Fault-Tolerant Shape Formation in the Amoebot Model

Irina Kostitsyna
Department of Mathematics and Computer Science, TU Eindhoven, The Netherlands

Christian Scheideler
Department of Computer Science, Universität Paderborn, Germany

Daniel Warner
Department of Computer Science, Universität Paderborn, Germany

Abstract

The amoebot model is a distributed computing model of programmable matter. It envisions programmable matter as a collection of computational units called amoebots or particles that utilize local interactions to achieve tasks of coordination, movement and conformation. In the geometric amoebot model the particles operate on a hexagonal tessellation of the plane. Within this model, numerous problems such as leader election, shape formation or object coating have been studied. One area that has not received much attention so far, but is highly relevant for a practical implementation of programmable matter, is fault tolerance. The existing literature on that aspect allows particles to crash but assumes that crashed particles do not recover. We proposed a new model [14] in which a crash causes the memory of a particle to be reset and a crashed particle can detect that it has crashed and try to recover using its local information and communication capabilities. We present an algorithm that solves the hexagon shape formation problem in our model if a finite number of crashes occur and a designated leader particle does not fail. At the heart of our solution lies a fault-tolerant implementation of the spanning forest primitive, which, since other algorithms in the amoebot model also make use of it, is also of general interest.

2012 ACM Subject Classification Theory of computation → Distributed computing models; Theory of computation → Computational geometry

Keywords and phrases programmable matter, amoebot model, fault tolerance, shape formation

Digital Object Identifier 10.4230/LIPIcs.DNA.2022.9

Supplementary Material *Software (Source Code)*: `https://github.com/dwarner-git/AmoebotSim-dna28`; archived at `swh:1:dir:f6dff854d1fc85d9446086ec307437c8487bb7af`

1 Introduction

The research on *programmable matter*, first introduced in [16], deals with small simple particles that locally interact to globally solve a given task like shape formation or coordinated movement. Many future applications are conceivable, such as smart materials, autonomous monitoring and repair, and minimal invasive surgery. The amoebot model, introduced in [4], is a popular model in the context of programmable matter, envisioning particles (or *amoebots*) as computational entities on the micro or nano level. Amoebots have only finite memory and move in a way inspired by amoeba. In this work we extend the amoebot model to include the aspect of *fault tolerance* by introducing *particle crashes* [14]. Crashed particles can recover using their local information and communication capabilities and thus rejoin solving the global task at hand. In order to gain initial insights into useful strategies towards fault tolerance in our model, we examine and solve the *hexagon shape formation problem*. Our solution builds upon two primitives, a propagation primitive and a safety primitive, which both may be of interest in their own.

1.1 Related Work

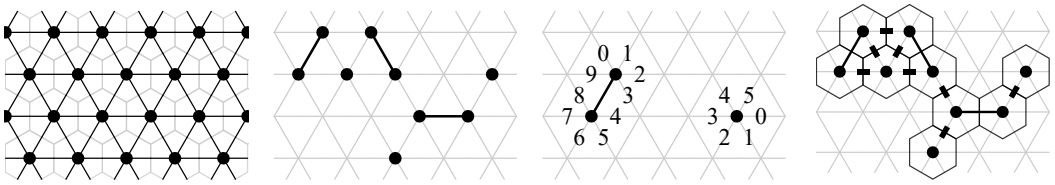
Shape formation in the amoebot model, in particular the hexagon shape formation problem, was investigated in [6, 7, 10]. The authors of [12] show how a constant number of amoebots can be used to simulate a Turing-complete entity that can move in the plane during computation and show how this can be used to form more general shapes than the previously published algorithms. The hexagon shape formation problem was recently revisited in [2] where it was shown how to define an algorithm for the problem that is correct in a concurrent setting.

In the literature on the amoebot model, the topic of fault tolerance has received little attention so far. We are only aware of the following publications: In [11] the authors present a solution to the *line recovery* problem in the amoebot model under a semi-synchronous adversarial scheduler: Initially, n particles, of which $f < n - 4$ are faulty, are positioned on a line. The algorithm forms either a single line or two lines of equal size consisting of all non-faulty particles. Faulty particles may not move or communicate with other particles. An essential difference to our work is that the faulty particles are predetermined by the initial configuration, no crashes occur during the execution of the algorithm and faulty particles cannot recover. The authors of [15] study the *Connected Line Recovery problem*: The proposed algorithm lets the non-faulty particles form a line while maintaining connectivity at all times, regardless of the initial distribution of the faulty particles. It is shown that a necessary condition for the solvability of the problem is that faulty particles can still communicate with their neighbours. Here too, however, it is assumed that faulty particles cannot recover. Daymude et al. present in [3] as an independent partial result the so-called *forest-prune-repair algorithm*, an implementation of the spanning forest primitive that copes with crash failures. Unlike in the present work, the crash failures are permanent there, i.e., particles that have crashed once cannot recover. Furthermore the authors assume that the subgraph induced by the nodes occupied by non-crashed particles is always connected - a strong assumption not necessary in our model where crashes are temporary.

2 Model

In this section, we introduce our model, which is based on the *geometric amoebot model*.

Particles. We assume that particles (*amoebots*) occupy nodes on the triangular lattice $G_\triangle = (V_\triangle, E_\triangle)$ (Figure 1a). A particle occupies either a single node (*contracted* particle) or two adjacent nodes (*expanded* particle). A node can be occupied by at most one particle (Figure 1b). Two particles occupying adjacent nodes are called *neighbours* or *connected*.



(a) A section of $G_\triangle$ (black) and its dual, the hexagonal tiling (gray).

(b) Example showing contracted and expanded particles on $G_\triangle$.

(c) Two particles with different orientations and common chirality.

(d) Adjacent particles establish bonds through the ports facing each other.

Figure 1 Illustration of some aspects of the geometric amoebot model.

Global directions. For $n \in \mathbb{N}$ let $[n] := \{ 0, 1, \ldots, n-1 \}$. A *direction* is an integer in [6] that represents a vector in the Euclidean plane. We distinguish between *global* and *local* directions (see below). The global direction 0 corresponds to the vector pointing right in the triangular lattice $G_\triangle$, and the other global directions increase counter-clockwise.

Port labels. A particle has ports for edges between a node it occupies and a neighbouring node not occupied by it, i.e., a particle has ports for those edges which are incident with exactly one of the nodes it occupies. A contracted particle has six ports and an expanded particle has ten ports (Figure 1c). The ports have unique labels from the particle's own local perspective, as follows: A particle p has a fixed *orientation* $O(p) \in [6]$ that is unknown to it. Particles may have different orientations. We assume that all particles have a common (counter-clockwise) *chirality*.[1] A contracted particle has port labels in [6], an expanded particle has port labels in [10]. If p is contracted, then the edge pointing in global direction $O(p)$ gets label 0. If p is expanded, then there may be two edges pointing in global direction $O(p)$. In this case, the edge pointing in global direction $O(p)$ and pointing "away" from p (i.e., to a node adjacent to only one of the two nodes occupied by p) gets label 0. From label 0, labels increase counter-clockwise around the particle.

Local directions. In order to be able to recognise whether it is contracted or expanded, and to be able to convert port labels into local directions, a particle p has a so-called *bearing* $B(p) \in \{4, 5, 6\} \cup \{\epsilon\}$ which is updated during its movement and can be read by it during its computation. If p is contracted, then $B(p)$ is ϵ. If p is expanded, then p occupies two nodes $\{u, v\} \in E_\triangle$. Let u be the node incident to the edge with port label 0. Then, the bearing $B(p)$ is the port label of the port furthest from u. For a port label l a particle p can easily compute the local direction $d \in [6]$ of the corresponding port using only l and the bearing $B(p)$. In the case of a contracted particle, the label and corresponding local direction are equal. A local direction d corresponds to the global direction $(d + O(p)) \bmod 6$, i.e., $O(p)$ encodes p's offset for local direction 0 from global direction 0. For neighbours p and q and a direction $d_p \in [6]$ local to p let $\Delta(d_p, p, q)$ denote the corresponding direction local to q.

Memory. Each particle has a constant-size local memory. Due to the constant-size memory constraint, particles cannot have a unique id and are therefore anonymous. We assume that a particle has a variable *state* in its local memory.

Communication. Adjacent particles establish bonds through the ports facing each other (see Figure 1d). A particle p can determine for a port label l whether the associated node is empty or occupied, and if it is occupied by another particle q, then p can read q's corresponding label, read q's bearing and read from and write to q's local memory. This effectively enables communication between neighbouring particles.

Movement. Particles can move through *expansion* and *contraction*: When a particle is contracted, it can *expand* into an unoccupied neighbouring node. When a particle is expanded it can *contract* into one of the two nodes it currently occupies. Additionally two neighbouring particles can combine expansion and contraction to perform a coordinated movement: one

[1] The assumption is justified: With the hexagon shape formation problem we assume that a unique leader (SEED particle) is initially available. The SEED could, in the case of non-common chirality, impose its chirality on all other particles in the system (see also Section 4 in [13])

particle can contract out of a node while another expands into that node. This so-called *handover* helps to maintain connectivity of the particle system. We distinguish between two operations: Consider a contracted particle p occupying a node $u \in V_\triangle$ and an expanded particle q occupying two nodes $\{v, w\} \in E_\triangle$. If p and q are neighbours, i.e. $\{u, v\} \in E_\triangle$, then p may *push* q into w. Similarly, q may *pull* p into v. In both cases the result is that p is expanded and occupies the nodes $\{u, v\}$ and that q is contracted and occupies the node w.

The bearing of the particles involved in a movement is updated accordingly.

Head and tail. We assume the following implicitly for our algorithm without noting the implementation in the pseudocode: A particle p has a variable *tailDir* in its local memory, which is ϵ if p is contracted and equal to the direction from which p last expanded if p is expanded. We call the node into which p expanded *head* and the node from which p expanded *tail*. If p is contracted, the node it occupies is called the head. The variable *tailDir* is kept up-to-date during movement operations. In the event of a crash, the information stored in *tailDir* is lost, so that a particle no longer knows which node previously was its head (or tail).

Scheduling. As is common in the amoebot literature, we will assume a sequential fair scheduler for the analysis of our algorithm: In every time step $t \in \mathbb{N}$ the scheduler may either *activate* or *crash* a particle (see below). At any given time at most one particle is active (sequential activations). Furthermore the interval between any two activations of the same particle is finite (fairness property). A *round* is completed as soon as every particle has been activated at least once. An activated particle may use the operations described above: It can read its own local memory and that of its neighbours, perform a computation, update its own memory and the memory of its neighbours and perform at most one movement/handover. Due to the sequential activation model, atomicity and isolation of these actions is ensured.[2]

Model extension: Particle crashes. In our work we extend the amoebot model by introducing *particle crashes*. We assume that the adversarial scheduler may arbitrarily crash particles. A crash of a particle p has the following effects: The scheduler sets the *state* in p's local memory to CRASHED, enabling p and its neighbours to detect that it has crashed, and resets the rest of p's local memory. Note that since the bearing of a particle is not part of the local memory, it cannot be changed by the scheduler. In our algorithm `HexagonFT`, a CRASHED particle detects that it has crashed when it is activated, initialises its memory and initially switches to the additional ERROR *state*; this means in particular that a CRASHED particle is not permanently disabled, but can resume its function after its memory has been reset. A particle in *state* CRASHED or ERROR is called *faulty*. We say a faulty particle *recovers* if it becomes non-faulty.

Configuration. The configuration of a particle $p \in P$ at the beginning of time step $t \in \mathbb{N}$ is the set of nodes $V(p) \subseteq V_\triangle$ it occupies (which implicitly gives its bearing $B(p)$) and its local memory $M(p)$. The configuration $C = (V(p), M(p))_{p \in P}$ of a system P of n particles at the beginning of time step $t \in \mathbb{N}$ describes the configuration of each particle $p \in P$. The *connectivity graph* $G(C)$ is the subgraph of $G_\triangle$ induced by the nodes occupied by particles in configuration C. A configuration is called *non-faulty* if no particles are faulty. A *stable* configuration is a configuration that does not change any more with activations.

[2] Daymude et al. present a formal framework for achieving atomicity and isolation in the amoebot model without this assumption [2]. Furthermore, the authors demonstrate the correctness of their version of the `Hexagon` algorithm under any unfair asynchronous scheduler. We are confident that the analysis could also be applied to our algorithm `HexagonFT` in our extended model.

Problem description: Hexagon shape formation problem. For any two nodes $u, v \in V_\triangle$ the *distance* $\delta(u, v) \in \mathbb{N}_0$ between u and v is defined as the length of a shortest path from u to v in $G_\triangle$. For a node $v \in V_\triangle$ and $i \in \mathbb{N}_0$ let $B(v, i) := \{ u \in V_\triangle \mid \delta(u, v) = i \}$. We call a set $V \subseteq V_\triangle$ a *hexagon with centre* $v \in V$ if there is a $k \in \mathbb{N}_0$ and a subset $S \subseteq B(v, k)$ such that $V = S \cup \bigcup_{i<k} B(v, i)$. We now define the *hexagon shape formation problem* **HEX**: We assume that the system of particles initially forms a single connected component of contracted particles, has a unique leader, called the SEED particle, and that all other particles are IDLE. Furthermore, since we are currently working on a manuscript in which we show that the leader election problem is solvable in our fault tolerance model, we assume in this paper that the SEED particle does not crash. The goal is to reach a stable configuration in which the set of nodes occupied by particles is a hexagon with the SEED in its centre.

3 Algorithm

3.1 Overall Description

General structure. The general structure of our fault-tolerant hexagon shape formation algorithm `HexagonFT` is given as pseudocode in Algorithm 1. All pseudocode is written from the perspective of an activated particle p. As long as no crashes occur, our algorithm `HexagonFT` basically behaves like the classical `Hexagon` algorithm. In Figure 2 we describe how the hexagon shape formation problem **HEX** is solved by `HexagonFT` in the geometric amoebot model without crashes. In the following Section 3.2, we explain the sub-algorithms of `HexagonFT` in detail, in particular how they deal with crashed particles.

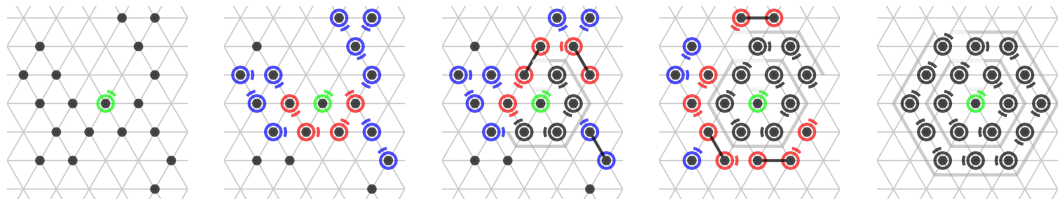


Figure 2 An example run of our hexagon shape formation algorithm `HexagonFT` with 19 particles without crashes. Particles have to assume the shape of a hexagon (but for the outer layer, which may not be completely full). The hexagon is built in a spiral ring in clockwise direction around the SEED as follows: (a) All particles except of the SEED are initially IDLE (black dots). (b) Particles adjacent to finished particles (SEED or RETIRED) become ROOT particles (sub-algorithm `tryToBecomeRoot`), and FOLLOWER particles form parent-child relationships with ROOT or FOLLOWER particles (sub-algorithm `tryToBecomeFollower`). (c)–(e) ROOT particles traverse the forming hexagon counter-clockwise (sub-algorithm `performMovement`), becoming RETIRED (sub-algorithm `tryToBecomeRetired`) when reaching the position marked by the last RETIRED particle. FOLLOWER particles follow ROOT particles via a series of handovers (sub-algorithm `performMovement`).

Variables. The local memory contains the following variables (the initial value is noted after the variable name):

1. *state*: General state. Domain: SEED, IDLE, FOLLOWER, ROOT, RETIRED, ERROR, CRASHED. We call a particle *faulty* if its *state* is CRASHED or ERROR and otherwise *non-faulty*.
2. *tailDir*: Direction of the tail of a non-faulty particle. Domain: $[6] \cup \{\epsilon\}$.

Algorithm 1 `HexagonFT`.

```
1  def HexagonFT(p):
2      if p is finished:
3          return ▷ Finished particles do nothing.
4      if p.state = CRASHED:
5          p.initializeAfterCrash()
6      if p has a neighbour in state CRASHED:
7          return ▷ Wait until all neighbours in state CRASHED are "awake", i.e. in state
                ERROR, and therefore initialized.
8      if p.state = ERROR:
9          p.updateFlags() ▷ Update safeFlags and safeState.
10
11     ▷ Try state change:
12     if p can retire:
13         p.tryToBecomeRetired()
14         if p.state = RETIRED:
15             return ▷ p has become retired and therefore does not do anything more.
16     else if p is in state ERROR, IDLE or FOLLOWER:
17         p.tryToBecomeRoot()
18         if p.state = ROOT:
19             p.pathToRootState ← VALID
20         else if p.state = IDLE:
21             p.tryToBecomeFollower()
22         else if p.state = ERROR and p.safeState = SAFE:
23             p.tryFollowerRecoveryByPropagation()
24
25     ▷ Propagation of pathToRootState:
26     if p.state = FOLLOWER and p.pathToRootState = INVALIDATE:
27         p.pathToRootState ← INVALID
28         p.propagateInvalidate()
29     else if p.state = ROOT or (p.state = FOLLOWER and p.pathToRootState = VALID):
30         p.propagateValid()
31
32     ▷ Movement:
33     if p is in state FOLLOWER or ROOT:
34         p.performMovement()
```

3. *constructionDir*, *moveDir*, *followDir*: Shape formation specific variables of a non-faulty particle. Domain: Direction in [6].
4. *pathToRootState* ← VALID: State of an ERROR or non-faulty particle as node on a path upwards to a ROOT particle. Domain: VALID, INVALID, INVALIDATE.
5. *safeState* ← UNDETERMINED, *safeFlags*[r][d] ← UNDETERMINED (round $r \in [2]$, direction $d \in [6]$): The *safeState* of an ERROR particle is used to determine whether it may become a FOLLOWER. The *safeFlags* are used to determine the current value of *safeState*. Domain: UNDETERMINED = 0, UNSAFE = 1, SAFE = 2.
6. *hasInvalidated*[l] ← FALSE: *hasInvalidated*[l] of an ERROR particle is TRUE if its neighbour at port label l has been invalidated. Domain: FALSE, TRUE.

Terminology. We introduce some notions: A particle is *finished* if its *state* is SEED or RETIRED. A particle p *can retire* if it is CONTRACTED and has a finished neighbour q with *constructionDir* set to the position of p. We call q *(direct) predecessor* of p and p *(direct) successor* of q. Based on this, the terms *predecessor* and *successor* are defined in the canonical way. We say a FOLLOWER particle p *follows* a particle q, if the node in the direction of $p.followDir$ relative to the head of p is occupied by q. We call q *the parent of* p and p *a child of* q. If q is expanded and the node in the direction of $p.followDir$ relative to the head of p is occupied by q's tail, we call p a *tail follower* of q. *Head follower* is defined analogously. An expanded particle has a *blocking tail neighbour* if it has a tail follower or a neighbour in *state* IDLE, ERROR or CRASHED that is adjacent to its tail.

3.2 Sub-Algorithms

In this section the sub-algorithms of algorithm `HexagonFT` are presented.

3.2.1 initializeAfterCrash

As soon as a CRASHED particle is activated, its memory is initialized by sub-algorithm `initializeAfterCrash` (see Algorithm 2): the *state* is set to ERROR and the other variables to default values. To ensure that an activated particle does not read the corrupt memory of CRASHED neighbouring particles, `HexagonFT` (Line 6) also checks whether the particle has CRASHED neighbours and only continues execution if it does not.

Algorithm 2 `initializeAfterCrash`.

```
1 def initializeAfterCrash(p):
2     ▷ Initialization of crashed particle after crash ("wake up"):
3     p.state ← ERROR
4     p.pathToRootState ← INVALID
5     p.safeFlags[r][d] ← UNDETERMINED for all rounds r ∈ [2] and all directions d ∈ [6]
6     p.safeState ← UNDETERMINED
7     p.hasInvalidated[l] ← FALSE for all port labels l
```

3.2.2 tryToBecomeRetired

A particle p that has a finished neighbour q with $q.constructionDir$ set to its position can become RETIRED, if it can successfully determine its *constructionDir*, i.e. the position for the next particle to become RETIRED. The sub-algorithm `tryToBecomeRetired` (see Algorithm 3 and Figure 3) determines whether a particle can become RETIRED, whereby the desired hexagon shape is successively constructed around the SEED in a snake-like fashion.

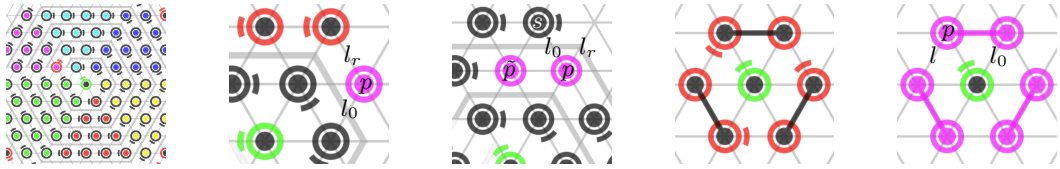
3.2.3 tryToBecomeRoot

A particle in *state* ERROR, IDLE or FOLLOWER that has a finished neighbour but cannot retire could potentially become a ROOT particle. Sub-algorithm `tryToBecomeRoot` (see Algorithm 4 and Figure 3) is responsible for changing the state of a particle to ROOT, if possible. ROOT particles move counter-clockwise around the hexagonal retired structure. The auxiliary

Algorithm 3 `tryToBecomeRetired`.

```
1  def tryToBecomeRetired(p):
2      if p has the SEED s as neighbour at direction d_p:
3          i ← (3 + s.constructionDir − Δ(d_p, p, s)) mod 6 ▷ clockwise index of p with
               respect to SEED s
4          if i = 5: i ← 4
5          p.state ← RETIRED
6          p.constructionDir ← Δ((s.constructionDir − i − 2) mod 6, s, p)
7      else:
8          d ← p's label for port leading to p's predecessor
9          Let q be the particle one label counter-clockwise of p's predecessor at label
               d_q = (d + 1) mod 6.
10         if q.state = RETIRED:
11             if Δ(q.constructionDir, q, p) = d_q: ▷ exception of the rule
12                 p.constructionDir ← (d + 2) mod 6
13             else:
14                 p.constructionDir ← Δ(q.constructionDir, q, p)
15             p.state ← RETIRED
```

function `computeMoveLabel` tries to determine the label in which direction the potential ROOT particle should move next (and returns ϵ if the label could not be determined due to a faulty particle). The auxiliary function `updateMoveDir` tries to update the *moveDir* of the potential ROOT particle and returns TRUE on success.



(a) (b) (c) (d) (e)

Figure 3 `tryToBecomeRetired`: (a) Particles of the same colour have the same *constructionDir*. A particle beyond the first layer can determine its *constructionDir* from a specific neighbour from the previous layer. `tryToBecomeRoot`: (b) p has a finished neighbour at label l_0, and the particle at label l_r is non-finished and non-faulty. p will become a ROOT with its *moveDir* set to the direction of l_r. (c) Special case: p has successfully determined its *moveDir*. Since p is inside the retired structure, it may not become a ROOT, which is ensured by Lines 5–8. (d)–(e) Lines 14–18 prevent that crashing a cycle of expanded ROOT particles leads to a deadlock.

3.2.4 tryToBecomeFollower

According to the spanning forest primitive an IDLE particle that has a ROOT or FOLLOWER as neighbour, follows it by becoming a FOLLOWER and setting its variable *followDir* appropriately, thereby becoming part of the spanning tree. This is realised by sub-algorithm `tryToBecomeFollower`, which is given as pseudocode in Algorithm 5.

Algorithm 4 `tryToBecomeRoot`.

```
1  def tryToBecomeRoot(p):
2      if p is contracted:
3          l_r ← p.computeMoveLabel()
4          if l_r ≠ ε:
5              Let s be the finished particle at label l_s = (l_r + 1) mod 6.
6              d ← Δ(s.constructionDir, s, p)
7              ▷ Check on the basis of s whether p is outside the retired structure:
8              if s is the SEED or (d − l_s) mod 6 ≤ 2:
9                  p.updateMoveDir()
10                 p.state ← ROOT
11     else if p is expanded:
12         if p.updateMoveDir():
13             p.state ← ROOT
14         else if p has a finished neighbour at label l_0 with constructionDir pointing at p:
15             Let l be the in clockwise direction next label after l_0 that does not point to
                   the same node as l_0.
16             Update p.tailDir such that l is a head label of p.
17             p.moveDir ← p's direction of l
18             p.state ← ROOT
19
20 def computeMoveLabel(p):
21     if p has a finished neighbour at some label l_0:
22         Starting at l_0 let l be the in clockwise direction first label at whose port there is no
               finished particle.
23         if there is no faulty particle at label l :
24             return l
25     return ε
26
27 def updateMoveDir(p):
28     l ← p.computeMoveLabel()
29     if l ≠ ε:
30         Update p.tailDir such that l is a head label of p.
31         p.moveDir ← p's direction of l
32         return true
33     return false
```

Algorithm 5 `tryToBecomeFollower`.

```
1  def tryToBecomeFollower(p):
2      if p has a neighbour q in direction d in state ROOT or FOLLOWER:
3          p.state ← FOLLOWER
4          p.followDir ← d
```

3.2.5 tryFollowerRecoveryByPropagation

We now discuss the sub-algorithm `tryFollowerRecoveryByPropagation` (see Algorithm 6), which tries to repair a crashed particle p that could potentially be a FOLLOWER. Our fault-tolerant solution builds upon two primitives, a *safety primitive* and a *validation primitive*, each of which solves a significant problem: Firstly, we have to make sure that p can actually be a FOLLOWER, i.e., we have to exclude the possibility that p could be RETIRED. If this is not ensured, various problematic consequences are possible: among other things, the particles may become disconnected (see Figure 4), it is not guaranteed that a hexagon will be built, and it is even possible that the algorithm will no longer terminate. We show that if a particle is in *safeState* SAFE, it can in principle become a FOLLOWER. Secondly, we need to ensure that p choosing a FOLLOWER *parent candidate* (a FOLLOWER neighbour not following p) does not lead to disconnection of the particles (see Figure 5). In order to avoid disconnection, we use a validation mechanism that determines for a faulty particle which of the FOLLOWER parent candidates it can attach to without closing a cycle. We will see that these are the previously invalidated FOLLOWER parent candidates in *pathToRootState* VALID.

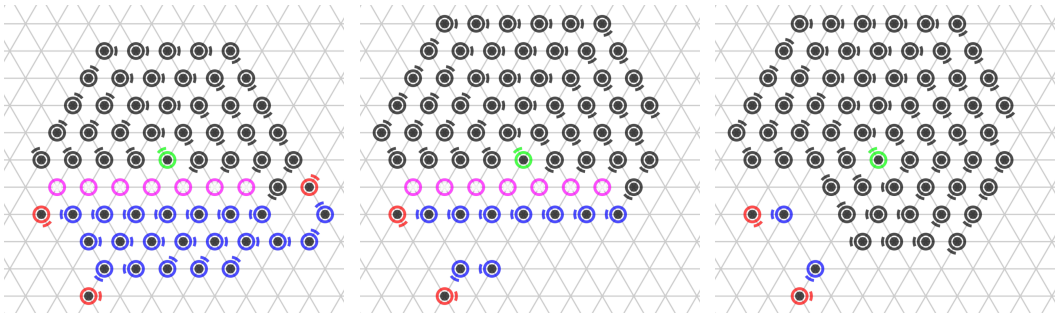


Figure 4 The figure illustrates problems that can arise if it has not been ruled out that a particle to be repaired could be part of the retired structure. (a) Crashed particles have become followers that pass through the structure of originally retired particles. (b) Some followers followed their root and thus split the retired structure. The particles are disconnected. (c) Final stable configuration.

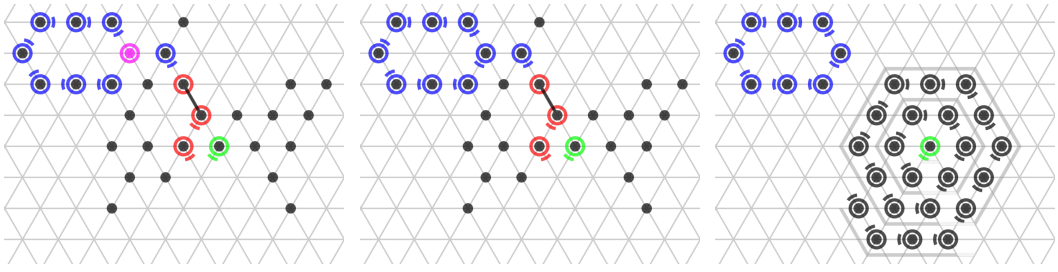


Figure 5 The figure shows that if a crashed particle attaches itself to an arbitrary FOLLOWER pointing away from it, this can lead to irreversible disconnection of the particles. (a) A particle has crashed. (b) The crashed particle attaches itself to the wrong of the two possible FOLLOWER candidates, closing a cycle. (c) Final stable configuration in which the particles are disconnected.

Here, we explain how the validation primitive solves the second problem (compare Figure 6). The safety primitive for solving the first problem is explained in Section 3.2.6.

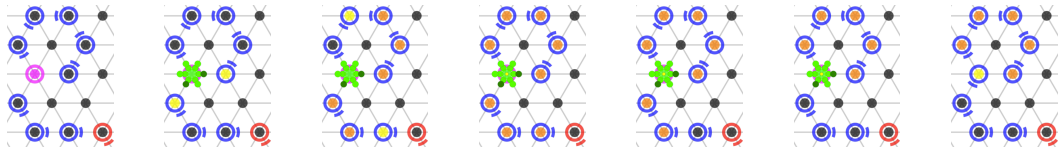


Figure 6 Illustration of the validation primitive. (a) A faulty particle needs to ensure that there is a path to a ROOT before following a particle. (b) The faulty particle sends INVALIDATE tokens to possible parent candidates. (c) INVALIDATE is propagated upwards, causing particles on the path to become INVALID. (d)–(e) An INVALIDATE that reaches an ERROR particle is stored by it, an INVALIDATE that reaches a ROOT is consumed by it. The ROOT generates a VALID token which is propagated downwards along the INVALID particles. (f)–(g) If the VALID token reaches the crashed SAFE particle, it may connect to the FOLLOWER. The INVALIDATE token previously stored by the crashed particle will then again be propagated upwards.

Algorithm 6 `tryFollowerRecoveryByPropagation`.

```
1  def tryFollowerRecoveryByPropagation(p):
2      if p has a neighbour q at label l with q.state = ROOT or (q.state = FOLLOWER and _
           q.pathToRootState = VALID and p.hasInvalidated[l] = TRUE):
3          ▷ Become a FOLLOWER following a ROOT or a previously invalidated VALID
               FOLLOWER parent candidate:
4          p.state ← FOLLOWER
5          Update p.tailDir such that l is a head label of p.
6          p.followDir ← p's direction of l
7      else:
8          ▷ Check whether new FOLLOWER parent candidates have emerged in the
               neighbourhood in the meantime:
9          if p has a FOLLOWER parent candidate q at label l with _
               p.hasInvalidated[l] = FALSE:
10             q.pathToRootState ← INVALIDATE
11             p.hasInvalidated[l] ← TRUE
```

An ERROR particle p in *safeState* SAFE can in principle become a FOLLOWER (see Section 3.2.6). In order to determine an admissible FOLLOWER parent candidate, p employs the following validation mechanism: p invalidates FOLLOWER parent candidates by setting their *pathToRootState* to INVALIDATE and keeps track of which neighbours it has already invalidated in its *hasInvalidated* array, so that each neighbour is invalidated at most once. Subsequently the *pathToRootState* INVALIDATE is propagated "upwards" by followers in the direction of a tree root, such that the particles on the path upwards become INVALID (safety). A special case is when a FOLLOWER in *pathToRootState* INVALIDATE points to an ERROR particle. In this case, the ERROR particle's *pathToRootState* is set to INVALIDATE, but the ERROR particle only continues to propagate if it is repaired to a FOLLOWER. INVALIDATE is finally consumed by a tree root after which the *pathToRootState* VALID is propagated downwards to the leaves (liveness). Note that here it is crucial for safety that an ascending INVALIDATE always has priority over a descending VALID. The propagation of this validation mechanism is realised by the two functions `propagateInvalidate` and `propagateValid` (see Algorithm 7).

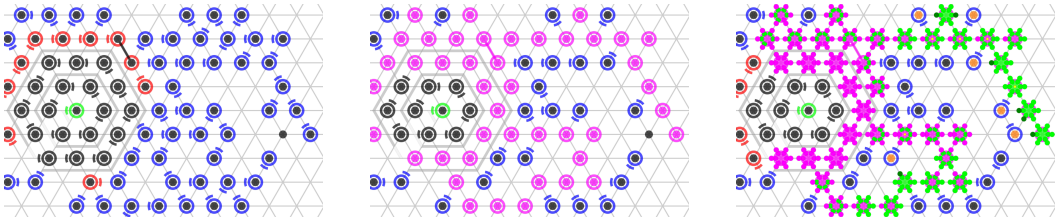
In summary, an ERROR particle p may become a FOLLOWER of a neighbour q if $p.safeState =$ SAFE and q is a ROOT or a FOLLOWER parent candidate in *pathToRootState* VALID which was previously invalidated by p.

Algorithm 7 `propagateInvalidate` and `propagateValid`.

```
1  def propagateInvalidate(p):
2      if p has a FOLLOWER or ERROR parent q:
3          q.pathToRootState ← INVALIDATE
4
5  def propagateValid(p):
6      for each FOLLOWER child q of p with q.pathToRootState = INVALID:
7          q.pathToRootState ← VALID
```

3.2.6 updateFlags

We now present the sub-algorithm `updateFlags` (see Algorithm 8) which implements the *safety primitive* mentioned in the previous Section 3.2.5. The safety primitive basically ensures that particles inside the retired structure (the hexagon built so far) cannot become FOLLOWER particles. The primitive is explained in Figure 7.



(a) (b) (c)

Figure 7 Illustration of the safety primitive. Crashed particles initially are UNDETERMINED and will become either SAFE or UNSAFE. Crashed particles that are connected to a finished particle via one or two line segments in $G_\triangle$ become UNSAFE, otherwise SAFE by the propagation of *safeFlags*. An UNSAFE particle cannot become a FOLLOWER.

Algorithm 8 `updateFlags`.

```
1  def updateFlags(p):
2      if p has a finished neighbour: ▷ Initialize p.safeState based on p's neighbours:
3          p.safeState ← UNSAFE
4      else:
5          p.safeState ← SAFE
6
7      for each round r ∈ [2]: ▷ Update p.safeFlags and p.safeState for two rounds:
8          for each direction d ∈ [6]:
9              S ← { q.safeFlags[r][d_q] | q is an ERROR neighbour of p in direction d }
10             p.safeFlags[r][d] ← min({p.safeState} ∪ S)
11
12         p.safeState ← min { p.safeFlags[r][d] | d ∈ [6] }
```

3.2.7 performMovement

FOLLOWER and ROOT particles move according to the rules in sub-algorithm `performMovement` (see Algorithm 9), which are essentially the same as the movement rules for the classical hexagon shape formation algorithm `Hexagon`, with the following differences: The definition of a *blocking tail neighbour* includes faulty particles. This ensures that the contraction of a particle does not disconnect faulty particles from the rest of the particle system. To ensure that INVALIDATE does not propagate downwards, a particle cannot push a particle that is in *pathToRootState* INVALIDATE. To maintain property F2 for segments (Lemma 4), a pushing FOLLOWER takes over the *pathToRootState* of the pushed particle. A ROOT particle can only move if its *moveDir* was successfully updated by sub-algorithm `updateMoveDir`.

Algorithm 9 `performMovement`.

```
1  def performMovement(p):
2      if p is an expanded FOLLOWER or ROOT that has no blocking tail neighbour:
3          p contracts into its head.
4      else if p is a contracted tail FOLLOWER of a FOLLOWER or ROOT particle q:
5          if q.pathToRootState ≠ INVALIDATE: ▷ Ensure that INVALIDATE does not
               propagate downwards.
6              p pushes q and updates p.followDir accordingly.
7              p.pathToRootState ← q.pathToRootState
8      else if p is a contracted ROOT and p.updateMoveDir():
9          if p has no neighbour in direction p.moveDir:
10             p expands in direction p.moveDir.
11         else if p has the tail of an expanded ROOT in direction p.moveDir:
12             p pushes in direction p.moveDir.
```

3.3 Analysis

In this section we show that our algorithm `HexagonFT` solves the hexagon shape formation problem **HEX**, if a finite number of crashes occur. Due to space constraints, most proofs are given in the appendix in Appendix A.1. We assume w.l.o.g. that the SEED is positioned on $(0,0) \in V_\triangle$ and has orientation 0. Our algorithm constructs the hexagon in a spiral ring around the SEED. A particle that retires sets the direction *constructionDir* for the next particle to become RETIRED. The *constructionDir* for a particle beyond the first layer is determined by the *constructionDir* of a specific neighbouring particle in the previous layer (compare algorithm `tryToBecomeRetired` and Figure 3a). The following lemma provides the basis for the correctness of this procedure:

▶ **Lemma 1.** *For $v \in V_\triangle$ and $d \in [6]$, let $n(v,d) \in V_\triangle$ denote the neighbour of v in direction d. The sequence of nodes v_i defined inductively by $v_0 = (0,0) \in V_\triangle$, $v_k = n(v_{k-1}, d(v_{k-1}))$, $d(v_k) = (0,4,3,2,1,0,0,5)$ for $0 \le k \le 7$, $d_k = d(n(v_{k-1}, (d(v_{k-1}) - 1) \bmod 6))$ for $k > 7$ is well-defined, and it holds: For any $k \ge 0$ the nodes $v_0, \dots, v_k$ form a* spiral ring *around the centre $(0,0)$ such that the set $\{v_j | 0 \le j \le k\}$ forms a hexagon with centre $(0,0)$.*

▶ **Definition 1.** *For a configuration C let $k \in \mathbb{N}_0$ such that for all $j \in [k]$ the node v_j of the spiral ring is occupied by a finished or faulty particle, and node v_{k+1} is empty or occupied by a non-finished or non-faulty particle. We call the set of particles that occupy the nodes v_j for $j \in [k]$ the* retired structure*. A particle inside (outside) the retired structure is called an* interior *(*exterior*) particle. An exterior particle adjacent to an interior particle is called a* boundary *particle.*

▶ **Definition 2.** *Based on a configuration C, we define a directed graph $A(C)$ as follows: The node set of $A(C)$ is the set of particles. If p is a* FOLLOWER *following a particle q that is a* FOLLOWER*, a* ROOT *or a faulty exterior particle, then $A(C)$ contains a directed edge from p to q. If p is a faulty exterior particle and q a neighbour of p that is a* FOLLOWER *parent candidate of p, a* ROOT *or a faulty exterior particle, then $A(C)$ contains a directed edge from p to q. A simple path or simple circuit $p_1, \ldots, p_k$, $k > 1$ in $A(C)$ from a faulty exterior particle p_1 to a faulty exterior particle p_k, where all p_j for $1 < j < k$ are* FOLLOWER *particles, is called a* segment*. We call a particle a* tree root *if it is a* FOLLOWER *following an interior particle, a* ROOT *or a faulty boundary particle.*

3.3.1 Safety

▶ **Lemma 2** (Retired structure)**.**

R1 *At any time, all finished particles are interior particles.*

R2 *At any time, a finished particle occupying a node v_j of the spiral ring has its* constructionDir *set in the direction of the node v_{j+1}.*

R3 *An interior particle will always remain an interior particle.*

▶ **Lemma 3** (Roots)**.** *At any time, the following* root property *holds: The head of a* ROOT *particle is adjacent to an interior particle.*

▶ **Lemma 4** (Followers)**.** *At any time, the following properties hold:*

F1 *For a* FOLLOWER *particle p, there is a simple path in $A(C)$ from p to a tree root.*

F2 *If $p_1, \ldots, p_k$ is a segment and l the label of p_1 for the port leading to p_2, then one of the two following statements holds:*

- **a.** p_1.state = CRASHED *or (*p_1.state = ERROR *and* p_1.hasInvalidated$[l]$ = FALSE*) or*
- **b.** p_1.state = ERROR *and* p.hasInvalidated$[l]$ = TRUE*, and one of the three following statements holds:*
  - **i.** *There exists $1 < j < k$ such that* p_i.pathToRootState = INVALID *for all $1 < i < j$ and* p_j.pathToRootState = INVALIDATE*.*
  - **ii.** p_i.pathToRootState = INVALID *for all $1 < i < k$ and (p_k crashed after p_1 or* p_k.pathToRootState = INVALIDATE*)*
  - **iii.** *There exists $1 < j < k$ such that* p_i.pathToRootState = INVALID *for all $1 < i < j$ and* p_j.pathToRootState = VALID*, and there is a simple path in $A(C)$ from p_j to a tree root, on which all faulty particles crashed after p_1.*

▶ **Lemma 5** (Connectivity)**.** *At any time, all particles are connected, i.e., $G(C)$ is one connected component.*

▶ **Lemma 6.** *Every connected component of* IDLE *particles is connected to at least one non-*IDLE *particle.*

3.3.2 Liveness: Recovery

▶ **Lemma 7** (Recovery)**.** *If a finite number of crashes occur during the execution of algorithm* `HexagonFT` *and m particles are faulty after the last crash, then a non-faulty configuration is reached within $\mathcal{O}(mn)$ rounds after the last crash.*

Proof. We show that within $\mathcal{O}(n)$ rounds at least one faulty particle recovers. Assume that at the beginning of round t all faulty particles are in *state* ERROR. Let C be the configuration at the beginning of round t. We consider the following cases at the beginning of round t, assuming that there is still at least one faulty particle:

Case 1: There is a faulty interior particle: Starting at the SEED, let p be the first faulty interior particle in the spiral ring around the SEED. Clearly, p will become RETIRED within one round.

Case 2: There is no faulty interior particle but a faulty boundary particle: Let v be the first node in the spiral ring around the SEED occupied by a faulty particle p and u the predecessor node of v in the spiral ring. When p is activated in round t, node u is either empty or occupied by a non-faulty particle. If p is contracted and u occupied by a finished particle, p retires. Otherwise, p becomes a ROOT.

Case 3: There is no faulty interior or boundary particle but a faulty exterior non-boundary particle q that has a ROOT or a FOLLOWER as neighbour: We only consider the case that q has a FOLLOWER q' as neighbour. The proof for the case that q has a ROOT as neighbour is analogous. By F1 there is a simple path in $A(C)$ from q' to a tree root. If there is no faulty particle on that path, then let $p = q$ and $p' = q'$, otherwise, let p be the faulty particle on that path nearest to the tree root and p' its successor occupying some node u adjacent to p. Let l be a port label of p pointing towards u. Let $k = 4$. Suppose no faulty particle recovers within kn rounds. By Lemma 5, p is part of a connected component F of faulty particles. The component F does not change within kn rounds since no particles crash and no faulty particles recover during this period. At the beginning of round t no particle in F has a interior or boundary particle as neighbour. This also does not change within kn rounds, as we assume that no faulty particles recover during this period. It follows that $p.safeState$ is SAFE at the beginning of round $t + t_1$ for some $t_1 \leq 2n$. Note that from round $t + t_1$ onwards, node u must be occupied by a FOLLOWER parent candidate for p that lies on a path in $A(C)$ of non-faulty particles up to a tree root. We can make the following observations: Since no faulty particle recovers within kn rounds, no new FOLLOWER parent candidates emerge in the neighbourhood of faulty particles. The *pathToRootState* INVALIDATE is propagated upwards by FOLLOWER particles. Therefore, after at most n rounds there are no FOLLOWER particles in *pathToRootState* INVALIDATE. It follows, since then the *pathToRootState* VALID propagates downwards, that after at most n further rounds the node u is occupied by a FOLLOWER parent candidate p'' for p in *pathToRootState* VALID. Since $p.hasInvalidated[l]$ = TRUE, it follows that p becomes a FOLLOWER after a total of at most kn rounds, i.e. recovers, a contradiction. Therefore, at least one faulty particle recovers within kn rounds.

Case 4: There is no faulty interior or boundary particle and no faulty exterior non-boundary particle that has a ROOT or a FOLLOWER as neighbour but a faulty particle that has an IDLE neighbour: By Lemma 5, the graph $G(C)$ is connected. Let P be any simple path in $G(C)$ from the SEED particle to some faulty particle that has an IDLE neighbour. Let r be the first faulty particle on P, i.e. the one closest to the SEED on the path P, and let q be its predecessor. Clearly, q is an IDLE particle and will become non-IDLE after $O(n)$ rounds. If r has not yet recovered by this time, the statement now follows with one of the previous cases. ◀

For the upper bound given in Lemma 7, there is a matching (algorithm-depended) lower bound: If all n particles lie on a line and all particles except the SEED are CRASHED, then the algorithm `HexagonFT` needs $\Omega(n^2)$ rounds until a non-faulty configuration is reached.

3.3.3 Termination

We now state our main result:

▶ **Theorem 1.** *If a finite number of crashes occur, then the algorithm* `HexagonFT` *solves the hexagon shape formation problem* **HEX** *in worst-case* $\mathcal{O}(n^2)$ *work (total number of moves executed by all particles). From the time when no more crashes occur and the configuration is non-faulty, the algorithm needs* $\mathcal{O}(n)$ *rounds until termination.*

4 Conclusion and Future Work

We have shown that our algorithm `HexagonFT` solves the hexagon shape formation **HEX** in our new model. `HexagonFT` can easily be transferred to other shapes (e.g. square, triangle, line), and the fault-tolerant implementation of the spanning forest primitive can be used for algorithms that employ it. We have assumed that there is a unique SEED particle in the initial configuration that does not fail. We show that this assumption is justified in a manuscript in progress in which we prove that the leader election problem can be solved in our fault tolerance model. Our result is based on the assumption of a finite number of particle crashes. We have extended our simulation environment *AmoebotSim* with the fault tolerance model presented here and implemented our fault-tolerant hexagon shape formation algorithm in it. In our simulations we observed that `HexagonFT` terminates successfully even if particle crashes occur continuously, as long as the rate of crashes is not too high. It could be interesting to investigate under what assumptions the algorithm terminates, if crashes occur continuously. Although the upper bound given in Lemma 7 has a matching algorithm-dependent lower bound, determining a problem-dependent lower bound remains an open problem. It might be interesting to transfer our ideas to other problems in the amoebot model, for example: coating problems [9, 5, 8], bridging problems [1] or general shape formation problems. In view of [11, 15], the question arises, how to deal with particles that permanently crash, i.e. cannot recover. Further research could investigate model variants, e.g. hybrid models in which failures can be both temporary and permanent, models in which the system is already not well-initialised to begin with, and models in the context of self-stabilization. Furthermore, we propose to investigate how a framework could look like that makes it possible to transform any algorithm A without error states into an equivalent algorithm A' with error states that is robust w.r.t. an adversary as specified above.

References

1 Marta Andrés Arroyo, Sarah Cannon, Joshua J Daymude, Dana Randall, and Andréa W Richa. A stochastic approach to shortcut bridging in programmable matter. *Natural Computing*, 17(4):723–741, 2018.

2 Joshua J Daymude, Andréa W Richa, and Christian Scheideler. The canonical amoebot model: Algorithms and concurrency control. In *35th International Symposium on Distributed Computing*, 2021.

3 Joshua J Daymude, Andréa W Richa, and Jamison W Weber. Bio-inspired energy distribution for programmable matter. In *International Conference on Distributed Computing and Networking 2021*, pages 86–95, 2021.

4 Zahra Derakhshandeh, Shlomi Dolev, Robert Gmyr, Andréa W Richa, Christian Scheideler, and Thim Strothmann. Brief announcement: amoebot–a new model for programmable matter. In *Proceedings of the 26th ACM symposium on Parallelism in algorithms and architectures*, pages 220–222, 2014.

5 Zahra Derakhshandeh, Robert Gmyr, Alexandra Porter, Andréa W Richa, Christian Scheideler, and Thim Strothmann. On the runtime of universal coating for programmable matter. In *International Conference on DNA-Based Computers*, pages 148–164. Springer, 2016.

6 Zahra Derakhshandeh, Robert Gmyr, Andréa W Richa, Christian Scheideler, and Thim Strothmann. An algorithmic framework for shape formation problems in self-organizing particle systems. In *Proceedings of the Second Annual International Conference on Nanoscale Computing and Communication*, pages 1–2, 2015.

7 Zahra Derakhshandeh, Robert Gmyr, Andréa W Richa, Christian Scheideler, and Thim Strothmann. Universal shape formation for programmable matter. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 289–299, 2016.

8 Zahra Derakhshandeh, Robert Gmyr, Andréa W Richa, Christian Scheideler, and Thim Strothmann. Universal coating for programmable matter. *Theoretical Computer Science*, 671:56–68, 2017.

9 Zahra Derakhshandeh, Robert Gmyr, Andréa W Richa, Christian Scheideler, Thim Strothmann, and Shimrit Tzur-David. Infinite object coating in the amoebot model. *arXiv preprint*, 2014. `arXiv:1411.2356`.

10 Giuseppe A Di Luna, Paola Flocchini, Nicola Santoro, Giovanni Viglietta, and Yukiko Yamauchi. Shape formation by programmable particles. *Distributed Computing*, 33(1):69–101, 2020.

11 Giuseppe Antonio Di Luna, Paola Flocchini, Giuseppe Prencipe, Nicola Santoro, and Giovanni Viglietta. Line recovery by programmable particles. In *Proceedings of the 19th International Conference on Distributed Computing and Networking*, pages 1–10, 2018.

12 Giuseppe Antonio Di Luna, Paola Flocchini, Nicola Santoro, Giovanni Viglietta, and Yukiko Yamauchi. Mobile ram and shape formation by programmable particles. In *European Conference on Parallel Processing*, pages 343–358. Springer, 2020.

13 Yuval Emek, Shay Kutten, Ron Lavi, and William K Moses Jr. Deterministic leader election in programmable matter. *arXiv preprint*, 2019. `arXiv:1905.00580`.

14 Irina Kostitsyna, Christian Scheideler, and Daniel Warner. Brief Announcement: Fault-Tolerant Shape Formation in the Amoebot Model. In James Aspnes and Othon Michail, editors, *1st Symposium on Algorithmic Foundations of Dynamic Networks (SAND 2022)*, volume 221 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 23:1–23:3, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. `doi:10.4230/LIPIcs.SAND.2022.23`.

15 Nooshin Nokhanji and Nicola Santoro. Line reconfiguration by programmable particles maintaining connectivity. In *International Conference on the Theory and Practice of Natural Computing*, pages 157–169. Springer, 2020.

16 Tommaso Toffoli and Norman Margolus. Programmable matter: concepts and realization. *Physica. D, Nonlinear phenomena*, 47(1-2):263–272, 1991.

A Appendix

A.1 Omitted Proofs

In this section we give all the proofs that have been omitted from the main part due to space constraints in a rigorous form.

Proof of Lemma 1. Define $v_{0,0} = (0,0) \in V_\triangle$, $d_{0,0} = 0$, $d_{1,k} = (4,3,2,1,0,0)_{1\le k\le 6}$ and $d_{2,1} = 5$. Inductively define $\pi(l,k) = (l-1, 6(l-1))$ for $l \ge 1$, $k = 1$, $\pi(l,k) = (l,k-1)$ for $l \ge 1$, $1 < k \le 6l$, $d_{l,1} = d_{l-1,1}$ for $l > 2$, $d_{l,jl+i} = d_{l-1,j(l-1)+(i-1)}$ for $l \ge 2$, $0 \le j \le 5$, $2 \le i \le l$, $d_{l,jl+1} = d_{l-1,j(l-1)}$ for $l \ge 2$, $1 \le j \le 5$, $v_{l,k} = n(v_{\pi(l,k)}, d_{\pi(l,k)})$ for $l \ge 1$, $1 \le k \le 6l$, $\sigma(v_{l,k}) = (l,k)$ for $l \ge 1$, $1 \le k \le 6l$ and $\rho(l,k) = n(v_{\pi(l,k)}, (d_{\pi(l,k)} - 1) \bmod 6)$ for $l \ge 2$, $1 \le k \le 6l$, $(l,k) \ne (2,1)$. By induction the following properties follow: $\rho(l,1) = v_{l-1,1}$ for $l > 2$, $\rho(l, jl+i) = v_{l-1,j(l-1)+(i-1)}$ for $l \ge 2$, $0 \le j \le 5$, $2 \le i \le l$, $\rho(l, jl+1) = v_{l-1,j(l-1)}$ for $l \ge 2$, $1 \le j \le 5$ and $d_{l,k} = (5^{l-1}, 4^l, 3^l, 2^l, 1^l, 0^{l+1})_{1\le k\le 6l}$ for $l \ge 1$. This implies $d_{l,k} = d_{\sigma(\rho(l,k))}$ for $l \ge 2$, $1 \le k \le 6l$, $(l,k) \ne (2,1)$ and that the nodes $v_{l,1}, \dots, v_{l,6l}$ for $l \ge 1$ form a single ring of radius l around the centre $(0,0) \in V_\triangle$ with corners $v_{l,jl}$ for

$1 \leq j \leq 6$. The nodes in the defined order therefore form a spiral ring around the centre $(0,0)$, in particular for any $l \geq 1$ and $1 \leq k \leq 6l$ the set of all predecessors of $v_{l,k}$ forms a hexagon with centre $(0,0)$. ◀

▶ **Lemma 2** (Retired structure)**.**

R1 *At any time, all finished particles are interior particles.*

R2 *At any time, a finished particle occupying a node v_j of the spiral ring has its* constructionDir *set in the direction of the node v_{j+1}.*

R3 *An interior particle will always remain an interior particle.*

Proof. In an initial configuration the properties hold trivially. Assume the properties hold for a configuration C. Consider that **a particle p becomes retired (Line 13):** Assume that `tryToBecomeRetired` will change p's *state* to RETIRED. Let $\tilde{p}$ be the finished direct predecessor of p. By R1, $\tilde{p}$ is an interior particle occupying some node v_j of the spiral ring. By R2, p occupies node v_{j+1} of the spiral ring. The structure of the spiral ring is precisely characterised by Lemma 1 and implemented accordingly in `tryToBecomeRetired`: The particles in the first layer of the hexagon are neighbours of the SEED. If p has the SEED as neighbour, then $p.constructionDir$ is uniquely determined by the *constructionDir* of the SEED and the position of p relative to the SEED (Lines 3–6). If p is a particle beyond the first layer of the hexagon, then $p.constructionDir$ is, with one exception (Line 12), determined by the previous layer of the hexagon (Line 14, Figure 3a; the exception is the first particle in the second layer; the particle is highlighted with a red border). In any case $p.constructionDir$ is set correctly in the direction of the node v_{j+2}. Therefore, all properties still hold after the state change. Now we show that **an interior particle will always remain an interior particle:** Suppose the contrary. Let t_1 be the first time when an interior particle p gets activated and subsequently is no interior particle. Since finished particles do nothing and CRASHED particles become initialized by `initializeAfterCrash`, we can assume that p is in *state* ERROR. Note that all predecessor nodes of the node occupied by p in the spiral ring around the SEED are occupied by finished or faulty particles. At the end of the activation, p is no interior particle; we distinguish the following *state* changes (Line 11):

Case 1: p becomes a ROOT (Line 17): Assume that `tryToBecomeRoot` will change p's *state* to ROOT. Note that p cannot be the successor of the SEED. Since $l_r \neq \epsilon$, p cannot be entirely surrounded by faulty or finished particles. Let $\tilde{p}$ be the direct predecessor of p in the spiral ring.

- **Case a:** $\tilde{p}$ is not entirely surrounded by faulty or finished particles: Then, starting at any finished neighbour of p, all in clockwise direction traversed neighbouring nodes of p up to and including $\tilde{p}$ are occupied by finished or faulty particles. Since p cannot retire, $\tilde{p}$ must be faulty. But from this follows $l_r = \epsilon$, a contradiction.
- **Case b:** $\tilde{p}$ is entirely surrounded by faulty or finished particles (Figure 3c): By an analogous argument as in the previous case, the finished particle s (Line 5) must be the last interior particle in the spiral ring. Since $(d - l_s) \bmod 6 \leq 2$, p is one or two labels counter-clockwise of $s.constructionDir$ w.r.t. s, i.e., p is outside of the retired structure, a contradiction.

Case 2: p becomes a FOLLOWER (Line 23): Note that the *state* change in `tryFollowerRecoveryByPropagation` only takes place if $p.safeState =$ SAFE holds. So we must have $p.safeState =$ SAFE after the call of p.`updateFlags`. By u denote the node occupied by p and by w the node occupied by the SEED. One of the following two statements holds:

- **Case a:** There exists a line segment L in $G_\triangle$ from u to w such that all nodes (except u) on L are predecessors of u in the spiral ring around the SEED.

Case b: There exists a node v and line segments L_1, L_2 in $G_\triangle$ from u to v, and from v to w, respectively, such that all nodes (except u) on L_1 and L_2 are predecessors of u in the spiral ring around the SEED.

We only consider the latter case; in the former case, the proof is analogous. Let $p = p_1, \ldots, p_j$ and $p_j, \ldots, p_k =$ SEED be the interior particles occupying the nodes of the line segments L_1 and L_2, respectively. Note that p_{i+1} is a predecessor of p_i in the spiral ring for all $1 \leq i < k$. To simplify the notation, we assume in the following that local and global directions correspond. By d_1 and d_2 denote the direction of L_1 and L_2, respectively. We argue in the following by means of backward analysis: At time t_1 particle p_2 cannot be finished or in *state* CRASHED, since otherwise $p.safeState \neq$ SAFE after the call of p.`updateFlags`. Therefore, p_2 must be in *state* ERROR with $p_2.safeFlags[1][d_1] =$ SAFE. Let t_2 be the time before t_1 when p_2 was last activated. At time t_2 particle p_3 cannot be finished or in *state* CRASHED, since otherwise $p_2.safeFlags[1][d_1] \neq$ SAFE. Therefore, p_3 must be in *state* ERROR with $p_3.safeFlags[1][d_1] =$ SAFE. By repeated application of the argument, it follows that at some time t_{j-1}, p_j must be in *state* ERROR with $p_j.safeFlags[1][d_1] =$ SAFE, and therefore $p_j.safeFlags[0][d_2] =$ SAFE. Further repeated application of the argument finally yields that at some time t_{k-2}, p_{k-1} must be in *state* ERROR with $p_{k-1}.safeFlags[0][d_2] =$ SAFE, contradicting that p_{k-1} has the finished neighbour $p_k =$ SEED. ◀

▶ **Lemma 3** (Roots). *At any time, the following* root property *holds: The head of a* ROOT *particle is adjacent to an interior particle.*

Proof. In an initial configuration the root property holds trivially. Assume the root property holds for a configuration C. By C' denote the successor configuration of C. One can easily check that a particle p becomes a ROOT by `tryToBecomeRoot` (`HexagonFT`: Line 17) only if it has a finished neighbour q, and that `tryToBecomeRoot` ensures that the head of p is adjacent to q. By R1, q is an interior particle. Therefore, the root property holds for p in C'. Now let p be a ROOT in C. By R3, an interior particle will always remain an interior particle. Therefore, the root property for p could only be violated in C' by movement of p (`HexagonFT`: Line 32):

Case 1: p is an expanded FOLLOWER or ROOT that has no blocking tail neighbour: Since p contracts into its head, the root property holds for p in C'.

Case 2: p is a contracted tail FOLLOWER of a FOLLOWER or ROOT particle q: Since p pushes q into its head, the root property holds for q in C'.

Case 3: p is a contracted ROOT and `updateMoveDir`(p) succeeds: In both subcases: After the expansion, the position of p's head has changed. Due to the choice of label by `computeMoveLabel` and making it a head label, p still has a finished (and therefore interior) particle adjacent to its head in C', so the root property holds for p in C'. ◀

▶ **Lemma 4** (Followers). *At any time, the following properties hold:*

F1 *For a* FOLLOWER *particle p, there is a simple path in $A(C)$ from p to a tree root.*

F2 *If $p_1, \ldots, p_k$ is a segment and l the label of p_1 for the port leading to p_2, then one of the two following statements holds:*

- **a.** p_1.state = *CRASHED or* (p_1.state = *ERROR and* p_1.hasInvalidated[l] = *FALSE*) *or*
- **b.** p_1.state = *ERROR and* p.hasInvalidated[l] = *TRUE, and one of the three following statements holds:*
  - **i.** *There exists* $1 < j < k$ *such that* p_i.pathToRootState = *INVALID for all* $1 < i < j$ *and* p_j.pathToRootState = *INVALIDATE*.

ii. p_i.pathToRootState = INVALID *for all* $1 < i < k$ *and (*p_k *crashed after* p_1 *or* p_k.pathToRootState = INVALIDATE*)*

iii. *There exists* $1 < j < k$ *such that* p_i.pathToRootState = INVALID *for all* $1 < i < j$ *and* p_j.pathToRootState = VALID*, and there is a simple path in* $A(C)$ *from* p_j *to a tree root, on which all faulty particles crashed after* p_1*.*

Proof. In an initial configuration the properties holds trivially. Assume the properties holds for a configuration C. We show in the following that the properties remain true for the successor configuration C' of C resulting from a crash or a configuration change by the algorithm. **Particle crash:** Clearly, F1 remains true. For F2 assume that an exterior particle p crashes and let $S = (p_1, \dots, p_k)$ be a segment in $A(C')$. We consider the following cases:

Case 1: p is not a particle on segment S: Clearly, F2 holds for S in C'.

Case 2: $p = p_1$: F2a holds for S in C'.

Case 3: $p = p_k$ and $p_1 \neq p_k$: It is sufficient to show that in the case of F2b one of the three properties holds for S in C'.

Case a: $p_i.pathToRootState$ = INVALID for all $1 < i < k$: F2(b)i holds for S in C'.

Case b: There exists $1 < j < k$ such that $p_i.pathToRootState$ = INVALID for all $1 < i < j$ and $p_j.pathToRootState \neq$ INVALID: If $p_j.pathToRootState$ = INVALIDATE, then F2(b)i holds for S in C'. Assume $p_j.pathToRootState$ = VALID. By F1, there is a simple path P in $A(C)$ from p_j to a tree root.

Case: There is no faulty particle on P: F2(b)iii holds for S in C'.

Case: There is a faulty particle on P: Let q be the first faulty particle on P. The path from p_1 to q in $A(C)$ is a segment for which F2(b)iii holds in C. Together with the fact that p_k crashed after p_1, it follows that there is a simple path in $A(C')$ from p_j to a tree root, on which all particles crashed after p_1. Therefore, F2(b)iii holds for S in C'.

Initialization of crashed particle: Assume a CRASHED particle p of a segment calls `initializeAfterCrash` (`HexagonFT`, Line 5). After the initialization it holds $p.state$ = ERROR and $p.hasInvalidated[l]$ = FALSE for all port labels l. It is immediately apparent that all properties still hold. **State change (`HexagonFT`, Lines 11–21):**

Case 1: p becomes a ROOT or RETIRED: Consider F1: For a FOLLOWER q, there is a simple path in $A(C)$ to a tree root. If a particle p on the path becomes a ROOT or RETIRED, two cases are possible: the path is shortened or split into two paths, in both cases F1 still applies. For F2 consider that a particle p of a segment becomes a ROOT or RETIRED. This results in the segment no longer being a segment, and thus F2 trivially holds.

Case 2: p becomes a FOLLOWER by `tryToBecomeFollower`: It is obvious that F1 and F2 will still hold after p has become a FOLLOWER.

State change (`HexagonFT`, Lines 22–23): *p tries to become a* FOLLOWER *by* `tryFollowerRecoveryByPropagation`*:*

Case: if-branch: Let C' be the successor configuration of C after p became a FOLLOWER following q.

Case: q is not on a segment in $A(C)$: By F1, there is a simple path $P_q = (q = q_2, \dots, q_r)$, $r > 1$ in $A(C)$ from q to a tree root q_r. By the definition of a segment, all particles q_i for $2 \leq i \leq r$ must be non-faulty. Therefore, $P_p = (p = q_1, q = q_2, \dots, q_r)$ is a simple path from p to a tree root in $A(C')$; in particular, F1 holds for p in C'. *Consider F1*: Let $\tilde{p} \neq p$ be any FOLLOWER particle. By F1, there is a simple path $P_{\tilde{p}}$ in $A(C)$ from $\tilde{p}$ to a tree root. We only need to consider the case that p lies on $P_{\tilde{p}}$. Connecting the subpath of $P_{\tilde{p}}$ from $\tilde{p}$ to p with the path P_p results in a path in $A(C')$ from $\tilde{p}$ to a tree root. *Consider F2*: Let $p_1, \dots, p_k$ be a segment in $A(C')$ and observe that it is also a segment in $A(C)$. Particle p becoming a FOLLOWER could only have an effect on the validity of

F2(b)iii: By F2(b)iii, there is a simple path P_{p_j} in $A(C)$ from p_j to a tree root, on which all faulty particles crashed after p_1. We only need to consider the case that p lies on P_{p_j}. Connecting the subpath of P_{p_j} from p_j to p with the path P_p results in a path in $A(C')$ from p_j to a tree root, on which all faulty particles crashed after p_1.

Case: q is on a segment $p = q_1, q = q_2, \ldots, q_k$, $k > 1$ in $A(C)$: Note that q is a FOLLOWER parent candidate of p with $q.pathToRootState$ = VALID. By F2(b)iii, there is a simple path $P_q = (q = q_2, \ldots, q_r)$, $r \geq k$ in $A(C)$ from q to a tree root q_r, on which all faulty particles crashed after p. Therefore, $P_p = (p = q_1, q = q_2, \ldots, q_r)$ is a simple path from p to a tree root in $A(C')$; in particular, F1 holds for p in C'. *Consider F1*: Let $\tilde{p} \neq p$ be any FOLLOWER particle. By F1, there is a simple path $P_{\tilde{p}}$ in $A(C)$ from $\tilde{p}$ to a tree root. We only need to consider the case that p lies on $P_{\tilde{p}}$. Connecting the subpath of $P_{\tilde{p}}$ from $\tilde{p}$ to p with the path P_p results in a path in $A(C')$ from $\tilde{p}$ to a tree root.

Consider F2: Let $S' = (p_1, \ldots, p_{k'})$ be a segment in $A(C')$.

Case 1: p is not a particle on segment S': Particle p becoming a FOLLOWER could only have an effect on the validity of F2(b)iii: By F2(b)iii, there is a simple path P_{p_j} in $A(C)$ from p_j to a tree root, on which all faulty particles crashed after p_1. We only need to consider the case that p lies on P_{p_j}. Connecting the subpath of P_{p_j} from p_j to p with the path P_p results in a path in $A(C')$ from p_j to a tree root, on which all faulty particles crashed after p_1.

Case 2: p is a particle on segment S': We have $p = p_{k''}$ for some $1 < k'' < k'$. Note that $S = (p_1, \ldots, p_{k''})$ is a segment in $A(C)$. It is sufficient to show that in the case of F2b one of the three properties holds for S' in C'.

Case a: F2(b)i holds for S in $A(C)$: Clearly, F2(b)i holds for S' in $A(C')$.

Case b: F2(b)ii holds for S in $A(C)$: If $p_{k''}.pathToRootState$ = INVALIDATE, then F2(b)ii clearly holds for S' in $A(C')$. If $p_{k''}.pathToRootState \neq$ INVALIDATE and $p_{k''}$ crashed after p_1, then F2(b)iii holds for S' in $A(C')$.

Case c: F2(b)iii holds for S in $A(C)$: Clearly, F2(b)iii holds for S' in $A(C')$.

Case: else-branch: It is obvious that the properties will still hold in C'.

Propagation of *pathToRootState* (`HexagonFT`, Lines 25–30): Clearly, F1 remains true. Furthermore, one can easily check that F2 remains true. **Movement (`HexagonFT`, Line 34):** Clearly, F1 remains true. Only the first else-branch of `performMovement` may have an effect on F2. Let p be a contracted tail FOLLOWER of a FOLLOWER or ROOT particle q. Consider Line 7 of Algorithm 9: After p has pushed q, it occupies the node with its head, which q previously had occupied with its tail. The paths to a tree root that previously ran over the tail of q now run over the head of p. Therefore and since propagation has already taken place, p takes on the *pathToRootState* of q. Together with the fact that the branch is only executed if $q.pathToRootState \neq$ INVALIDATE, which ensures that INVALIDATE does not propagate downwards, this guarantees that F2 still holds after the movement. ◀

▶ **Lemma 5** (Connectivity). *At any time, all particles are connected, i.e., $G(C)$ is one connected component.*

Proof. In an initial configuration all particles are connected. Assume the property holds for a configuration C. The property could only be violated by a contraction of a particle, which only occurs in `performMovement` if p is an expanded FOLLOWER or ROOT that has no blocking tail neighbour (Line 3 of Algorithm 9). Let q be a neighbour of p before p contracts into its head and C' the configuration after p contracted. Since p has no blocking tail neighbour, we only need to consider the following cases:

Case 1: q is finished or a ROOT By 2 and the definition of a boundary particle, all finished and boundary particles are connected in C'. By F1, p is connected to a boundary particle in C'. Since q is finished or, by 3, a boundary particle, in C', p and q remain connected.
Case 2: q is a FOLLOWER of some particle other than p Similar to the previous case with the additional observation that, by F1, q is connected to a boundary particle.
Case 3: q is a head follower of p or an IDLE, ERROR or CRASHED particle adjacent to p's head Since p contracts into its head, p and q remain connected. ◀

▶ **Lemma 6.** *Every connected component of* IDLE *particles is connected to at least one non-*IDLE *particle.*

Proof. Consider a connected component H of IDLE particles. If H were not connected to a non-IDLE particle it would follow with 5 that all particles IDLE, contradicting the fact that there is always at least one non-idle particle, namely the SEED. ◀

The following properties are used in the proof of Theorem 1. As in its proof, we assume in the following that there is a time t^* from which on the configuration is legal.

▶ **Lemma 8.** *Eventually, every* IDLE *particle becomes a* FOLLOWER, ROOT *or* RETIRED *particle.*

Proof. Consider a configuration at time $t \geq t^*$. If there is an IDLE particle, then by Lemma 6 there exists an IDLE particle p that is connected to a non-IDLE particle. When p is activated, p becomes a FOLLOWER, ROOT or RETIRED particle. ◀

▶ **Lemma 9.** *Eventually, every expanded particle contracts.*

Proof. Consider a configuration at time $t \geq t^*$. By Lemma 8 we can assume that all particles are non-IDLE. Furthermore, we can assume that all FOLLOWER particles are in *pathToRootState* VALID. Let p be an expanded particle. If p has no tail FOLLOWER, then p can contract. If p has a contracted tail FOLLOWER or, p is a ROOT and has a contracted ROOT with *moveDir* set to p's tail, then p can contract by a handover. If p has an expanded tail FOLLOWER q, then we can apply this argument recursively for a finite number of steps to conclude that q eventually contracts and therefore also p. ◀

▶ **Theorem 1.** *If a finite number of crashes occur, then the algorithm* `HexagonFT` *solves the hexagon shape formation problem* **HEX** *in worst-case $\mathcal{O}(n^2)$ work (total number of moves executed by all particles). From the time when no more crashes occur and the configuration is non-faulty, the algorithm needs $\mathcal{O}(n)$ rounds until termination.*

Proof. By Lemma 7, there is a time from which on the configuration is non-faulty. From the safety properties in Section 3.3.1 the following statements can be derived:
1. Eventually, every IDLE particle becomes a FOLLOWER, ROOT or RETIRED particle. (Lemma 8)
2. Eventually, every expanded particle contracts. (Lemma 9): This implies in particular that as long as ROOT particles move, FOLLOWER particles will eventually follow.
3. A FOLLOWER that has a finished neighbour becomes a ROOT or RETIRED.
4. A ROOT will eventually become RETIRED or expand in direction *moveDir*.
5. All particles eventually become RETIRED.

The proof of these statements is essentially the same as for the analysis of the "classical" hexagon algorithm (cf. e.g. [7, 8]). Since all particles eventually become RETIRED and RETIRED particles do nothing, we eventually reach a stable configuration. Together with Lemma 2 this shows that `HexagonFT` solves **HEX**. ◀