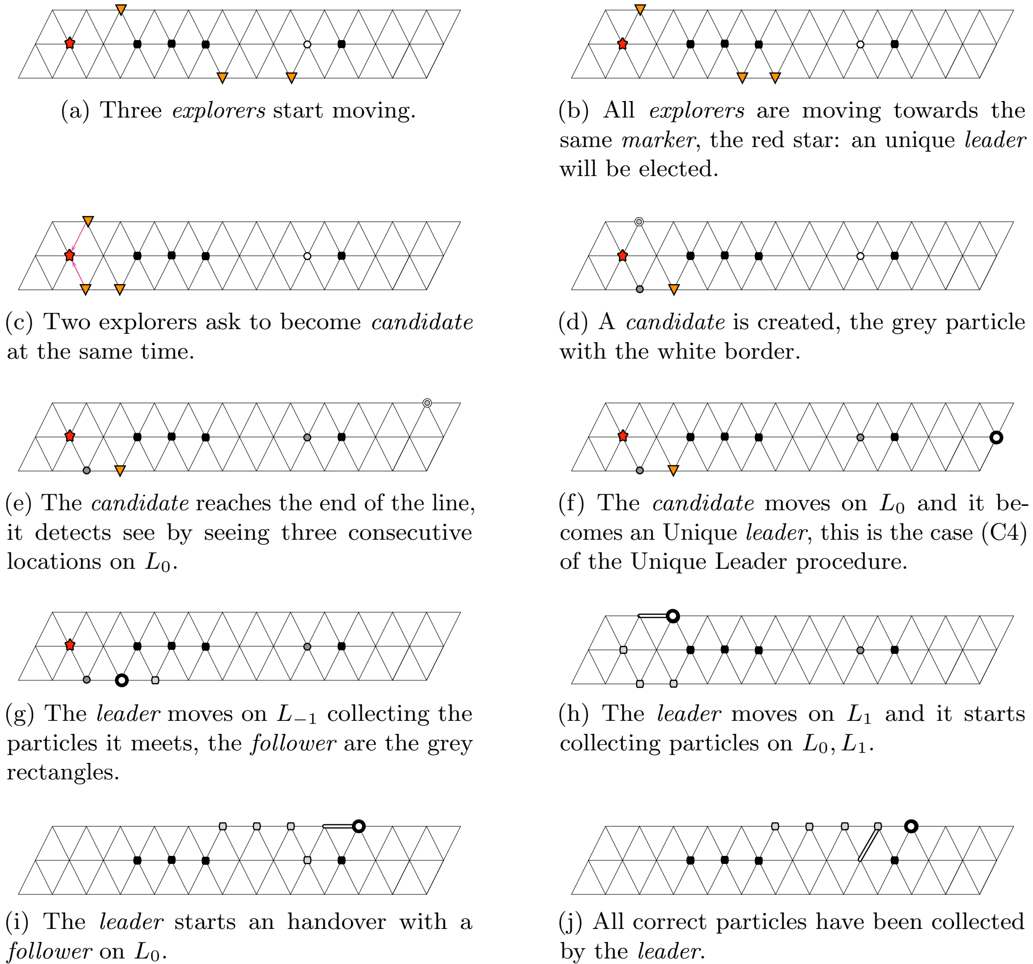
Let us assume that the *leader* is on line L_1: it moves until it sees a *marker*. During its movements, when it finds some particle p on L_1 (either an *explorer* or a *slave*), it forces p to become a *leader*, and it changes its own state to *follower*. By doing so, a line of moving particles is created: this group moves compact using handovers starting from the *leader* until the last *follower* of the line. In particular, the movements are carried out as follows.

Virtual Movement. When a *leader* particle p meets a particle p' that is a *slave* (or an *explorer*), it forces p' to become *leader* and it changes its own state to non-*leader*. We will refer to this protocol as a *move*, and we will say that the *leader* moves in the position of particle p' (even if there has not been any actual movement of p', but just an exchange of roles).

Collection of Particles: When a particle p starts building a line of moving (and correct) particles, we will say that it is collecting the particles. In particular, if p is only interested in collecting particles on its direction of movement, then the collection of a particle p' by p is done by moving virtually to p', while setting the state of p to *follower*. The effect of this is that p will follow p'; also, all other *follower* (behind p) will follow using the handovers. If p wants to collect a particle p' on L_0, then it will send to p' a message $msg.switchtofollower$, and it will wait until p' switches to *follower* state. After the switch, p keeps moving in its direction; also, p' will join the line by an handover by either p or by some other *follower* already in the line that is being built by p.

When a *leader* is a neighbor of a *marker* for the first time, the *marker* becomes the *leader*, and the *leader* becomes a *follower*: at this point, the new appointed *leader* is on L_0. Now, the *leader* starts moving on L_{-1}: during its movements, it makes particles on both L_0 (clearly only the correct ones) and on L_{-1} to follow it. In detail, when the *leader*, while moving on L_{-1}, becomes neighbour of a particle p on L_0, it sends to this particle a message $msg.switchtofollower$; the *leader* does not move until p becomes *follower*. When p becomes *follower* on L_0, it will join the line of the *leader*: in particular, when a tail of an expanded *follower* or *leader* that is neighbour of p contracts, it does an handover with p forcing it to join the line.

When the *leader* reaches the other *marker*, it switches again side, from L_{-1} to line L_1, and it keeps "collecting" the other correct particles following the same strategy. An example run with an Unique Leader is in Figure 10.



(a) Three *explorers* start moving.

(b) All *explorers* are moving towards the same *marker*, the red star: an unique *leader* will be elected.

(c) Two explorers ask to become *candidate* at the same time.

(d) A *candidate* is created, the grey particle with the white border.

(e) The *candidate* reaches the end of the line, it detects see by seeing three consecutive locations on L_0.

(f) The *candidate* moves on L_0 and it becomes an Unique *leader*, this is the case (C4) of the Unique Leader procedure.

(g) The *leader* moves on L_{-1} collecting the particles it meets, the *follower* are the grey rectangles.

(h) The *leader* moves on L_1 and it starts collecting particles on L_0, L_1.

(i) The *leader* starts an handover with a *follower* on L_0.

(j) All correct particles have been collected by the *leader*.

Figure 10: Unique Leader, example run.

```
1: Upon Activation in State follower do:
2:     if expanded(myself) then
3:         if ∃p ∈ tailNeighbour ∧ p = follower then
4:             contractToHeadAndHandover(p)
5:         else
6:             contractToHead()
7:     else if ∃port ∈ Ports|msg.switchtoleader ∈ port then
8:         Set direction and flags from msg.switchtoleader
9:         Set State leader
10:    else
11:        cond_1 := ∃p ∈ Port|msg.probe ∈ p ∧ contracted(myself)
12:        if cond_1 ∧ contracted(opposite(p)) then
13:            send((opposite(p), msg.probe)
14:        else if cond_1 ∧ empty(opposite(p)) then
15:            direction = opposite(p)
16:            Set State probe
17: End
18: Upon Activation in State leader do:
19:    if myself ∈ L_{j∈{−1,1}} ∧ getLocation(direction) ∉ L_j  then
20:        direction = setDirectionToOtherLineExtreme()
21:    else if contracted(myself) ∧ flag.firstsideswitch ∧ ∃p ∈ L_0|p ≠ follower then
22:        send(p, msg.switchtofollower)
23:    else if ∃p ∈ getLocation(direction) then
24:        send(p, msg.switchtoleader)
25:        Set State follower
26:    else if ∃p ∈ L_0 ∧ p = marker then
27:        Set flag.firstsideswitch
28:        direction = p
29:        send(p, msg.switchtoleader)
30:        Set State follower
31:    else if expanded(myself) then
32:        if ∃p ∈ tailNeighbour ∧ p = follower then
33:            contractToHeadAndHandover(p)
34:        else
35:            contractToHead()
36:    else if contracted(myself) ∧ empty(getLocation(direction)) then
37:        expand(getLocation(direction))
38: End
```

Figure 11: Unique Leader

Lemma 6. *At any round r, there is at most one particle p in state* leader*.*

Proof. The proof is by case analysis on how a particle becomes *leader*. Let r be the first round at which there is a *leader*.

- (C1) The *candidate* particle p on side L_1 receives a message *msg.candidate* from a *marker* p' (see Line 41 of Figure 9). By construction, each *marker* may send only a *msg.candidate*. If p is a *candidate* at round r, then it received a *msg.candidate* at round $r' \leq r$. Also, since there are only two *markers*, no other *candidate* becomes a *leader* by receiving a *msg.candidate* (either Line 41 of Figure 9 or Line 9 of Figure 8). Moreover, since p is not anymore a *candidate*, and since there could be at most two *candidate*s, no particle can become *leader* (by Line 35 of Figure 7), otherwise there would be two *candidate*s. Notice that, after round r, the *leader* moves on side L_{-1}, blocking any new *explorer*. Additionally, no *explorer* can be on L_0 since the *candidate* has sent a message *msg.switchtoslave* to every particle on L_0. Therefore, no new *marker* can be created after round r. This ensures that no new *candidate* can be created, for any round $r' \geq r$; therefore, there is no creation of a new *leader* (by Lines 41 of Figure 9,Line 9 of Figure 8, and by Line 50 of Figure 9).

- (C2) A *slave* receives a message *msg.candidate* from a *marker* p', and a message *msg.switchtocandidate* (see Line 9 of Figure 8). The proof follows an argument similar to the one of the previous case.

- (C3) An *explorer* particle p on side L_1 receives two *msg.switchtocandidate* on two opposing ports (Line 35 of Figure 7). Since there could be at most two *candidates*, and a message *msg.switchtocandidate* can only be generated by a *candidate*, and no other *candidate* can be created by the *markers* after round r, it follows that no other particle p becomes *leader* at a round $r' \geq r$ (by Line 35 of Figure 7). Also, all the *explorer* on side L_1 have been turned into *slave* by the two *candidates*. Moreover, no *explorer* can be created after round r: every particle still on L_0 received the message *msg.switchtoslave* by one *candidate*. The proof follows similarly to previous Case (C1).

- (C4) A *candidate* particle p, moving on side L_1 occupies the last of the three empty locations at the end of the line (see Line 50 of Figure 9). In this case, only one *marker* exists, so none of Cases (C1), (C2), and (C3) can be verified at round $\geq r$. Also, the *leader* will move on the other side of the line, blocking any moving *explorer*; thus, since all *explorer* on L_1 and L_0 have been blocked by the *candidate* itself, it is not possible that Case (C4) is verified after round r.

- (C5) Two *candidates* meets (Line 26 of Figure 9). It is easy to see that in this case only one of the *candidates* is elected. Once the *candidate* becomes *leader*, the same scenario of previous Case (C4) occurs, the proof follows similarly.

- (C6) A particle p becomes *leader* after receiving the *msg.switchtoleader*: this message can only be sent by a *leader* at a previous round. Therefore, r cannot be the first round at which there is a *leader*.

Thus, after round r, Cases (C1), (C2), (C3), (C4) and (C5) cannot be verified. The only case left is case (C6): it is easy to see that in this case the number of *leaders* cannot increase. Therefore, the *leader* is at most one. □

Theorem 2. *If there exists a* leader *at round r, then the* LineRecovery *problem is solved.*

Proof. Let r be the first round at which a *leader* is elected. By Lemma 6, the *leader* will be unique. We distinguish the possible cases.

1. At round r, the *leader* particle p is created by Line 50 of Figure 9, and it is moving on side L_1. So p occupied an empty location $l \in L_0$ at round $r - 1$. All *explorers* on side L_1 have been blocked by moving on L_1 as *candidate.* Moreover, after round $r - 1$ no other *explorer* will leave L_0 since all particles on line L_0 have received $msg.switchtoslave$.

   Note that, from round $r - 1$, any *explorer* on side L_{-1} near to location l will switch to state *slave* if it sees p. Since any *explorer* (at distance 1 from l) on side L_{-1} tries to occupy l, no *explorer* on side L_{-1} can move further location l in a round $r' \leq r - 1$ (Line 48 of Figure 9). In the following activations, p moves on side L_{-1}, towards *marker* (p has the flag $flag.firstsideswitch$ set). In particular, the *leader* p collects every non *marker* particle on line L_0, by sending to each of them a $msg.switchtofollower$ message, and by pulling each one of them on its followers line using an handover; this pulling operation is executed either by the *leader* itself, or by one of the *followers* (there will always be at least one *follower* on L_{-1} nearby a *follower* still on L_0). Moreover, the *leader* collects also each particle on L_1 (see Line 23 of Figure 11).

   Note that the *marker* has to exists, because otherwise no *candidate* could have been created. When *leader* reaches the other *marker* at the opposite end of the line, it switches side again (Lines 26 of Figure 11). At this point, the *leader* has collected every particle on L_{-1}. From now on, the *leader* will collect any remaining particle on L_1 and L_0; these particles are not moving, since they are all in state either *slave* or *follower.* Eventually, the *leader* and its followers will form a straight connected line that includes every particle in C.

2. At round r, the *leader* particle p at location l on side L_1 is generated by Line 35 of Figure 7. All particles on L_1 are in state *slave*, and, since all particles on line L_0 have received *msg.switchtoslave*, no other *explorer* will leave L_0. Similarly to the previous case, the *leader* will reach one of the *marker* p', collecting all particles between l and the position of p'. Meanwhile, no *explorer* on side L_{-1} may move further than the *markers* (see Line 27 of Figure 7). Now, the *leader* switches side (Lines 26 of Figure 11), and sets $flag.firstsideswitch$. The last part of the the proof is analogous to the previous case.

3. There are still three possible cases to consider: a *slave* becomes *leader* executing Line 9 of Figure 8; a *candidate* executes Line 41 of Figure 9; and two *candidates* meets (Line 33 of Figure 9). All of them lead to the same scenario, in which the *leader* p on L_1 is nearby a *marker*, all particles on L_1 are *slave*, and all particle on L_0 received $msg.switchtoslave$. The analysis is similar to the previous case.

In all cases, the theorem follows. □

Opposite Sides. This sub-algorithm starts when a *candidate* on L_1 (respectively, L_{-1}) reaches a *marker* p with flag $flag.candidate$ set: the *candidate* realizes that there is another *candidate* moving in the same direction (either clockwise or counter-clockwise) on L_{-1} (respectively, L_1) (see Lines 36-38 of Figure 9). Also, the *candidate* becomes *collector*, switches direction, and moves towards the other *marker* p'; during this movements, the *collector* forces every particle it encounters (also on L_0) to become a *follower.* Moreover, when contracting, it pulls any *follower* near to its tail.

Once the *collector* reaches p', it reverts again direction to reach p (Line 33 of Figure 12); once it reaches p, it switches state to *collector.counting* (Line 36 of Figure 12). Let us assume that the *collector* has recruited at least one particle. The *collector.counting* propagates a message *msg.probe* on its *followers* on L_1 (Line 43 of Figure 13). Once the *msg.probe* reaches a *follower* p'' with only one neighbour, particle p'' switches state to *probe* (Line 14 of Figure 11). The *probe* travels to reach *marker* p', and it temporarily stops if it sees another *probe.* The code for the *probe* is in Figure 12 Lines 2-14. When *marker* p' sees a *probe* and a *collector.counting*, it sends to the *probe* a message $msg.seen$. The *probe* switches state to *slave* when it reads such message.

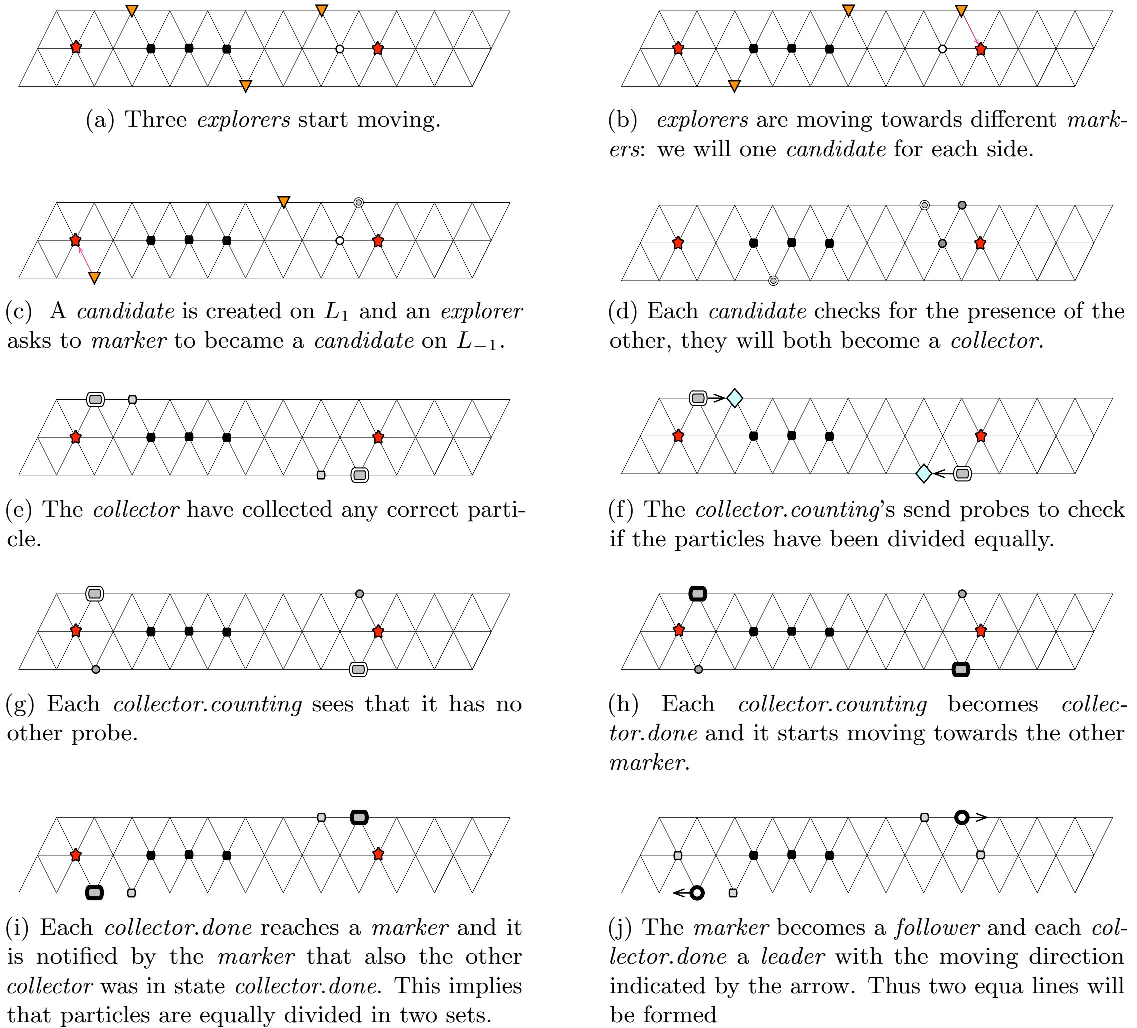
```
1: Upon Activation in State probe do:
2:     if ∃p ∈ L_0 ∧ p = marker then
3:         if ∃p ∈ Port|msg.seen ∈ p then
4:             Set State slave
5:     else if contracted(myself) ∧ empty(getLocation(direction)) then
6:         expand(getLocation(direction))
7:     else if expanded(myself) then
8:         contractToHead()
9:     else if ∃p ∈ getLocation(direction) ∧ contracted(p) ∧ p = slave then
10:        send(p, msg.switchtoprobe)
11:        Set State slave
12:    else if ∃port ∈ Ports|msg.switchtoleader ∈ port then
13:        Set direction and flags from msg.switchtoleader
14:        Set State leader
15: End
16:
17: Upon Activation in State collector do:
18:    if contracted(myself) ∧ ∃p ∈ L_0|p ≠ follower then
19:        send(p, msg.switchtofollower)
20:    else if ∃p ∈ getLocation(direction) ∧ contracted(p) then
21:        send(p, msg.switchtocollector)
22:        Set State follower
23:    else if expanded(myself) then
24:        if ∃p ∈ tailNeighbour ∧ p = follower ∧ p ∈ L_0 then
25:            contractToHeadAndHandover(p)
26:        else if ∃p ∈ tailNeighbour ∧ p = follower then
27:            contractToHeadAndHandover(p)
28:        else
29:            contractToHead()
30:    else
31:        checkAngle := angle(location(p), getLocation(direction)) = 60°
32:        cond_1 = ∃p ∈ L_0 ∧ p = marker ∧ checkAngle
33:        if cond_1 ∧ ¬flag.firstchange then
34:            Set flag.firstchange
35:            direction = oppositeDirection()
36:        else if cond_1 ∧ flag.firstchange then
37:            Set State collector.counting
38:        else if contracted(myself) ∧ empty(getLocation(direction)) then
39:            expand(getLocation(direction))
40: End
```

Figure 12: Collector Algorithm – Part One

```
41: Upon Activation in State collector.counting do:
42:     if ¬flag.firstrun then
43:         send(myself, msg.other)
44:         Set flag.firstrun
45:     else if ∃p ∈ Port|msg.other ∈ p then
46:         if ∃p′ ∈ Neighbours ∧ p′ = follower then
47:             send(p′, msg.probe)
48:         else
49:             ∄p′′|p′′ ∈ Neighbours ∧ p′′ = follower
50:             send(p′′, msg.done)
51:             direction = oppositeDirection()
52:             Set State collector.done
53:     else if ∃p ∈ Port|msg.winner ∈ p then
54:         Set State leader
55: End
56:
57: Upon Activation in State collector.done do:
58:     if ∃p ∈ L0 ∧ (p = marker ∨ p = leader) then
59:         End of Cycle
60:     else if ∃p ∈ Port|msg.even ∈ p then
61:         Set State leader
62:     else if contracted(myself) ∧ empty(getLocation(direction)) then
63:         expand(getLocation(direction))
64:     else if expanded(myself) then
65:         contractToHead()
66:     else if ∃p ∈ getLocation(direction) ∧ contracted(p) ∧ p = slave then
67:         send(p, msg.switchtocollectordone)
68:         Set State slave
69:     else if ∃port ∈ Ports|msg.switchtoleader ∈ port then
70:         Set direction and flags from msg.switchtoleader
71:         Set State leader
72: End
```

Figure 13: Collector Algorithm – Part Two

For each *msg.seen* sent, the *marker* sends a message *msg.other* to a neighbour *collector.counting*. When the *collector.counting* receives a *msg.other* from a *marker*, it propagates a new message *msg.probe* (Line 45 of Figure 13). Eventually, the *collector.counting* receives *msg.other* and it sees that it does not have any *follower* (Line 49 of Figure 13); in this case, it switches state to *collector.done* and it moves until it becomes neighbour of p'. Before moving, it notifies the *marker*, by sending a message *msg.done*. While the *collector.done* moves, it turns to *follower* any *slave* it meets on its way, and it waits if it sees a *probe* (The code fo the *collector.done* is in Lines 58-71 of Figure 13). If the *marker* sees a *collector.done* and a *collector.counting*, that does not have a *msg.other* pending, it sends *msg.winner* to it (Line 27 of Figure 2). Otherwise, if the *marker* sees only a *collector.done* and it was notified by a *msg.done*, then it sends *msg.even* to the *collector.done* and it becomes a *follower* (Line 29 of Figure 2). If a *collector.done* receives a *msg.even*, then it switches state to *leader*. If a *collector.counting* receives a *msg.winner*, then it switches state to *leader*. An example run with *collectors* is in Figure 14.



(a) Three *explorers* start moving.

(b) *explorers* are moving towards different *markers*: we will one *candidate* for each side.

(c) A *candidate* is created on L_1 and an *explorer* asks to *marker* to became a *candidate* on L_{-1}.

(d) Each *candidate* checks for the presence of the other, they will both become a *collector*.

(e) The *collector* have collected any correct particle.

(f) The *collector.counting*'s send probes to check if the particles have been divided equally.

(g) Each *collector.counting* sees that it has no other probe.

(h) Each *collector.counting* becomes *collector.done* and it starts moving towards the other *marker*.

(i) Each *collector.done* reaches a *marker* and it is notified by the *marker* that also the other *collector* was in state *collector.done*. This implies that particles are equally divided in two sets.

(j) The *marker* becomes a *follower* and each *collector.done* a *leader* with the moving direction indicated by the arrow. Thus two equa lines will be formed

Figure 14: Opposite Sides, example run.

Theorem 3. *If there exists a* collector *at round* r*, then the* LINERECOVERY *problem is solved.*

Proof. If a *candidate*, on side L_1, becomes a *collector* when it reaches *marker* p, then there exists a *candidate* on side L_{-1} that will become *collector* when it reaches *marker* p'; let c' be

such *collector*. The *collector* c moves from p to p', and then back to p. Let C' be the set of particle in C excluding the *markers* and the *collectors*. When c reaches p for the second time, it switches state to *collector.counting*; similarly, also c' will eventually reach p', switching state to *collector.counting*. In the following, we will show that, when this occurs, all particles in C' are followers of either c or c', thus forming two segments s and s', respectively. Note that, for each of the *collector.counting* followers, it might be that one of the particles could be a *probe* moving to a *marker*. For now, let us assume that this *probe* is still belonging to its initial segment.

Also, let us assume that there exists a particle p^* that is still in L_0 and that does not belong to c or c'; that is, no one pulled p^*. However, when c is going back from p to p', no particle is on L_1; therefore, there will be a round in which c is expanded and its tail is neighbour of p^*. By construction, when p contracts, it will pull p^* (Line 24 of Figure 12). For particles on L_1 and L_{-1} the theorem easily follows, since they will be collected when the corresponding *collector* is going from the *marker* on which it switched state to the other one.

In the the next step of the sub-algorithm, the sizes of s and s' are compared by using the *probes*. Each segment decreases by 1 by moving a *probe* to the next *marker*, and then it waits for a signal from the probe of the other *collector* to further decrease. We have two possible cases:

1. If $|s| \leq |s'|$, it is easy to see that c first receives a *msg.other*, and then it becomes a *collector.done* moving towards p'. Thus the *marker* p' sees a *collector.counting* c' and a *collector.done* c; therefore, c' becomes a *leader*. From now on, it is easy to see that c' will collect every particle on its way: this case is similar to the case of the Unique Leader sub-algorithm (refer to Therem 2).

2. If $|s| = |s'|$, both *collector.counting* will receive a *msg.other* while there is no *follower* in the neighbourhood. Since the scheduler is semi-synchronous, it might be that, before c' it is activated, c becomes *collector.done* and it reaches the *marker*. In this case, if the *marker* sends the *msg.winner* to c', upon activation c' will find this message and it will become *leader*, and previous case applies. Otherwise, c' sent a *msg.done* to the *marker*; when c reaches p', the *marker*, upon activation, sends *msg.even* to c.

   Since $|s| = |s'|$ and while c is moving it turns every *slave* into a *follower*, we have that half of the processes in C' are followers of c and connected. Also, when the *marker* sends *msg.even* to c it also becomes a *follower*. At this point, c becomes a *leader* and it moves, eventually bringing p' in a straight line. The same will be done by c'. Therefore, in this case we will have two lines of equal size, and the theorem follows.

□

Finally, we have:

Theorem 4. *Starting from any initial configuration, the* LineRecovery *problem is solved.*

Proof. If $f = 0$, the theorem follows by Theorem 1. Let us thus consider the case where $f > 0$. By Lemma 5, a leader will eventually be elected.

By construction, the *candidate* creation always occurs at a marker p, and then it moves towards the other extreme of the line. Let us suppose that an *explorer* sees the extremity of the initial line, and let us consider the first round r' at which this occurs. Since before round r' no *leader* or *collector* can be created, by Lemma 5 three consecutive empty locations correctly mark the end of the line. Therefore, a *candidate* can correctly recognise the end of the line either by the presence of a *marker*, or by three consecutive empty locations.

Three cases can occur:

(1) There is an empty location that is occupied by *candidate*: in this case, Unique Leader sub-algorithm is run, a *leader* is elected, and, by Theorem 2, the theorem follows.

(2) There is *marker* that has not elected a *candidate*, and the leader locks the *marker* by receiving a *msg.switchtocandidate*: in this case, Unique Leader sub-algorithm is run, a *leader* is elected, and, by Theorem 2, the theorem follows.

(3) There is *marker* that has elected a *candidate*: in this case, Opposite Sides is run, and, by Theorem 3, the theorem follows.

Notice that, it might also be possible that a leader reaches the end of the line marked by an location and that another particle occupies it because of the semi-synchronous scheduler. In this case, a *marker* is created, and either Case (2) or (3) above applies. Therefore, in all possible cases, the theorem follows. □

References

[1] N. Agmon and D. Peleg. Fault-tolerant gathering algorithms for autonomous mobile robots. *SIAM Journal on Computing*, 36(1):56–82, 2006.

[2] S. Cannon, J.J. Daymude, D. Randall, and A.W. Richa. A Markov chain algorithm for compression in self-organizing particle systems. In *Proceedings of the 35th ACM Symposium on Principles of Distributed Computing*, 279–288, 2016.

[3] M. Cieliebak, P. Flocchini, G. Prencipe, and N. Santoro. Distributed computing by mobile robots: gathering. *SIAM Journal on Computing*, 41(4):829–879, 2012.

[4] G.A. Di Luna, P. Flocchini, N. Santoro, G. Viglietta, and Y. Yamauchi. Brief Announcement: Shape formation by programmable particles. In *Proceedings of the 31st International Symposium on Distributed Computing*, to appear, 2017.

[5] G.A. Di Luna, P. Flocchini, G. Prencipe, N. Santoro, and G. Viglietta. A Rupestrian algorithm. In *Proceedings of 8th International Conference on Fun with Algorithms*, 14:1-14:20, 2016.

[6] Z. Derakhshandeh, S. Dolev, R. Gmyr, A.W. Richa, C. Scheideler, and T. Strothmann. Brief announcement: Amoebot - a new model for programmable matter. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, 220–222, 2014.

[7] J.J Daymude, Z. Derakhshandeh, R. Gmyr, A. Porter, A.W. Richa, C. Scheideler, and T. Strothmann. On the runtime of universal coating for programmable matter. In *Proceedings of 22nd International Conference on DNA Computing and Molecular Programming*, 148–164, 2016..

[8] J.J. Daymude, R. Gmyr, A.W. Richa, C. Scheideler and T. Strothmann. Improved leader election for self-organizing programmable matter. arXiv:1701.03616, 2017.

[9] Z. Derakhshandeh, R. Gmyr, A.W. Richa, C. Scheideler, and T. Strothmann. Universal shape formation for programmable matter. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, 289–299, 2016.

[10] Z. Derakhshandeh, R. Gmyr, A.W. Richa, C. Scheideler, and T. Strothmann. An algorithmic framework for shape formation problems in self-organizing particle systems. In *Proceedings of the 2nd International Conference on Nanoscale Computing and Communication*, 21:1–21:2, 2015.

[11] Z. Derakhshandeh, R. Gmyr, T. Strothmann, R.A. Bazzi, A.W. Richa, and C. Scheideler. Leader election and shape formation with self-organizing programmable matter. In *Proceedings of the 21st International Conference on DNA Computing and Molecular Programming*, 117–132, 2015.

[12] Z. Derakhshandeh, R. Gmyr, A.W. Richa, C. Scheideler, and T. Strothmann. Universal coating for programmable matter. *Theoretical Computer Science*, to appear, 2017.

[13] S. Dolev, S. Frenkel, M. Rosenbli, P. Narayanan, and K.M. Venkateswarlu. In-vivo energy harvesting nano robots. In *the 3rd IEEE International Conference on the Science of Electrical Engineering*, 1–5, 2016.

[14] Y. Emek, T. Langner, J. Uitto, R. Wattenhofer. Solving the ANTS problem with asynchronous finite state machines. In *Proceedings of the 41st International Colloquium on Automata, Languages, and Programming*, 471–482, 2014.

[15] P. Flocchini, G. Prencipe, and N. Santoro. *Distributed computing by oblivious mobile robots.* Morgan & Claypool, 2012.

[16] P. Flocchini, G. Prencipe, N. Santoro, and P. Widmayer. Arbitrary pattern formation by asynchronous, anonymous, oblivious robots. *Theoretical Computer Science*, 407(1):412–447, 2008.

[17] N. Fujinaga, Y. Yamauchi, S. Kijima, and M. Yamashita. Pattern formation by oblivious asynchronous mobile robots. *SIAM Journal of Computing*, 44(3):740–785, 2016.

[18] M.J. Patitz. An introduction to tile-based self-assembly and a survey of recent results. *Natural Computing*, 13(2):195–224, 2014.

[19] I. Suzuki and M. Yamashita. Distributed anonymous mobile robots: formation of geometric patterns. *SIAM Journal on Computing*, 28(4):1347–1363, 1999.

[20] T. Toffoli and N. Margolus. Programmable matter: concepts and realization. *Physica D: Nonlinear Phenomena*, 47(1):263–272, 1991.

[21] J.E. Walter, J.L. Welch, and N.M. Amato. Distributed reconfiguration of metamorphic robot chains. *Distributed Computing*, 17(2):171–189, 2004.

[22] Y. Yamauchi, T. Uehara, S. Kijima, and M. Yamashita. Plane formation by synchronous mobile robots in the three dimensional euclidean space. *Journal of the ACM*, 64(3):1–43, 2017.