A Distributed Algorithm for Gathering Many Fat Mobile Robots in the Plane[*]

Chrysovalandis Agathangelou Chryssis Georgiou Marios Mavronicolas
Department of Computer Science
University of Cyprus
CY-1678 Nicosia, Cyprus
{cs06ac2, chryssis, mavronic}@cs.ucy.ac.cy

ABSTRACT

We revisit the problem of gathering autonomous robots in the plane. In particular, we consider non-transparent unit-disc robots (i.e., *fat*) in an asynchronous setting with vision as the only means of coordination and robots only make local decisions. We use a state-machine representation to formulate the gathering problem and develop a distributed algorithm that solves the problem for any number of fat robots. The main idea behind the algorithm is to enforce the robots to reach a configuration in which all the following hold:

(*i*) The robots' centers form a convex hull in which all robots are on the convex hull's boundary;

(*ii*) Each robot can see all other robots;

(*iii*) The configuration is *connected*: every robot touches another robot and all robots form together a connected formation.

We show that starting from any initial configuration, the fat robots eventually reach such a configuration and terminate yielding a solution to the gathering problem.

Categories and Subject Descriptors

I.2.9 [**Robotics**]: Autonomous vehicles; F.2.2 [**Nonnumerical Algorithms and Problems**]: Geometric problems and computations; C.2.4 [**Distributed Systems**]: Distributed applications

General Terms

Algorithms, Theory

Keywords

Gathering, Fat robots, Asynchrony, State-machines

1. INTRODUCTION

Motivation and Prior Work. There is an increasing number of applications that benefit from having a team of autonomous robots to cooperate and complete various tasks in a self-organizing manner. Such application tasks may require, for example, that robots work in dangerous and harsh environments (e.g., for space, underwater or military purposes) or achieve high accuracy or speed (e.g., in nanotechnology, scientific computing). It is usually desirable for the robots to be as simple as possible and have limited computing power, in order to be able to produce them fast in large numbers and cheap.

A fundamental problem that has drawn much attention recently is *gathering* [2, 3, 5, 12, 14, 15], where a team of autonomous mobile robots must gather to a certain point or region or form a certain formation (e.g., geometric shape) in the plane. The problem has been studied under various modeling assumptions; for example, *asynchronous*, *semi-synchronous* and *synchronous* settings have been considered. Robots may have a common coordination system, or have common sense of direction and use compasses to navigate in the plane; they may have stable memory or be history-oblivious. In all considered models, robots are equipped with a vision device (e.g., a camera) and their range of visibility is either limited or unlimited. Robots operate under the *Look-Compute-Move* cycle. Within a cycle, a robot takes a snapshot of the plane (*Look*), performs some local computation (*Compute*), and possibly decides to move to some other point in the plane (*Move*). We refer the reader to surveys [5, 15] and the recent monograph [11] for a more comprehensive exposition of works on the gathering problem.

Up until the work of Czyzowicz *et al.* [9], the gathering problem was considered only under the assumption that each robot is a point on the plane and transparent: a robot can "see" through another robot. This assumption does not reflect reality as real robots are not points but have a physical extent. This means that robots may collide with each other. Furthermore, robots are not transparent: they may block the view of other robots. To depart from such assumptions, Czyzowicz *et al.* [9] initiated the study of the gathering problem with *fat* robots: non-transparent, unit discs. As fat robots cannot occupy the same space on the plane, the gathering problem no longer requires robots to gather at the same point. Instead, per [9], *gathering fat robots means forming a configuration for which the union of all discs is connected.*

In the model considered in [9], robots operate in *Look-Compute-Move* cycles, they are identical, anonymous, history-oblivious, non-transparent, and fat. They do not share a common coordination system and vision is the only mean of coordination; robots have unlimited visibility unless their view is obstructed by another robot. An asynchronous setting is considered, where an adaptive adversary can stop a robot for finite time, control the "speed" of a robot, or cause robots moving into intersecting trajectories to collide. Under this model, the authors present solutions to the gathering problem for *three* and *four* robots. The proposed solutions rely on exhaustive consideration of all possible classes of configurations; a different gathering strategy corresponds to each possible case. As the num-

[*]This work is supported by research funds from the University of Cyprus.

ber of cases may grow exponentially with the number of robots, this approach fails to generalize. The authors of [9] left open the question of whether it is possible to solve gathering for any collection of $n \geq 5$ fat robots.

Our Contribution. We provide a positive answer to the above question. In particular, we consider the model of [9] with the additional assumption of *chirality* [11]: robots agree on the orientation of the axes of their local coordination system. We present a distributed algorithm for the gathering problem for *any* number n of fat robots.

The key feature of our algorithm is to bring the robots into a configuration of *full visibility* where all robots can see all other robots. Given the power of the adversary and the fact that robots are non-transparent, this task becomes challenging; that was in fact, the main challenge of our work. The idea for settling the challenge is for the robots to aim in forming a *convex hull* in which all robots will be on the convex hull's boundary. During the computation, robots on the boundary do not move; robots inside the convex hull try to move towards the boundary. However, if robots that are on the hull's boundary realize that they obstruct other robots that are also on the boundary from seeing each other, then they move away from the convex hull so that they no longer cause any obstruction. If a robot on the boundary realizes that there is no "enough space" for robots inside the convex hull to be placed on the boundary, then it moves to a direction away from the convex hull to make space. Asynchrony only makes things harder as robots may have very different local views of the system. We show that eventually the convex hull will "expand" so that all robots will be on the boundary of the convex hull and *no three robots are on the same line*[1]; this leads to a configuration that all robots have full visibility. This is the first conceptual phase of the algorithm.

In the second conceptual phase, once all robots have full visibility and are aware of this, robots start to converge in a way that full visibility is not lost. To do so, robots exploit their knowledge of n and the common unit of distance (since all robots are unit-discs, this gives them "for free" a common measure of distance [9]). We show that eventually all robots form a connected configuration and terminate yielding a solution to the gathering problem. Note that robots must have full visibility to be aware that gathering is accomplished [9].

The key to successfully proving the correctness of the algorithm is the formulation of the model, the problem and the algorithm with a state-machine representation. This enables employing typical techniques for proving safety and liveness properties and argue on the state transitions of the robots, which, against asynchrony, becomes a very challenging task.

Other Works Considering Fat Robots. After the work in [9] some attempts were made to solving the gathering problem with $n \geq 5$ fat robots in different models [6, 7, 8, 10]. In [7], it is assumed that the fat robots are transparent. This assumption makes the problem significantly easier as robots have full visibility at all times. As discussed above, having the robots reach a configuration with full visibility was the main challenge in our work. In [8], fat robots are non-transparent and have limited visibility, but a synchronous setting is considered. Furthermore, the gathering point is predefined and given as an input to the robots; the goal is for the robots to gather in an area as close as possible to this point. Two versions of the problem are studied: in continuous space and time, and in discrete space (essentially $\mathbb{Z}^2$) and time. In the continuous case, a randomized solution is proposed; in the discrete case the proposed solutions require additional modeling assumptions such as unique robot ids, or direct communication between robots. The work in [10] also considers fat robots with limited visibility, but in an asynchronous setting. In contrast with the model we consider, robots have a common coordination system: they agree both on a common origin and axes (called *Consistent Compass* in [11]). The objective of the robots is to gather to a circle with a center given as an input along with the radius of the circle. The common coordination system and the predefined knowledge of the circle to be formed enables the use of geometric techniques that cannot be used in our model. In [6], they consider fat robots with limited visibility and without a common coordination system, but in a synchronous setting. Furthermore the correctness of their proposed algorithm is not proven but rather demonstrated via simulations.

2. MODEL AND DEFINITIONS

Our model of computation is a formalization of the one presented in [9] with the additional assumption of *chirality*; the formalism follows the one from [4].

Robots. We assume n asynchronous, fault-free robots that can move along straight lines on the (infinite) plane. The robots are *fat* [9]: they are closed unit discs. They are identical and anonymous (i.e., they are indistinguishable). They do not have access to any global coordination system, but we assume *chirality* [11]: the robots agree on the orientation of the axes[2]. Robots are equipped with a 360-degree-angle vision device (e.g., camera) that enables the robots to take snapshots of the plane. The vision device has unlimited range and captures any point of the plane provided there is no obstacle (e.g., another robot). We assume that robots know n.

Geometric configuration. A *geometric configuration* is a vector $\mathcal{G} = (c_1, c_2, \ldots, c_n)$ where each c_i represents the center of the position of robot r_i on the plane. So, a configuration can be viewed as a snapshot of the robots on the plane. Note that the fact that robots are fat prohibits the formation of a configuration in which any two robots share more than a point in the plane. (Two robots share a point if the discs representing them touch each other.)

We say that a geometric configuration $\mathcal{G}$ is *connected* if between any two points of any two robots there is a polygonal line each of whose points belongs to some robot. Informally, a configuration is connected if every robot touches another robot and all robots form together a connected formation.

Visibility and fully visible configuration. We say that point p in the plane is *visible* by a robot r_i (or equivalently, r_i can see p) if there is a point p_i in the circle bounding robot r_i such that the straight segment (p_i, p) does not contain any point of any other robot. So, a robot r_i can see another robot r_j if there is at least one point on the bounding circle of r_j that is visible by r_i. Given a geometric configuration $\mathcal{G}$, robot r_i has *full visibility* in $\mathcal{G}$ if r_i can see all other $n-1$ robots. If *all* robots have full visibility in $\mathcal{G}$, then configuration $\mathcal{G}$ is *fully visible*.

Robots' states. Each robot r_i is modeled as a (possibly infinite) state machine with state set S_i; i is the index of robot r_i (used only for reference purposes). Each set S_i contains five states: **Wait**, **Look**, **Compute**, **Move**, and **Terminate**. Initially each robot is in state **Wait**. State **Terminate** is a terminal state: once a robot reaches this state it does not take any further steps. We now describe each state:

- In state **Wait**, robot r_i is idling. In addition, the robot has

[1]Note that there are cases where having all robots on the boundary does not necessarily imply that all robots can see each other.

[2]Note that this is a weaker assumption than having a common coordinate system or having a consistent compass [11, Section 2.7].

no memory of the steps occurred prior entering this state, i.e., robots are *history-oblivious*.

- In state **Look**, robot r_i takes a snapshot of the plane and identifies all robots that are visible to it. We denote by V_i the set of the centers of the robots that are visible to robot r_i when it takes a snapshot in configuration $\mathcal{G}$. So, $V_i \subseteq \mathcal{G}$ is the *local view* of robot r_i in configuration $\mathcal{G}$. This view does not change in subsequent configurations unless the robot takes a new snapshot. In a nutshell, in state **Look**, the robot takes as an input a configuration $\mathcal{G}$ and outputs the local view $V_i \subseteq \mathcal{G}$.
- In state **Compute**, robot r_i runs a local algorithm A_i that takes as an input the local view V_i (i.e., the output of the previous state **Look**) and outputs a point p in the plane. This point is specified from V_i, hence we will write $p = A_i(V_i)$. If A_i returns the special output $\perp$, then the robot's state changes into state **Terminate**. Otherwise it changes into state **Move**; intuitively, in this case p is the point that the center of the robot will move to. Note that it is possible for $p = c_i$ — the robot might decide not to move.
- In state **Move**, robot r_i, starting from its current position, called *start* point, moves on a straight line towards point $A_i(V_i)$ (as calculated in state **Compute**). We call $A_i(V_i)$ the *target* point of r_i. If during its motion the robot touches another robot (i.e., the circles representing these robots become tangent), then it stops and the robot's state changes into state **Wait**. As we discuss next, the adversary may also stop a robot at any point before reaching its target point. Again, in this case, the robot's state changes into state **Wait**. If the robot finds no obstacles or it is not stopped by the adversary, then it eventually reaches its target point (its center is placed on $A_i(V_i)$) and its state changes into state **Wait**.

State configuration. A *state configuration* is a vector $\mathcal{S} = (s_1, s_2, \ldots, s_n)$ where each s_i represents the state of robot r_i. An *initial* state configuration is a configuration $\mathcal{S}$ in which each s_i is an initial state of robot r_i (that is, $\forall i \in [1, n],\ s_i = \textbf{Wait}$). Similarly, a *terminal* state configuration is a configuration $\mathcal{S}$ in which each s_i is a terminal state of robot r_i (that is, $\forall i \in [1, n],\ s_i = \textbf{Terminate}$).

Robot configuration. A *robot configuration* is a vector $\mathcal{R} = (\langle s_1, c_1\rangle, \ldots, \langle s_n, c_n\rangle)$ where each pair $\langle s_i, c_i\rangle$ represents the state of robot r_i and the position of its center on the plane. Informally, a robot configuration is the combination of a geometric configuration with the corresponding state configuration.

Adversary and events. We model asynchrony as a sequence of events caused by an online and omniscient adversary. The adversary can control the speeds of the robots, it can stop moving robots, and it may cause moving robots to collide, provided that their trajectories have an intersection point. Specifically, we consider the following events (state transitions — see Figure 1 for a pictorial):

Look(r_i): Causes robot r_i that is in state **Wait** to get into state **Look**.

Compute(r_i): Causes robot r_i that is in state **Look** to get into state **Compute**.

Done(r_i): Causes robot r_i that is in state **Compute** and its local algorithm A_i has returned the special point $\perp$, to get into the terminating state **Terminate**.

Move(r_i): Causes robot r_i that is in state **Compute** and its local algorithm A_i has returned a point other than $\perp$, to get into state **Move**.

Stop(r_i): Causes r_i that is in state **Move** to get into state **Wait**. Robot r_i is stopped at some point in the straight segment between its start point and its target point $A_i(V_i)$ (under a constraint discussed next).

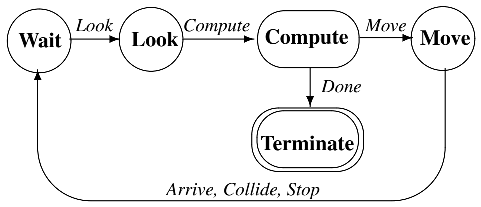

Figure 1: A cycle of the state transitions of robot r_i.

Collide(R): Causes a subset of the robots R that are in state **Move** and their trajectories have an intersecting point to collide (i.e., their circles become tangent). Note that $2 \le |R| \le n$ (two or more robots could collide between them but only one collusion occurs per a *Collide* event). Also, other robots that are in state **Move** could be stopped (without colluding with other robots). Then all affected robots enter in state **Wait**.

Arrive(r_i): Causes robot r_i that is in state **Move** to arrive at its target point and change its state into **Wait**.

Note that events *Look*(r_i), *Move*(r_i), *Stop*(r_i) and *Arrive*(r_i) may also cause robots other than r_i that are in state **Move** to remain in that state, but on a different position (along their trajectories, and closer to their destination).

Execution. A distributed algorithm is a collection of local algorithms, one per robot. An *execution fragment* α of a distributed algorithm is a (finite or infinite) alternating sequence $\mathcal{R}_0, e_1, \mathcal{R}_1, e_2, \ldots$, where each $\mathcal{R}_k$ is a robot configuration and each e_k is an event. If α is finite, then it ends in a configuration. An *execution* of an algorithm is an execution fragment where $\mathcal{R}_0$ is an initial configuration.

Liveness conditions. We impose the following liveness conditions (i.e., restrictions on the adversary):

1. In an infinite execution, each robot is allowed to take infinitely many steps.
2. During a **Move** event, each robot traverses at least a distance $\delta > 0$ unless its target point is closer than δ. Formally, each robot r_i traverses at least a distance $\min\{dist_i(start, target), \delta\}$, where $dist_i(start, target)$ is the distance between the start and target points of robot r_i. Parameter δ is not known to the robots.

Gathering problem. We now state the problem studied in this work:

DEFINITION 1 (GATHERING). *In any execution, there is a connected, fully visible, terminal robot configuration.*

3. GEOMETRIC FUNCTIONS

We list a collection of functions that perform geometric calculations. These functions are used by the robots' local algorithm as shown in Section 4. Here we present the problems these functions solve. Pseudocodes, their proofs of correctness and more details and insights can be found in the full paper [1].

Function `On-Convex-Hull`**:** We denote by $CH(c_1, c_2, \ldots, c_m)$ the *convex hull* formed by points $c_1, c_2, \ldots, c_m$, and by $\vartheta CH(c_1, c_2, \ldots, c_m) \subseteq \{c_1, c2, \ldots, c_m\}$ the set of points that are *on the boundary* of the convex hull. Then, function `On-Convex-Hull` gets as ***input*** a set of m points $c_1, c_2, \ldots, c_m$ and another point c and ***outputs*** *(i)* $CH(c_1, c_2, \ldots, c_m)$ and *(ii)* whether $c \in \vartheta CH(c_1, c_2, \ldots, c_m)$ or not.

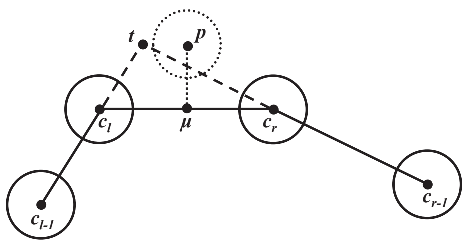

Figure 2: An example where point p is not valid and hence it will not be returned by Function `Find-Points`**.**

Function `Move-to-Point`**:** This function gets as ***input*** two points c_1 and c_2 and a positive integer m and ***outputs*** a point μ defined as follows: Consider the straight segment $\overline{c_1c_2}$ and let $\overline{pc_2}$ be the straight segment which is vertical to $\overline{c_1c_2}$ with p on the perimeter of the unit disc with center c_2, and with direction towards inside of the convex hull. Next consider the point c on segment $\overline{pc_2}$ which has distance $\frac{1}{2m} - \varepsilon$ from c_2. Then point μ is the intersection of the straight segment $\overline{c_1c}$ and the perimeter of the unit disc with center c_2.

Function `Find-Points`**:** This function is significant for our algorithm. It gets as ***input*** a convex hull of n points, where $\vartheta CH(c_1, c_2, \ldots, c_m),\ m \leq n$, and ***outputs*** a set of $k < m$ points $p_1, \ldots, p_k$ so that a unit disc with center $p_i,\ 1 \leq i \leq k$, can be placed on the convex hull *without* causing the convex hull to change. (It is possible that $k = 0$.) The following claim is essential for our solution to gathering.

LEMMA 3.1. *Given a convex hull, let c_l and c_r be the centers of any two unit discs that are adjacent on the hull's boundary. Then there is a minimum distance between c_l and c_r for which Function* `Find-Points` *returns a point between them. We refer to this distance as **safe**.*

PROOF. Consider that given a number of points, a convex hull always exists. Consider four neighbor points on a convex hull, as shown on Figure 2, without loss of generality. In order for a unit disc with center p to be on the convex hull and not cause the current convex hull to change, the distance between μ , the middle point of $\overline{c_lc_r}$ and p must be at least $\frac{1}{n}$. Note that p is outside of the current convex hull. Additionally, consider q the point on the line segment $\overline{pc_{r+1}}$, where a vertical line to c_r starts from line segment $\overline{pc_{r+1}}$ with direction to the inside of the convex hull. Then $d(q, c_r)$ must be equal with at least $\frac{1}{n}$, where r is the point that $\overline{pc_{r+1}}$ is tangent with $\overline{\mu c_r}$. Angle $\widehat{pr\mu}$ is equal with angle $\widehat{c_rrq}$.

We need to calculate the distance between c_l and c_r which will give us the safe distance. The distance between c_l to μ must be equal with the distance between c_r to μ. We need to calculate both $d(\mu, c_r)$ and $d(\mu, c_l)$, find the biggest and double it, in order to find the safe distance. First we must calculate the necessary distance between μ and c_r. Observe that $d(\mu, c_r) = d(\mu, r) + d(r, c_r)$. We now have that $tan(\widehat{pr\mu}) = \frac{\frac{1}{n}}{d(\mu,r)}$, hence $d(\mu, r) = \frac{1}{n \cdot tan(\widehat{pr\mu})}$. We now calculate $d(r, c_r)$; we have that $sin(\widehat{c_rrq}) = \frac{\frac{1}{n}}{d(r,c_r)}$, hence $d(r, c_r) = \frac{1}{n \cdot sin((\widehat{c_rrq}))} = \frac{1}{n \cdot sin((\widehat{pr\mu}))}$. Finally, it follows that $d(\mu, c_r) = \frac{1}{n \cdot tan(\widehat{pr\mu})} + \frac{1}{n \cdot sin((\widehat{pr\mu}))}$.

This is the minimum distance that $\overline{\mu c_r}$ must be. We do the same as above with $\overline{\mu c_l}$ and choose the biggest distance between the two, double it and set it as the safe distance. ■

Function `Connected-Components`**:** Consider a set of m unit discs on the plane. A *connected component* of this set is a collection of connected unit discs; in a connected component there can be up to two empty spaces of distance less or equal to $1/2m$ among the unit discs. Note that a given set of unit discs may contain many connected components and only one in the case that all unit discs are connected. Then, Function `Connected-Components` gets as ***input*** a set of m points $c_1, c_2, \ldots, c_m$ and an additional point c and ***outputs*** a set of pairs of the form $\langle(c_l, c_r),\ k\rangle$. Each pair (c_l, c_r) represents a connected component of unit discs, where c_l is the center of the leftmost (counter clock-wise) unit disc and c_r the center of the rightmost (clock-wise) unit disc in the component; k is the number of unit discs contained in this component (including those with centers c_l and c_r).

As this function plays an important role in forming the final, single connected component, we provide some additional insight: Function `Connected-Components` is called by a robot r with center c. The m points are the centers of the robots that robot r can see (its local view) in the current configuration. As we will see later, this function is called when the robot can see all other robots, i.e., $m = n$. The robot wishes to find the connected components formed in the current configuration. Intuitively, we can include two spaces of length $1/2n$ in a configuration, since if all the robots can see each other, then the robots can move taking steps of length $1/2n$ until they meet (see also the proof of Lemma 5.4).

Function `In-Straight-Line-2`**:** This function gets as ***input*** three points c_l, c_m and c_r and ***outputs*** YES, if the three points are on the same line, and NO otherwise.

The next three functions make use of Function `Connected-Components`.

Function `How-Much-Distance`**:** This function gets as **input** a set of m points $c_1, c_2, \ldots, c_m$ and an additional point c and **outputs** 1,2 or 3. Consider the connected components formed by the unit discs with centers $c_1, c_2, \ldots, c_m$. If the unit disc with center c is the rightmost (the straight direction is considered to be the inside of the convex hull) element of the component that has the smallest (space-wise) distance between the components, then the answer is 1. If all components have the same distance, then the answer is 2. Otherwise the answer is 3.

Function `In-Largest-Component`**:** This function gets as **input** a set of m points $c_1, c_2, \ldots, c_m$ and an additional point c and **outputs** 1,2 or 3. Consider the connected components formed by the unit discs with centers $c_1, c_2, \ldots, c_m$. If the unit disc with center c belongs in the largest component (wrt the number of discs), then the answer is 1; if all the components are larger than the one it belongs, then the answer is 2. Otherwise the answer is 3.

Function `In-Smallest-Component`**:** This function gets as **input** a set of m points $c_1, c_2, \ldots, c_m$ and an additional point c and **outputs** 1,2 or 3. Consider the connected components formed by the unit discs with centers $c_1, c_2, \ldots, c_m$. If the unit disc with center c belongs in the smallest component (wrt the number of discs), then the answer is 1; if all the components are smaller than the one it belongs, then the answer is 2. Otherwise the answer is 3.

4. LOCAL ALGORITHM FOR COMPUTE

We present the algorithm that each robot runs locally while in state **Compute**. It takes as input the view of the robot (obtained in state **Look**) and calculates the position the robot should move next (in state **Move**). In Section 4.1 we overview in verbose the states of the algorithm; in Section 4.2 we list the procedures that implement the transitions from one state to another. Full details are given in the full paper [1].

4.1 States of the Algorithm

Once a robot r_i is in state **Compute** it starts executing the local algorithm A_i. Recall that V_i denotes robot's r_i local view: the set of robots that are visible to r_i upon entering state **Compute**. The algorithm consists of 17 states; we refer to them using the notation **Compute.**$\langle$**algorithm-state-name**$\rangle$. Figure 3 describes these states and Figure 4 depicts all possible states and transitions of the algorithm run by robot r_i.

4.2 Description of the Algorithm

The algorithm consists of 17 procedures, each treating a corresponding algorithmic state. In particular, once the algorithm is in a state **Compute.**$\langle$**algorithm-state-name**$\rangle$ it runs the corresponding procedure `algorithm-state-name` that either implements a state transition or outputs a point the robot should move to (in the next state **Move**); it implements a state transition if it is in a non-terminal state and outputs a point otherwise. In a nutshell, the algorithm consists of conditional expressions:

LOCAL ALGORITHM

if state= **Compute.**$\langle$**algorithm-state-name**$\rangle$ **then** run procedure `algorithm-state-name`.

We now overview the procedures and their properties. The procedures are given with respect to a robot r_i and its center c_i. The robot takes action based on its local view V_i, which might be different from other robots' views. But as we show in Section 5, the local decisions made by each robot are designed in such a way that robots coordinate correctly in the face of asynchrony and eventually reach a solution to the gathering problem. Recall that $\vartheta CH(V_i)$ is the set of points that are on the boundary of the convex hull formed by the points in V_i. Detailed pseudocodes and omitted proofs can be found in the full paper [1].

1. Procedure `Start`**:** It calls Function `On-Convex-Hull` and if $c_i \in \vartheta CH(V_i)$, it changes the state from **Compute.Start** to **Compute.OnConvexHull**; otherwise it changes the state to **Compute.NotOnConvexHull**. The correctness of Function `On-Convex-Hull` (e.g., Graham's algorithm [13]) yields the following:

LEMMA 4.1. `Start`(**Compute.**$\langle$**Start**$\rangle$) = **Compute.**$\langle$**OnConvexHull**$\rangle$ *iff* $c_i \in \vartheta CH(V_i)$.

2. Procedure `OnConvexHull`**:** It changes the state from **Compute.OnConvexHull** to **Compute.AllOnConvexHull** if $|\vartheta CH(V_i)| = n$ and no three robots are on the same line; otherwise it changes the state to **Compute.NotAllOnConvexHull**. It checks whether three robots are on the same line, using Function `In-Straight-Line-2`. Then:

LEMMA 4.2. `OnConvexHull`(**Compute.**$\langle$**OnConvexHull**$\rangle$) = **Compute.**$\langle$**AllOnConvexHull**$\rangle$ *iff* $|V_i| = n$ *and* $|\vartheta CH(V_i)| = n$ *and all robots have full visibility in* V_i.

3. Procedure `AllOnConvexHulll`**:** It changes the state from **Compute.AllOnConvexHull** to **Compute.Connected** if all robots form a *connected configuration*; otherwise it changes state to **Compute.NotConnected**. Then:

LEMMA 4.3. `AllOnConvexHull`(**Compute.**$\langle$**AllOnConvexHull**$\rangle$) = **Compute.**$\langle$**Connected**$\rangle$ *iff* V_i *is a connected configuration.*

4. Procedure `Connected`**:** It returns the special output $\perp$, which leads to the termination of the algorithm for robot r_i (it enters state **Terminate** in which r_i does not perform any further steps).

5. Procedure `NotConnected`**:** Its purpose is to eventually cause all robots to form a *connected* configuration. This procedure gives first priority to components with the smallest size, and then to components that the distance to their neighboring component on the right is the smallest distance between any two components. The rightmost robot of the component with the highest priority moves to the left of its right neighbor component using Function `Move-To-Point`. If all components have equal priority (i.e., all components have the same size and the distance between any two components is the same), then, using Function `Connected-Components`, the robots start to converge. Of course, the procedure is run by each robot locally and individually, but as it is shown in Section 5, global convergence is eventually reached. Robots can start moving only if for any three neighboring robots of the component, say r_l, r_m and r_r, the vertical distance from line $\overline{r_l r_r}$ to r_m is equal or greater than $\frac{1}{n}$. Then:

LEMMA 4.4. *The point returned by* `NotConnected`(**Compute.**$\langle$**NotConnected**$\rangle$) *keeps* V_i *as a fully visible configuration and* $|\vartheta CH(V_i)| = n$.

6. Procedure `NotAllOnConvexHull`**:** It changes the state from **Compute.NotAllOnConvexHull** to **Compute.OnStraightLine** if r_i is on the same line with at least two other robots that are also on the boundary of the convex hull; otherwise it changes the state to **Compute.NotOnStraightLine**. Then:

LEMMA 4.5. `NotAllOnConvexHull`(**Compute.**$\langle$**NotAllOnConvexHull**$\rangle$) = **Compute.**$\langle$**OnStraightLine**$\rangle$ *iff* r_i *is on the same line with any two other robots that are also on the boundary of the convex hull.*

7. Procedure `NotOnStraightLine`**:** It changes the state from **Compute.NotOnStraightLine** to **Compute.SpaceForMore** if there is enough space for at least one robot on the boundary of the convex hull; otherwise it changes the state to **Compute.NoSpaceForMore**. Then:

LEMMA 4.6. `NotOnStraightLine`(**Compute.**$\langle$**NotOnStraightLine**$\rangle$) = **Compute.**$\langle$**SpaceForMore**$\rangle$ iff $|\vartheta CH(V_i)| = n$ *or there is enough space for at least one robot between any two adjacent robots that are on the boundary of the convex hull.*

8. Procedure `SpaceForMore`**:** It returns a point p outside the convex hull if r_i is touching with another robot on the convex hull's boundary that is not adjacent to r_i on the boundary. Otherwise, it returns c_i and the robot does not move.

LEMMA 4.7. `SpaceForMore`(**Compute.**$\langle$**SpaceForMore**$\rangle$) = c_i *iff* r_i *is not tangent with any robot* r_j, $r_j \in \vartheta CH(V_i)$ *that they are not adjacent on* $\vartheta CH(V_i)$, **else** `SpaceForMore`(**Compute.**$\langle$**SpaceForMore**$\rangle$) = p, *were* p *is at distance* $\frac{1}{2n} - \varepsilon$ *away from* $\vartheta CH(V_i)$.

The reason that p is outside of the convex hull by a distance $\frac{1}{2n} - \varepsilon$ is because if two robots are not adjacent on the boundary and are touching, then it would be possible to obstruct other robots from seeing each other.

9. Procedure `NoSpaceForMore`**:** It returns a point p with direction away from the convex hull such that:

LEMMA 4.8. `NoSpaceForMore`(**Compute.**$\langle$**NoSpaceForMore**$\rangle$) = p, *were* p *is at distance* $\frac{1}{2n} - \varepsilon$ *away from* $\vartheta CH(V_i)$.

10. Procedure `OnStraightLine`**:** It changes the state from **Compute.OnStraightLine** to **Compute.SeeOneRobot** if r_i is not in straight line with its left and right neighbors on the boundary of the convex hull; otherwise it changes the state to **Compute.SeeTwoRobot**.

1. **Compute.Start**:
   - The initial state of the algorithm run by robot r_i.
2. **Compute.OnConvexHull**:
   - Robot r_i is on the boundary of the convex hull formed by the robots in V_i: $c_i \in \vartheta CH(V_i)$.
3. **Compute.AllOnConvexHull**:
   - $c_i \in \vartheta CH(V_i)$
   - Robot r_i has full visibility.
   - $\forall k \in [1, n], k \neq i,\ c_k \in \vartheta CH(V_i)$ and V_i is a fully visible configuration.
4. **Compute.Connected**:
   - Same conditions as in state 3.
   - V_i is a connected component.
5. **Compute.NotConnected**:
   - Same conditions as in state 3.
   - V_i is not a connected component.
6. **Compute.NotAllOnConvexHull**:
   - $c_i \in \vartheta CH(V_i)$.
   - $|V_i| \neq n$ **or** $\exists\ r_k$ s.t. $c_k \notin \vartheta CH(V_i)$ **or** $|V_i| = n,\ \forall k \in [1, n],\ c_k \in \vartheta CH(V_i)$, but $\exists r_j$ with no full visibility.
7. **Compute.NotOnStraightLine**:
   - Same conditions as in state 6.
   - There are no two other robots on the same line with robot r_i (on the boundary).
8. **Compute.SpaceForMore**:
   - Same conditions as in state 7.
   - According to V_i there is space on the boundary for another robot: there are two neighboring robots on the boundary with distance at least 2.
9. **Compute.NoSpaceForMore**:
   - Same conditions as in state 7.
   - According to V_i there is no space on the boundary for another robot.
10. **Compute.OnStraightLine**:
    - Same conditions as in state 6.
    - There are at least two other robots on the same line with robot r_i on the boundary.
11. **Compute.SeeOneRobot**:
    - Same conditions as in state 10.
    - Robot r_i can see only one robot on the line.
12. **Compute.SeeTwoRobot**:
    - Same conditions as in state 10.
    - Robot r_i can see two robots on the line; this implies that robot r_i is between these two robots.
13. **Compute.NotOnConvexHull**:
    - Robot r_i is *inside* the convex hull $CH(V_i)$.
14. **Compute.IsTouching**:
    - Same conditions as in state 13.
    - Robot r_i is touching another robot.
15. **Compute.NotTouching**:
    - Same conditions as in state 13.
    - Robot r_i does not touch another robot.
16. **Compute.ToChange**:
    - Same conditions as in state 15.
    - If robot r_i moves as calculated, then the convex hull will change, and this cannot be avoided.
17. **Compute.NotChange**:
    - Same conditions as in state 15.
    - If robot r_i moves as calculated, then there is a way to avoid a change to the convex hull.

Figure 3: The states of the local algorithm, given for a robot r_i, its center c_i and its local view V_i.

LEMMA 4.9. `OnStraightLine`(**Compute.⟨OnStraightLine⟩**) = **Compute.⟨SeeTwoRobots⟩** *iff r_i is on the same line with two robots on the boundary, its left neighbor r_l and its right neighbor r_r.*

11. Procedure `SeeOneRobot`**:** It returns c_i.

LEMMA 4.10. `SeeOneRobot`(**Compute.⟨SeeOneRobot⟩**)= c_i.

12. Procedure `SeeTwoRobot`**:** It returns a point *p* with direction away from the convex hull such that:

LEMMA 4.11. *The point p returned by* `SeeTwoRobot`(**Compute.⟨SeeTwoRobot⟩**) *is such that if robot r_i moves there (c_i is on p), then r_i will no longer be in a straight line with its two adjacent robots on the boundary of the convex hull.*

13. Procedure `NotOnConvexHull`**:** It changes the state from **Compute.NotOnConvexHull** to **Compute.IsTouching** if r_i is touching another robot; otherwise it changes the state to **Compute.NotTouching**.

LEMMA 4.12. `NotOnConvexHull`(**Compute.⟨NotOnConvexHull⟩**) = **Compute.⟨IsTouching⟩** *iff r_i's unit disc is tangent with a unit disc of another robot.*

14. Procedure `IsTouching`**:** Given a geometric configuration (e.g., a robot's local view), we consider that a robot has *higher proximity* compared to the other robots of the configuration if it is the closest to its closest point on the boundary of the convex hull or to the closest point that Function `FindPoints` returns (depending on the case). If more than one robots in the configuration have the same distance to the closest point, then the rightmost of these robots has the highest proximity (straight direction is considered to be towards the outside of the convex hull of the target point). If r_i has the highest proximity, then Procedure `IsTouching` returns a point on the boundary of the convex hull; otherwise it returns c_i.

LEMMA 4.13. `IsTouching`(**Compute.⟨IsTouching⟩**) *will result robot r_i's unit disc to no longer be tangent with any other robot's unit disc (from the robots that r_i touches)* `if` *r_i has the highest proximity (among the robots that are touching). If there is no space of size at least 2 on the boundary of the convex hull, then r_i stays in the same position.*

LEMMA 4.14. `IsTouching`(**Compute.⟨IsTouching⟩**) *will cause at least one of the robots touching each other to move towards the convex hull if there is space of size at least 2 on the boundary.*

15. Procedure `NotTouching`**:** It calls Function `Find-Points`. If it returns at least one point, then the state

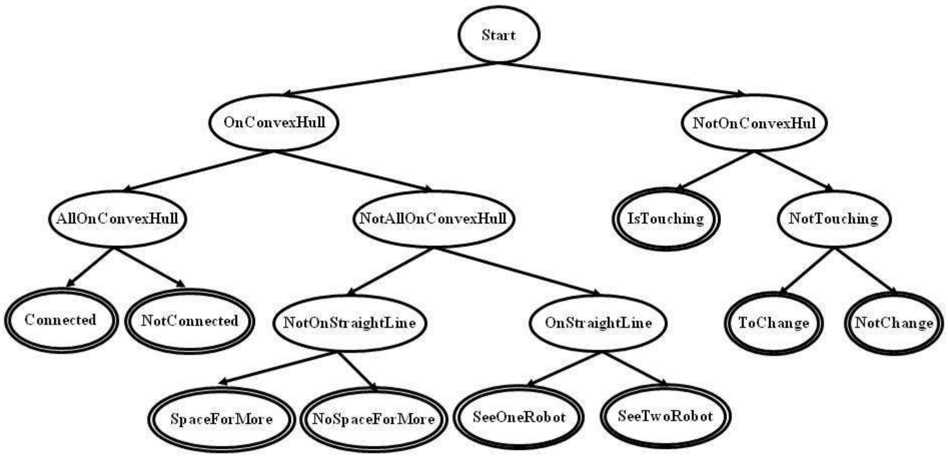

Figure 4: All possible states and transitions of the algorithm run by robot r_i. For better readability the prefix Compute is voided. States with no transition to another state are terminal, and they output the position that the robot will move next (and the robot exits state Compute and enters state Move). State Compute.Connected outputs the special point $\perp$ which causes robot r_i to exit state Compute and enter state Terminate, in which the robot takes no further steps.

changes from **Compute.NotTouching** to **Compute.NotChange**; else it changes to **Compute.ToChange**.

LEMMA 4.15. `NotTouching`(**Compute.⟨NotTouching⟩**) = **Compute.⟨NotChange⟩** *iff r_i can move on the boundary of the convex hull while maintaining the convex hull.*

16. Procedure `ToChange`**:** If there is space of size at least 2 on the convex hull's boundary, then it returns the closest space to r_i; else it returns c_i.

LEMMA 4.16. `ToChange`(**Compute.⟨ToChange⟩**) = *p, when $p \in \vartheta CH(V_i)$ if there is space of size at least 2 on the convex hull; else* `ToChange`(**Compute.⟨ToChange⟩**) = c_i.

17. Procedure `NotChange`**:** It returns the closest point to r_i among the points that Function `Find-Points` returns.

LEMMA 4.17. `NotChange`(**Compute.⟨NotChange⟩**) = *p, where $p \in \vartheta CH(V_i)$.*

5. THE DISTRIBUTED ALGORITHM

The high level idea of the algorithm is as follows: The objective is for the robots to form a convex hull and be able to see each other. In particular, all robots are intended to be on the boundary of the convex hull in such a way that no three robots are on the same line. Once this is achieved, then the robots start to converge (to get closer) while maintaining the convex hull formation, so that they form a connected component. It follows that when all robots are on the boundary of the convex hull, they can see each other, and are connected, then each robot terminates and the gathering problem has been solved. The ***distributed algorithm*** is essentially composed of the asynchronous execution of the robots' state transition cycles, including the local algorithm when in state **Compute**.

Before showing that the distributed algorithm correctly solves the gathering problem, we provide necessary definitions. Given a robot configuration $\mathcal{R}$, we denote by $\mathcal{G}_{\mathcal{R}}$ the geometric configuration of $\mathcal{R}$. Recall that for a geometric configuration $\mathcal{G}$, we denote by $CH(\mathcal{G})$ the convex hull formed by the points in $\mathcal{G}$. Also, we denote by $\vartheta CH(\mathcal{G}) \subseteq \mathcal{G}$ the set of points in $\mathcal{G}$ that are *on the boundary* of the convex hull.

5.1 Bad and Safe Configurations

Our proof of correctness (presented in the next subsection) relies on the notions of *bad* and *safe* configurations, which we discuss here.

Bad Configurations. A robot configuration $\mathcal{R}_x$ is a *bad* configuration when one of the two following cases is true:

1. *Bad configuration of Type 1.* When all of the following hold:
   - Configuration $\mathcal{G}_{\mathcal{R}_x}$ is *fully visible* and $|\vartheta CH(\mathcal{G}_{\mathcal{R}_x})| = n$;
   - A robot r_i in this configuration has as local view V_i, a previous configuration $\mathcal{G}_{\mathcal{R}_y}$, $y < x$, such that $|\vartheta CH(\mathcal{G}_{\mathcal{R}_y})| < n$, $r_i \in \vartheta CH(\mathcal{R}_y)$ and r_i sees that there is not enough space for more robots to get on the boundary of the convex hull.
2. *Bad configuration of Type 2.* When all of the following hold:
   - Configuration $\mathcal{G}_{\mathcal{R}_x}$ is *fully visible* and $|\vartheta CH(\mathcal{G}_{\mathcal{R}_x})| = n$;
   - There is a preceding configuration $\mathcal{G}_{\mathcal{R}_y}$, $y < x$, in which at least four robots, call them r_l, r_{m1}, r_{m2} and r_r, are on a straight line and $r_l, r_{m1}, r_{m2}, r_r \in \vartheta CH(\mathcal{G}_{\mathcal{R}_y})$.

Both types are considered bad since they can potentially lead to a configuration following $\mathcal{R}_x$ that is no longer fully visible or all robots are on the boundary of the convex hull; these are properties that we would like, once holding, to hold for all succeeding configurations.

We now explain how the adversary can cause such bad configurations:

Type 1. According to the local algorithm, when robot r_i witnesses a view as described in the second bullet of Type 1 bad configuration, robot r_i must start moving with direction outside of the convex hull so to make space for more robots to get on the convex hull's boundary. This is also the case for all robots sharing the same or similar view with r_i. When r_i starts moving (it gets in state **move**), the adversary can impose the following strategy: It makes r_i to "move too slow" and lets the other robots move with such "a speed" that the robots reach configuration $\mathcal{R}_x$. Since r_i has not changed its state (it is still in state **move**), it continues to move outside of the convex hull. This may cause a neighboring robot of r_i not to be on the convex hull's boundary anymore or not be able to see all robots. Hence, while $\mathcal{G}_{\mathcal{R}_x}$ was a *fully visible* configuration and

$|\vartheta CH(\mathcal{G}_{\mathcal{R}_x})| = n$, it is possible for a succeeding configuration not to have one (or both) of the these properties anymore.

Type 2. According to the local algorithm, if robots r_l, r_{m1}, r_{m2}, r_r witness configuration $\mathcal{G}_{\mathcal{R}_y}$, then robots r_{m1} and r_{m2} must start moving with direction outside of the convex hull (the robots that realize they are in the middle of the straight line must move outside so to enable the "edge" robots to see each other; the "edge" robots do not move). When r_{m1} and r_{m2} start moving (they get in state **move**) the adversary can impose the following strategy: It lets robot r_{m1} to move slightly and then it stops it (with a $stop(r_{m1})$ event). It lets robot r_{m2} to move slightly and then the adversary makes it to move very slow (so robot r_{m2} is still in state **move**). The adversary could stop robot r_{m1} and delay r_{m2} in such a way that configuration $\mathcal{R}_x$ is reached (recall that $|\vartheta CH(\mathcal{G}_{\mathcal{R}_x})| = n$ and $\mathcal{G}_{\mathcal{R}_x}$ is a *fully visible* configuration). But since r_{m2} continues to move, it is possible to cause robot r_{m1} to no longer be in ϑCH or some other robot (including r_{m2}) not be able to see all other robots. Hence it is possible for a succeeding configuration of $\mathcal{G}_{\mathcal{R}_x}$ not to have one (or both) of the these properties anymore.

Safe Configurations. We say that a robot configuration $\mathcal{R}$ is a *safe* configuration, when the following is true:

$|\vartheta CH(\mathcal{G}_{\mathcal{R}})| = n$, $\mathcal{G}_{\mathcal{R}}$ is *fully visible* **and** $\forall r_i$, $|\vartheta CH(V_i)| = n$ **and** V_i is a *fully visible* configuration (i.e., all robots know that the configuration is fully visible).

The reason we consider such configurations as safe is that, as we will show in the next subsection, once an execution of the algorithm reaches a safe configuration, then no succeeding configuration can be a bad configuration.

We define a *bad execution fragment* (resp., bad execution) of the algorithm to be an execution fragment (resp., execution) that contains at least one bad robot configuration. Similarly, we define a *good execution fragment* (resp., good execution) to be an execution fragment (resp., execution) that contains only good configurations.

5.2 Proof of Correctness

The proof of correctness is broken into two parts. In the first part we prove safety and liveness properties considering only good executions. Then we show that the algorithm is correct for any execution, including ones containing bad configurations. *Omitted and full proofs can be found in the full paper [1].*

5.2.1 Correctness for Good Executions

We first prove safety and then liveness properties.

Safety Properties. The following lemma states that as long as not all robots are on the convex hull's boundary, or even if all robots are on the boundary but there is at least one robot that cannot see all other robots, then the convex hull can only "expand". (Note that this property holds *even* for bad execution fragments.)

LEMMA 5.1. *Given an execution fragment $\mathcal{R}_0, e_1, ..., \mathcal{R}_{m-1}$ such that for all $\mathcal{R}_k$, $0 \le k \le m-1$ holds that:*

c1: $|\vartheta CH(\mathcal{G}_{\mathcal{R}_k})| < n$, **or**

c2: $|\vartheta CH(\mathcal{G}_{\mathcal{R}_k})| = n$ and $\mathcal{G}_{\mathcal{R}_k}$ is not a $fully$ $visible$ configuration.

Then for any step $\langle \mathcal{R}_{m-1}, e_m, \mathcal{R}_m \rangle$, $CH(\mathcal{G}_{\mathcal{R}_{m-1}}) \subseteq CH(\mathcal{G}_{\mathcal{R}_m})$.

PROOF SKETCH: For each possible event e_m, we show that either the invariant is not affected or it is reestablished in configuration $\mathcal{R}_m$. The challenge lies to the fact that robots, due to asynchrony, might have different local views. The detailed case-by-case analysis can be found in [1]. ■

The next lemma states that if in a non-connected configuration all robots are on the boundary of the convex hull and it is fully visible, then these properties are not lost and the convex hull can only "shrink".

LEMMA 5.2. *Given a good execution fragment $\mathcal{R}_x, e_x, \ldots, \mathcal{R}_{m-1}$ such that $\forall \mathcal{R}_k$, $x \le k \le m-1$ holds that*

c1: $|\vartheta CH(\mathcal{G}_{\mathcal{R}_k})| = n$ and $\mathcal{G}_{\mathcal{R}_k}$ is a $fully$ $visible$ configuration, **and**

c2: $\mathcal{G}_{\mathcal{R}_k}$ is not a connected configuration.

Then, for any step $\langle \mathcal{R}_{m-1}, e_m, \mathcal{R}_m \rangle$, c1 holds for $\mathcal{G}_{\mathcal{R}_m}$ and $CH(\mathcal{G}_{\mathcal{R}_{m-1}}) \supseteq CH(\mathcal{G}_{\mathcal{R}_m})$.

As with the previous proof, all possible events e_m are examined.

Liveness Properties. The first liveness lemma states that a configuration where all robots are on the convex hull's boundary and it is fully visible is eventually reached.

LEMMA 5.3. *Given any good execution of the algorithm, there is a configuration $\mathcal{R}_m$ such that $|\vartheta CH(\mathcal{G}_{\mathcal{R}_m})| = n$ and $\mathcal{G}_{\mathcal{R}_m}$ is a fully visible configuration.*

PROOF SKETCH: If $\mathcal{R}_0$ has the stated properties we are done. So consider the case that $\mathcal{R}_0$ satisfies either c1: $|\vartheta CH(\mathcal{G}_{\mathcal{R}_0})| < n$ or c2: $|\vartheta CH(\mathcal{G}_{\mathcal{R}_0})| = n$ and $\mathcal{G}_{\mathcal{R}_0}$ is not a fully visible configuration. By Lemma 5.1, if c1 or c2 holds, then $\vartheta CH(\mathcal{G}_{\mathcal{R}_0})$ can only expand; hence, $\vartheta CH(\mathcal{G}_{\mathcal{R}_0})$ will not shrink unless c1 and c2 do not hold. Then several cases need to be examined, depending whether c1 or c2 is true. For example, if c1 is true, then based on the local algorithm, the robots that are on the boundary of the convex hull do not move (but the robots inside the convex hull do move towards the boundary), unless there is no space for more robots to get on the boundary; in such a case they move to the outside of the convex hull. From Lemma 3.1, and using chirality and the liveness conditions, we argue that eventually there is space for all robots to get on the convex hull's boundary. Then we consider the case of full visibility (i.e., c2 holds). In this case we investigate the cases of how robots might "block" the views of other robots and how the local algorithm arranges so that eventually all robots are able to see all other robots, while all robots keep being on the convex hull. The proof completes by examining various combinations of cases; see [1] for full details. ■

The next lemma states that starting from any initial configuration, when the robots form a configuration where all robots are on the convex hull and they can see each other, they eventually form a connected configuration.

LEMMA 5.4. *Given any good execution of the algorithm, if $\mathcal{R}_l$ is such that $|\vartheta CH(\mathcal{G}_{\mathcal{R}_l})| = n$ and $\mathcal{G}_{\mathcal{R}_l}$ is a fully visible configuration and not a* connected *configuration, then $\exists$ $\mathcal{R}_k$, $l \le k$ so that $\mathcal{R}_k$ is a connected configuration.*

PROOF. Based on Lemma 5.2, if a configuration $\mathcal{R}_m$ is such that $|\vartheta CH(\mathcal{G}_{\mathcal{R}_m})| = n$ and $\mathcal{G}_{\mathcal{R}_m}$ is a *fully visible* configuration, then $|\vartheta CH(\mathcal{G}_{\mathcal{R}_{m+1}})| = n$, $\mathcal{G}_{\mathcal{R}_{m+1}}$ is a *fully visible* configuration and $CH(\mathcal{G}_{\mathcal{R}_m}) \subseteq CH(\mathcal{G}_{\mathcal{R}_{m+1}})$.

Based on Procedure `NotConnected`, no robot will start moving unless: Between any three adjacent robots on the convex hull's boundary, say r_l, r_m and r_r left robot, middle robot and right robot respectively, the distance between line segment $\overline{r_l r_r}$ and r_m must be equal or more than $\frac{1}{n}$. This, along with Lemma 5.3 guarantee that no robot will move unless the distance of at least $\frac{1}{n}$ exists and that eventually all robots will be on the convex hull's boundary and have full visibility. Because no robot moves unless the distance of at least $\frac{1}{n}$ exists, all robots will eventually move to state

Look and see that the observed configuration is *fully visible* and $|\vartheta CH(V_i)| = n$. We get the three following cases:

1. There exists at least one component that is smaller than at least one other component, with respect to the number of the robots that consist each component (size).

   Function `NotConnected` results in all robots of the smallest component(s) to join another larger component. Given the liveness condition that whenever a robot decides to move, it moves at least a distance of δ, eventually the number of the components become smaller and eventually the convex hull shrinks. Also the robots of the components that are not the smallest, do not move.

2. All components are of the same size. The distance between two neighboring components is not the same for all the neighboring components.

   Function `NotConnected` results in all robots of the component that has the smallest distance to its neighbor component on the right, to join the component on its right (here chirality is needed). Given the liveness condition that whenever a robot decides to move, it moves at least a distance of δ, eventually the number of the components become smaller and eventually the convex hull shrinks. The robots of the other components do not move.

3. All components are of the same size, and the distance between any two neighboring components is the same.

   Function `NotConnected` results in all the components start moving with direction to the inside of the convex hull. First, the leftmost and rightmost robots need to move forward at distance $\frac{1}{2n-\varepsilon}$ (due to the required minimum distance of $\frac{1}{n}$, the small steps of $\frac{1}{2n-\varepsilon}$ cannot cause three robots to be on the same line). These robots will not move again until the component has no spaces (this is the reason that a component can have up to 2 spaces). After these robots, the second leftmost and second rightmost robots move to touch the first robots; these robots will not move again until there is a space between them and the first robots. The same happens for the remaining robots. Full visibility and the design of Function `NotConnected` results the component to converge with small steps each time. Given the liveness condition that whenever a robot decides to move, it moves at least a distance of δ, it follows that eventually all the components will touch, because the convex hull shrinks (while preserving its formation).

From the cases above, it follows that either all the robots of any component that has the smallest number of robots (first case) or of any component that has the smallest distance (second case) to its right neighbor will move to its right neighbor until the number of components become one, or the components will move to the inside of the convex hull until all the components touch (third case).

In every case, robot r_i runs the Procedure `NotConnected`. Hence, per Lemma 4.4, robot r_i moves in such a way that it does not cause $|\vartheta CH(\mathcal{G}_{\mathcal{R}_{m+1}})| < n$ or $\mathcal{G}_{\mathcal{R}_{m+1}}$ not to be a $fully\ visible$ configuration. This completes the proof. ■

From Lemmas 5.3 and 5.4 we get the following:

COROLLARY 5.5. *Given any good execution of the algorithm, there is a configuration $\mathcal{R}_m$ so that $\mathcal{G}_{\mathcal{R}_m}$ is a connected and fully visible configuration.*

5.2.2 Correctness for All Executions

We now consider any execution (including ones with bad configurations).

LEMMA 5.6. *Given any execution of the algorithm, if there is a bad execution fragment α_{bad}, then eventually a $safe$ configuration $\mathcal{R}_{safe}$ is reached, after which there are no longer bad configurations.*

PROOF. There are 2 possible cases:
(a) The adversary deploys a strategy that aims in causing bad configurations as long as it can (i.e., indefinitely if possible).
(b) The adversary, at some point of the execution, stops causing bad configurations.

We focus on the first case and we show that any execution under this adversarial strategy will eventually reach a configuration in which the adversary will no longer be able to cause bad configurations. It is easy to see that this case covers also the second case.

Recall that both types of bad configurations involve configurations in which the robots are momentarily in a configuration in which all robots are on the convex hull and it is fully visible, but the adversary manages to break this property. The adversary, as explained, exploits the fact that some robots, due to asynchrony, are not aware that such a configuration has been reached. We now consider the two types of bad configurations.

(i) Bad configuration of type 1. Consider the case in which the first bad configuration, call it $\mathcal{R}_x$, that appears in the bad execution fragment α_{bad} is of type 1 (the other type is considered later). As explained, the adversary may deploy a strategy which can result into a configuration $\mathcal{R}_z,\ z > x$, so that $\mathcal{G}_{\mathcal{R}_z}$ is no longer fully visible or/and not all robots are on the convex hull. The adversary can do so, if there is at least one robot that according to its local view in configuration $\mathcal{R}_x$, not all robot are on the convex hull and there is no more space for an "internal" robot to get on the convex hull (per Function `NoSpaceForMore` this robot will move to a direction outside of the convex hull). It follows that $CH(\mathcal{G}_{\mathcal{R}_z}) \supseteq CH(\mathcal{G}_{\mathcal{R}_x})$. Furthermore, from Lemma 5.1 we get that for all successive configurations of $\mathcal{R}_z$ in which not all robots are on the convex hull or are fully visible, the convex hull can only expand (until a configuration in which these properties hold is reached). The adversary may repeat this strategy (e.g., involving other robots on the convex hull), every time causing the convex hull to expand. However, per Lemma 3.1, this cannot be repeated indefinitely, as the convex hull will expand that much, that the *safe* distance will be reached for all pairs of adjacent robots on the convex hull. From this and that the adversary must allow a robot to move by at least δ distance, it follows that a configuration is eventually reached after which no bad configuration of type 1 can exist (no robot will get into state **Compute.NoSpaceForMore**). Observe that when such a configuration is reached, it is still possible for a bad configuration of type 2 to be reached. This is covered by the next case we consider (with the difference that this bad configuration is not the first appearing in α_{bad}).

(ii) Bad configuration of type 2. Consider the case in which the first bad configuration, call it R_x, that appears in the bad execution fragment α_{bad} is of type 2. This is the situation where in a preceding configuration there are at least four robots on a straight line on the convex hull. As explained in Section 2, the adversary can yield a configuration in which not all robots are any longer on the convex hull, or there is no full visibility. However, per Function `SeeOneRobot` and Lemma 4.10 the robots on the straight line that are not in the middle (i.e., they see only one robot) do not move. In contrast, according to Function `SeeTwoRobot` and Lemma 4.11, each robot in the middle of the straight line moves in a direction outside of the convex hull, in such a way that it will no longer be in a straight line with its two adjacent robots (on the convex hull). It follows that if every time the adversary repeats the same strategy, and say initially there are x robots on straight line, then in every iteration the number of robots that are on the same line is $x-2$. This may continue only until x is less than 3, hence it eventually stops. Observe that during these iterations, since robots in the middle move towards a direction outside of the convex hull and per Lemma 5.1, the convex hull can only expand. Hence a bad configuration of type 2 can no longer exist. Furthermore, note that if during this expansion, the robots involved have also reached the safe distance (per Lemma 3.1's definition), then as explained above, a bad configuration of type 1 also cannot exist. Otherwise, we are back in case (i) as discussed above. Note however that once robots reach the safe distance, and a bad configuration of type 2 is reached, a configuration of type 1 can no longer exist again: when a robot has already safe distance between its adjacent robots on the convex hull, then the middle robots by moving towards outside the convex hull can only increase the safe distance (and hence it will not be possible for a robot to get into state **Compute.NoSpaceForMore**).

From cases (i) and (ii) and Lemma 5.3 it follows that a fully visible configuration in which $|\vartheta CH| = n$ is reached. By a similar argument as in the proof of Lemma 5.4 we get that eventually a safe configuration is reached (all robots are on the convex and they are aware that the configuration is fully visible). From Function `NotConnected` and Lemma 4.4 it follows that any succeeding configuration maintains the property that all robots can see each other and that are on the convex hull. Hence, the algorithm is such that once a safe configuration is reached, it is no longer possible for a bad configuration to exist. This completes the proof. ■

Finally, we prove the correctness of our gathering algorithm.

THEOREM 5.7 (GATHERING). *In any execution of the algorithm, there is a configuration $\mathcal{R}_m$, so that $\mathcal{G}_{\mathcal{R}_m}$ is a connected, fully visible configuration and $\forall s_i \in \mathcal{S}_{\mathcal{R}_m}$, $s_i = \textbf{Terminate}$.*

PROOF. Consider the following two cases.

- If no bad configurations exist, based on Corollary 5.5, given any good execution of the algorithm, there exists $\mathcal{R}_m$ so that $\mathcal{G}_{\mathcal{R}_m}$ is a connected and fully visible configuration.
- If bad configurations exist, based on Lemma 5.6, given any execution of the algorithm, if there is a bad execution fragment α_{bad}, then eventually a $safe$ configuration $\mathcal{R}_{safe}$ is reached, and after a $safe$ configuration there are no longer any bad configurations in the execution until termination. Therefore, from this point onward, we get from Corollary 5.5 that there exists $\mathcal{R}_m$ so that $\mathcal{G}_{\mathcal{R}_m}$ is a connected and fully visible configuration.

When a $connected$ and $fully\ visible$ configuration is reached, it is easy to see that robots no longer move and eventually all robots get into state **Compute.Connected** and hence into state **Terminate**. ■

6. CONCLUSIONS

We have considered the problem of gathering non-transparent, fat robots in an asynchronous setting. We formulated the problem and the model with a state-machine representation and developed a distributed algorithm for any number of robots. The correctness of the algorithm exploits the assumption of *chirality* [11]: robots agree on the orientation of the axes of their local coordination system. This is the only assumption we needed to add to the model considered in [9]. We believe this is a very small price to pay, while we feel it would not be manufacturally unrealistic to provide, in order to solve the gathering problem for any number of fat robots. Nevertheless, an intriguing open problem is to investigate whether it is possible to remove the assumption of chirality and still be able to solve the gathering problem for any number of fat robots, or whether chirality is necessary.

7. REFERENCES

[1] Ch. Agathangelou, Ch. Georgiou, and M. Mavronicolas. A distributed algorithm for gathering many fat robots in the plane. In arXiv:1209.3904, 2012.

[2] N. Agmon and D. Peleg. Fault-tolerant gathering algorithms for autonomous mobile robots. In *Proc. of the 15th ACM-SIAM Symposium on Discrete Algorithms (SODA 2004)*, pages 1070–1078.

[3] H. Ando, Y. Oasa, I. Suzuki, and M. Yamashita. Distributed memoryless point convergence algorithm for mobile robots with limited visibility. *IEEE Transactions on Robotics and Automation*, 15(5):818–828, 1999.

[4] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics.* Second edition, Wiley & Sons, 2004.

[5] A. Bandettini, F. Luporini, G. Viglietta. A survey on open problems for mobile robots. In arXiv:1111.2259v1, 2011.

[6] K. Bolla, T. Kovacs, and G. Fazekas. Gathering of fat robots with limited visibility and without global navigation. In *Proc. of ICAISC/SIDE-EC 2012*, pages 30–38.

[7] S.G. Chaudhuri and K. Mukhopadhyaya. Gathering asynchronous transparent fat robots. In *Proc. of the 6th International Conference on Distributed Computing and Internet Technology (ICDCIT 2010)*, pages 170–175.

[8] A. Cord-Landwehr, B. Degener, M. Fischer, M. Hüllmann, B. Kempkes, A. Klaas, P. Kling, S. Kurras, M. Märtens, F.M.A Der Heide, C. Raupach, K. Swierkot, D. Warner, C. Weddemann, and D. Wonisch. Collisionless gathering of robots with an extent. In *Proc. of SOFSEM 2011*, pages 178–189.

[9] J. Czyzowicz, L. Gasieniec, and A. Pelc. Gathering few fat mobile robots in the plane. *Theoretical Computer Science,* 410(6–7):481–499, 2009.

[10] A. Dutta, S. G. Chaudhuri, S. Datta, and K. Mukhopadhyaya. Circle formation by asynchronous fat robots with limited visibility. In *Proc. of ICDCIT 2012*, pages 83–93.

[11] P. Flocchini, G. Prencipe, N. Santoro. *Distributed Computing by Oblivious Mobile Robots*. Synthesis Lectures on Distributed Computing Theory, Morgan & Claypool, 2012.

[12] P. Flocchini, G. Prencipe, N. Santoro, and P. Widmayer. Gathering of asynchronous robots with limited visibility. *Theoretical Computer Science*, 337(1–3):147–168, 2005.

[13] R.L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Information Processing Letters,* 1(4):132–133, 1972.

[14] S. Kamei, A. Lamani, F. Ooshita, and S. Tixeuil. Gathering an even number of robots in an odd ring without global multiplicity detection. In *Proc. of the 37th International Symposium on Mathematical Foundations of Computer Science (MFCS 2012)*, pages 542–553.

[15] S. Souissi, T. Izumi, and K. Wada. Distributed algorithms for cooperative mobile robots: A survey. In *Proc. of the 2nd Second International Conference on Networking and Computing (ICNC 2011)*, pages 364–371.