Distributed Computing (2023) 36:159–192
https://doi.org/10.1007/s00446-023-00443-3

The canonical amoebot model: algorithms and concurrency control

Joshua J. Daymude[1,2] · **Andréa W. Richa**[1] · **Christian Scheideler**[3]

Received: 19 September 2022 / Accepted: 19 January 2023 / Published online: 17 February 2023

Abstract

The *amoebot model* abstracts active programmable matter as a collection of simple computational elements called *amoebots* that interact locally to collectively achieve tasks of coordination and movement. Since its introduction at SPAA 2014, a growing body of literature has adapted its assumptions for a variety of problems; however, without a standardized hierarchy of assumptions, precise systematic comparison of results under the amoebot model is difficult. We propose the *canonical amoebot model*, an updated formalization that distinguishes between core model features and families of assumption variants. A key improvement addressed by the canonical amoebot model is *concurrency*. Much of the existing literature implicitly assumes amoebot actions are isolated and reliable, reducing analysis to the sequential setting where at most one amoebot is active at a time. However, real programmable matter systems are concurrent. The canonical amoebot model formalizes all amoebot communication as message passing, leveraging adversarial activation models of concurrent executions. Under this granular treatment of time, we take two complementary approaches to *concurrent algorithm design*. We first establish a set of *sufficient conditions* for algorithm correctness under any concurrent execution, embedding concurrency control directly in algorithm design. We then present a *concurrency control framework* that uses locks to convert amoebot algorithms that terminate in the sequential setting and satisfy certain conventions into algorithms that exhibit equivalent behavior in the concurrent setting. As a case study, we demonstrate both approaches using a simple algorithm for *hexagon formation*. Together, the canonical amoebot model and these complementary approaches to concurrent algorithm design open new directions for distributed computing research on programmable matter.

Keywords Programmable matter · Self-organization · Distributed algorithms · Concurrency

1 Introduction

The vision of *programmable matter* is to realize a material that can dynamically alter its physical properties in a programmable fashion, controlled either by user input or its own autonomous sensing of its environment [46]. Towards a formal characterization of the minimum capabilities required by individual modules of programmable matter to achieve a given system behavior, many abstract models have been proposed over the last several decades [3,12–14,35,42,44,45,47]. We focus on the *amoebot model* [20,24] which is motivated by micro- and nano-scale robotic systems with strictly limited computational and locomotive capabilities [7,39–41,48,49]. The amoebot model abstracts active programmable matter as a collection of simple computational elements called *amoebots* that utilize local interactions to collectively achieve tasks involving coordination, movement, and reconfiguration. Since its introduction at SPAA 2014, the amoebot model has been used to study both fundamental problems—such as leader election [5,14,19,28,32–34,37,38] and shape formation [10,25,26,31,32,43]—as well as more complex behaviors including object coating [17,27], convex hull formation [18], bridging [2], spatial sorting [9], and fault tolerance [23,30].

With this growing body of amoebot model literature, it is evident that the model has evolved—and, to some extent,

✉ Joshua J. Daymude
jdaymude@asu.edu

Andréa W. Richa
aricha@asu.edu

Christian Scheideler
scheideler@upb.de

[1] School of Computing and Augmented Intelligence, Arizona State University, Tempe, AZ 85281, USA

[2] Biodesign Center for Biocomputing, Security and Society, Arizona State University, Tempe, AZ 85281, USA

[3] Department of Computer Science, Paderborn University, Paderborn, Germany

fractured—during its lifetime as assumptions were updated to support individual results, capture more realistic settings, or better align with other models of programmable matter. This makes it difficult to conduct any systematic comparison between results under the amoebot model (see, e.g., the overlapping but distinct features used for comparison of leader election algorithms in [5,34]), let alone between amoebot model results and those of related models (e.g., those from the established *autonomous mobile robots* literature [35]). To address the ways in which the amoebot model has outgrown its original rigid formulation, we propose the *canonical amoebot model* that includes a standardized, formal hierarchy of assumptions for its features to better facilitate comparison of its results. Moreover, such standardization will more gracefully support future model generalizations by distinguishing between core features and assumption variants.

A key area of improvement addressed by the canonical amoebot model is *concurrency*. The original model treats concurrency at a high level, implicitly assuming an isolation property that prohibits concurrent amoebot actions from interfering with each other. Furthermore, amoebots are usually assumed to be *reliable*; i.e., they cannot crash or exhibit Byzantine behavior. Under these simplifying assumptions, most existing algorithms are analyzed for correctness and runtime as if they are executed *sequentially*, with at most one amoebot acting at a time. Notable exceptions include the recent work of Di Luna et al. [30–32] and Nokhanji and Santoro [43] that adopt ideas from the "look-compute-move" paradigm used in autonomous mobile robots to bring the amoebot model closer to a realistic, concurrent setting. Our canonical amoebot model furthers these efforts by formalizing all communication and cooperation between amoebots as message passing while also addressing the complexity of potential conflicts caused by amoebot movements. This careful formalization allows us to use standard adversarial activation models from the distributed computing literature to describe concurrency [1].

This fine-grained treatment of concurrency in the canonical amoebot model lays the foundation for the design and analysis of *concurrent amoebot algorithms*. Concurrency adds significant design complexity, allowing concurrent amoebot actions to mutually interfere, conflict, affect outcomes, or fail in ways far beyond what is possible in the sequential setting. As a tool for controlling concurrency, we introduce a Lock operation in the canonical amoebot model enabling amoebots to attempt to gain exclusive access to their neighborhood.

We then take two complementary approaches to concurrent amoebot algorithm design: a direct approach that embeds concurrency control directly into the algorithm's design without requiring locks, and an indirect approach that relies on the Lock operation to mitigate issues of concurrency. In the first approach, we establish a set of *general sufficient conditions* for amoebot algorithm correctness under any adversary—sequential or asynchronous, fair or unfair—using the *hexagon formation problem* (see, e.g., [20,25]) as a case study. Our Hexagon-Formation algorithm demonstrates that locks are not necessary for correctness even under an unfair, asynchronous adversary. However, this algorithm's asynchronous correctness relies critically on its actions succeeding despite any concurrent action executions, which may be a difficult property to obtain in general.

For our second approach, we present a *concurrency control framework* using the Lock operation that, given an amoebot algorithm that terminates under any sequential execution and satisfies some basic conventions, produces an algorithm that exhibits equivalent behavior under any asynchronous execution. This framework establishes a general design paradigm for concurrent amoebot algorithms: one can first design an algorithm with correct behavior in the simpler sequential setting and then, by ensuring it satisfies our framework's conventions, automatically obtain a correct algorithm for the asynchronous setting. The convenience of this approach comes at the cost of limiting the full generality of the canonical amoebot model to comply with the framework's conventions. Nevertheless, we prove that the Hexagon-Formation algorithm satisfies these conventions and thus is compatible with the framework.

Our Contributions. We summarize our contributions as follows.

- The *canonical amoebot model*, an updated formalization that treats amoebot actions at the fine-grained level of message passing and distinguishes between core model features and hierarchies of assumption variants (Sect. 2).
- General *sufficient conditions* for amoebot algorithm correctness under any adversary and an algorithm for *hexagon formation* that satisfies these conditions (Sect. 3).
- A *concurrency control framework* that converts amoebot algorithms that terminate under any sequential execution and satisfy certain conventions into algorithms that exhibit equivalent behavior under any asynchronous execution (Sect. 4), and an application of this framework to the algorithm for hexagon formation (Sect. 4.1).

Relationship to prior versions. This work improves over its conference version published at DISC 2021 [21] in several aspects. First, this work contains all details and proofs that were omitted due to conference space constraints, including the message passing implementations of amoebot operations. Second, whereas the original publication treated the newly added Lock and Unlock operations as black boxes, this work suggests a possible implementation based on the recent algorithm for local mutual exclusion in dynamic

networks [22]. Third, this work improves the usability of the concurrency control framework. Of the three algorithm conventions required for compatibility with the framework in [21], the most difficult to understand and verify is *monotonicity*. In fact, it was not known whether any amoebot algorithm involving movement could satisfy monotonicity, posing a serious limitation to the framework's use. This work replaces monotonicity with a more general and more easily-understood convention, *expansion-robustness*, without changing the framework's guarantees. Finally, this work proves that the algorithm for hexagon formation (Sect. 3) is expansion-robust, thus identifying the first algorithm involving movement that is compatible with the concurrency control framework and resolving the previously open question.

1.1 Related work

There are many theoretical models of programmable matter, ranging from the non-spatial *population protocols* [3] and *network constructors* [42] to the tile-based models of *DNA computing* and *molecular self-assembly* [12,44,47]. Most closely related to the amoebot model studied in this work is the well-established literature on *autonomous mobile robots*, and in particular those using discrete, graph-based models of space (see Chapter 1 of [35] for a recent overview). Both models assume anonymous individuals that can actively move, lacking a global coordinate system or common orientation, and having strictly limited computational and sensing capabilities. In addition, stronger capabilities assumed by the amoebot model also appear in more recent variants of mobile robots, such as persistent memory in the $\mathcal{F}$*-state* model [4,36] and limited communication capabilities in *luminous robots* [15,16,29].

There are also key differences between the amoebot model and the standard assumptions for mobile robots, particularly around their treatment of physical space, the structure of individuals' actions, and concurrency. First, while the discrete-space mobile robots literature abstractly envisions robots as agents occupying nodes of a graph—allowing multiple robots to occupy the same node—the amoebot model assumes *physical exclusion* that ensures each node is occupied by at most one amoebot at a time, inspired by the real constraints of self-organizing micro-robots and colloidal state machines [7,39–41,48,49]. Physical exclusion introduces conflicts of movement (e.g., two amoebots concurrently moving into the same space) that must be handled carefully in algorithm design.

Second, mobile robots are assumed to operate in *look-compute-move* cycles, where they take an instantaneous snapshot of their surroundings (look), perform internal computation based on the snapshot (compute), and finally move to a neighboring node determined in the compute stage (move). While it is reasonable to assume robots may instantaneously snapshot their surroundings due to all information being visible, the amoebot model—and especially the canonical version presented in this work—treats all inter-amoebot communication as asynchronous message passing, making snapshots nontrivial. Moreover, amoebots have *read and write* operations allowing them to access or update variables stored in the persistent memories of their neighbors that do not fit cleanly within the look-compute-move paradigm.

Finally, the mobile robots literature has a well-established and carefully studied hierarchy of *adversarial schedulers* capturing assumptions on concurrency that the amoebot model has historically lacked. In fact, other than notable recent works that adapt look-compute-move cycles and a semi-synchronous scheduler from mobile robots to the amoebot model [30–32,43], most amoebot literature assumes only sequential activations. A key contribution of our canonical amoebot model presented in this work is a hierarchy of concurrency and fairness assumptions similar in spirit to that of mobile robots, though our underlying message passing design and lack of explicit action structure require different formalizations.

2 The canonical amoebot model

We introduce the *canonical amoebot model* as an update to the model's original formulation [20,24]. This update has two main goals. First, we model all amoebot actions and operations using message passing, leveraging this finer level of granularity for a formal treatment of concurrency. Second, we clearly delineate which assumptions are fixed features of the model and which have stronger and weaker variants, providing unifying terminology for future amoebot model research. Unless variants are explicitly listed, the following description of the canonical amoebot model details its core, fixed assumptions. The variants are summarized in Table 1; we anticipate that this list will grow as future research develops new adaptations and generalizations of the model.

In the canonical amoebot model, programmable matter consists of individual, homogeneous computational elements called *amoebots*. The structure of an amoebot system is represented as a subgraph of an infinite, undirected graph $G = (V, E)$ where V represents all relative positions an amoebot can occupy and E represents all atomic movements an amoebot can make. Each node in V can be occupied by at most one amoebot at a time. There are many possible representations of space; previous amoebot literature most commonly assumes the *geometric* variant where $G = G_\Delta$, the triangular lattice (Fig. 1a).

An amoebot has two *shapes*: CONTRACTED, meaning it occupies a single node in V, or EXPANDED, meaning it occupies a pair of adjacent nodes in V (Fig. 1b). For a contracted amoebot, the unique node it occupies is considered its *head*; for an expanded amoebot, the node it has most recently come

Table 1 Summary of assumption variants in the canonical amoebot model, each organized from most to least general

	Variant	Description
Space	General*	G is any infinite, undirected graph
	Geometric*,†	$G = G_\Delta$, the triangular lattice
Orientation	Assorted*,†	Assorted direction and chirality
	Common Chirality*	Assorted direction but common chirality
	Common Direction	Common direction but assorted chirality
	Common	Common direction and chirality
Memory	Oblivious	No persistent memory
	Constant-Size*,†	Memory size is $\mathcal{O}(1)$
	Finite	Memory size is $\mathcal{O}(f(n))$, some function of the system size
	Unbounded	Memory size is unbounded
Concurrency	Asynchronous†	Any amoebots can be simultaneously active
	Synchronous*	Any amoebots can simultaneously execute a single action per discrete step. Each step has an evaluation phase and an execution phase
	k-Isolated	No amoebots within hop distance k can be simultaneously active
	Sequential*	At most one amoebot is active per time
Fairness	Unfair†	Some enabled amoebot is eventually activated
	Weakly Fair*	Every continuously enabled amoebot is eventually activated
	Strongly Fair	Every amoebot enabled infinitely often is activated infinitely often

Variants marked with ∗ have been considered in existing literature, and variants marked with † are the focus of the algorithmic results in this work

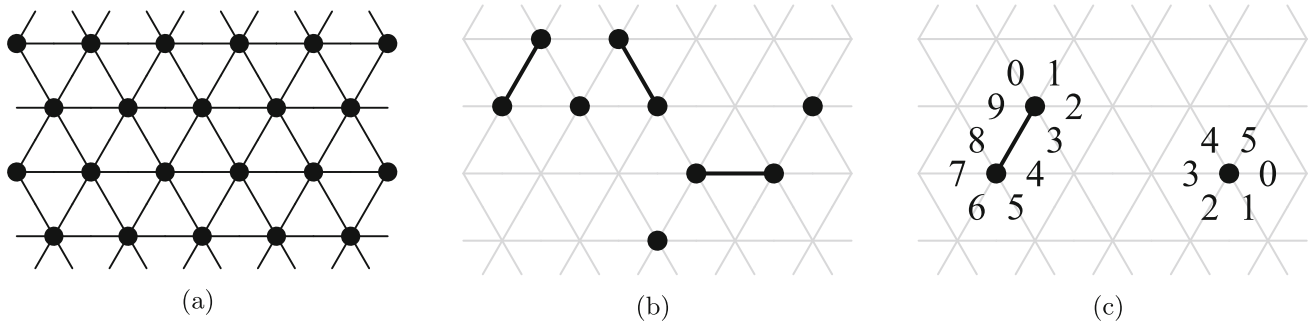


Fig. 1 The Canonical Amoebot Model. **a** A section of the triangular lattice G_Δ used in the geometric variant; nodes of V are shown as black circles and edges of E are shown as black lines. **b** Expanded and contracted amoebots; G_Δ is shown in gray, and amoebots are shown as black circles. Amoebots with a black line between their nodes are expanded. **c** Two amoebots that agree on their chirality but not on their direction, using different offsets for their clockwise-increasing port labels

to occupy (due to movement) is considered its head and the other is its *tail*. Each amoebot keeps a collection of ports—one for each edge incident to the node(s) it occupies—that are labeled consecutively according to its own local, persistent *orientation*. For any space variant where G is a planar graph (i.e., those that can be thought of as "two-dimensional"), an amoebot's orientation depends on its *direction*—i.e., which incident edge it perceives as "north"—and its *chirality*, or sense of clockwise and counter-clockwise rotation. Different variants may assume that amoebots share one, both, or neither of their directions and chiralities in common (see Table 1), Fig. 1c gives an example of the *common chirality* variant where amoebots share a sense of clockwise rotation but have different directions.

Two amoebots occupying adjacent nodes are said to be *neighbors*. Although each amoebot is *anonymous*, lacking a unique identifier, we assume an amoebot can locally identify its neighbors using their port labels. In particular, we assume that amoebots A and B connected via ports p_A and p_B each know one another's orientations and labels for p_A and p_B. If A is expanded, we also assume B knows the direction A is expanded in with respect to its own local direction, and vice versa. This is sufficient for an amoebot to reconstruct which adjacent nodes are occupied by the same neighbor and to translate its local orientation into those of its neighbors, but is not so strong so as to collapse the hierarchy of orientation assumptions. More details on an amoebot's anatomy are given in Sect. 2.1.

Table 2 Summary of operations exposed by an amoebot's system layer to its application layer

Operation	Return value on success
CONNECTED(p)	TRUE iff a neighboring amoebot is connected via port p
CONNECTED()	$[c_0, \ldots, c_{k-1}] \in \{N_1, \ldots, N_k, \text{FALSE}\}^k$ where $c_p = N_i$ if N_i is the locally identified neighbor connected via port p and $c_p = \text{FALSE}$ otherwise
READ(p, x)	The value of x in the public memory of this amoebot if $p = \perp$ or of the neighbor incident to port p otherwise
WRITE(p, x, x_{val})	Confirmation that the value of x was updated to x_{val} in the public memory of this amoebot if $p = \perp$ or of the neighbor incident to port p otherwise
CONTRACT(v)	Confirmation of the contraction out of node $v \in \{\text{HEAD}, \text{TAIL}\}$
EXPAND(p)	Confirmation of the expansion into the node incident to port p
PULL(p)	Confirmation of the pull handover with the neighbor incident to port p
PUSH(p)	Confirmation of the push handover with the neighbor incident to port p
LOCK()	Port labels corresponding to the amoebots that were locked
UNLOCK($\mathcal{L}$)	Confirmation that the amoebots of $\mathcal{L}$ were unlocked

An amoebot's functionality is partitioned between a higher-level *application layer* and a lower-level *system layer*. Algorithms controlling an amoebot's behavior are designed from the perspective of the application layer. The system layer is responsible for an amoebot's core functions and exposes a limited programming interface of *operations* to the application layer that can be used in amoebot algorithms. The operations are defined in Sect. 2.2 and their organization into algorithms is described in Sect. 2.3. Throughout, we assume amoebots execute their algorithms *reliably*, without crash or Byzantine faults.[1] Although theoretical models usually abstract away from a system layer, we describe it in detail to justify the interface to the application layer since amoebots are not a standard computing platform. In future publications, one may abstract from the system layer and focus only on the interface.

2.1 Amoebot anatomy

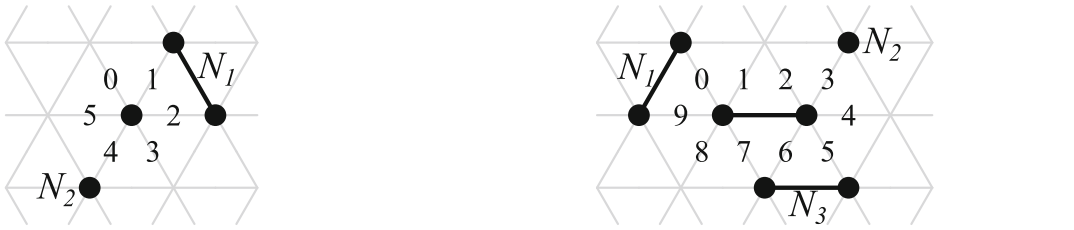
Each amoebot has memory whose size is a model variant; the standard assumption is *constant-size* memory. An amoebot's memory consists of two parts: a persistent *public memory* that is read-writeable by the system layer but only accessible to the application layer via communication operations (see Sect. 2.2.1), and a volatile *private memory* that is inaccessible to the system layer but read-writable by the application layer. The public memory of an amoebot A contains (i) the shape of A, denoted $A.\texttt{shape} \in \{\text{CONTRACTED}, \text{EXPANDED}\}$, (ii) the lock state of A, denoted $A.\texttt{lock}$ (see Sect. 2.2.3), and (iii) any variables used in the algorithm being run by the application layer. An amoebot's private memory can be modified by the application layer as needed.

Neighboring amoebots (i.e., those occupying adjacent nodes) form *connections* via their ports facing each other. An amoebot's system layer receives instantaneous feedback whenever a new connection is formed or an existing connection is broken. Communication between connected neighbors is achieved via *message passing*. To facilitate message passing communication, each of an amoebot's ports has a FIFO *outgoing message buffer* managed by the system layer that can store up to a fixed (constant) number of messages waiting to be sent to the neighbor incident to the corresponding port. If two neighbors disconnect due to some movement, their system layers immediately flush the corresponding message buffers of any pending messages. Otherwise, we assume that any pending message is sent to the connected neighbor in FIFO order in finite time. Incoming messages are processed as they are received.

2.2 Amoebot operations

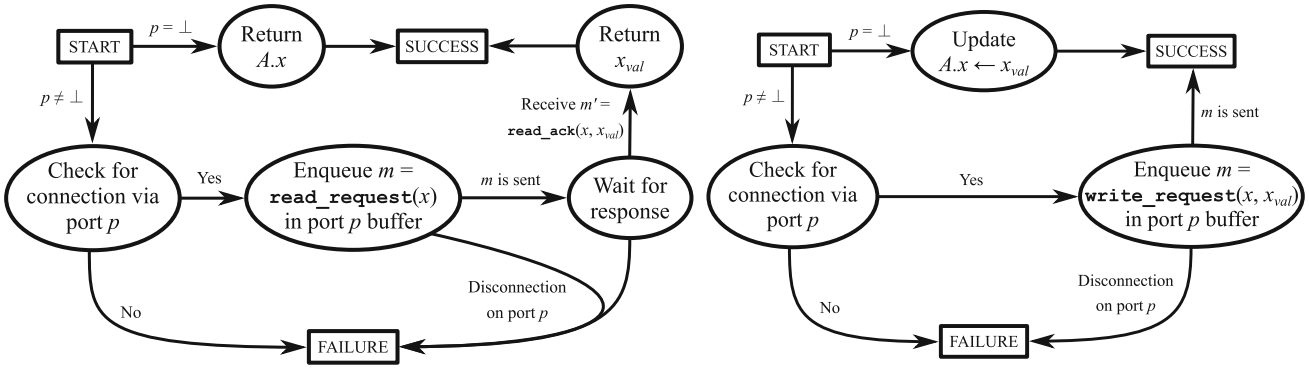
Operations provide the application layer with a programming interface for controlling the amoebot's behavior; the application layer calls operations and the system layer executes them. We assume the execution of an operation is *blocking* for the application layer; that is, the application layer can only call one operation at a time. We formally define the communication, movement, and concurrency control operations and their execution details in Sects. 2.2.1–2.2.3; see Table 2 for a summary and Appendix A for complete distributed pseudocode. As we will show in Sect. 2.2.4, each operation is carefully designed so that any operation execution terminates in finite time (Observation 1) and, at any time, there are at most a constant number of messages being sent or received between any pair of neighboring amoebots as a result of any set of concurrent operation executions (Observation 2). Com-

[1] As we discuss in Sect. 5, designing *fault tolerant* algorithms is an important research direction for programmable matter. We leave the formalization of different fault models under the canonical amoebot model for future work.



(a) $[\text{FALSE}, N_1, N_1, \text{FALSE}, N_2, \text{FALSE}]$ (b) $[N_1, \text{FALSE}, \text{FALSE}, N_2, \text{FALSE}, N_3, N_3, N_3, \text{FALSE}, N_1]$

Fig. 2 Different neighborhood configurations for an amoebot A and their corresponding CONNECTED() return values. The ports of A are shown with their labels and neighboring amoebots are shown with their local identifiers according to A



(a) READ(p, x) (b) WRITE(p, x, x_{val})

Fig. 3 Execution flows of the READ and WRITE operations for the calling amoebot A

bined with the blocking and reliability assumptions, these design principles prohibit outgoing message buffer overflow and deadlocks in operation executions.

2.2.1 Communication operations

An amoebot checks for the presence of neighbors using the CONNECTED operations and exchanges information with its neighbors using the READ and WRITE operations. When the application layer calls CONNECTED(p), the system layer simply returns TRUE if there is a neighbor connected via port p and FALSE otherwise. The application layer may instead call CONNECTED() to obtain a full snapshot of its current port connectivity. Specifically, the system layer returns an array $[c_0, \ldots, c_{k-1}]$ mapping the amoebot's k ports to local identifiers for its neighbors, of which there can be at most k. If there is no neighbor connected via port p, then $c_p = \text{FALSE}$; otherwise, $c_p = N_i$ where $N_i \in \{N_1, \ldots, N_k\}$ locally identifies the neighbor connected via port p (see Fig. 2). Note that, depending on amoebots' shapes and the geometry of the space variant, multiple ports may connect to the same neighbor N_i.

The application layer calls READ(p, x) to issue a request to read the value of a variable x in the public memory of the neighbor connected via port p. Analogously, the application layer calls WRITE(p, x, x_{val}) to issue a request to update the value of a variable x in the public memory of the neighbor connected via port p to a new value x_{val}. If $p = \perp$, an amoebot's own public memory is accessed instead of a neighbor's.

Suppose that the application layer of an amoebot A calls READ(p, x), illustrated in Fig. 3a. If $p = \perp$, the system layer simply returns the value of x in the public memory of A to the application layer and this READ succeeds. Otherwise, the system layer checks if there is a neighbor connected via port p: if so, the system layer enqueues $m = \texttt{read\_request}(x)$ in the message buffer on p; otherwise, this READ fails. Let B be the neighbor connected to A via port p and let p' be its corresponding port. Eventually, m is sent in FIFO order and the system layer of B receives it, prompting it to access variable x with value x_{val} in its public memory and enqueue $m' = \texttt{read\_ack}(x, x_{val})$ in the message buffer on p'. Message m' is eventually sent in FIFO order by B and received by the system layer of A, prompting it to unpack x_{val} and return it to the application layer, successfully completing this READ. If A and B are disconnected (i.e., due to a movement) any time after A enqueues message m but before A receives message m', this READ fails.

A WRITE(p, x, x_{val}) operation is executed analogously, though it does not need to wait for an acknowledgement after its write request is sent (see Fig. 3b).

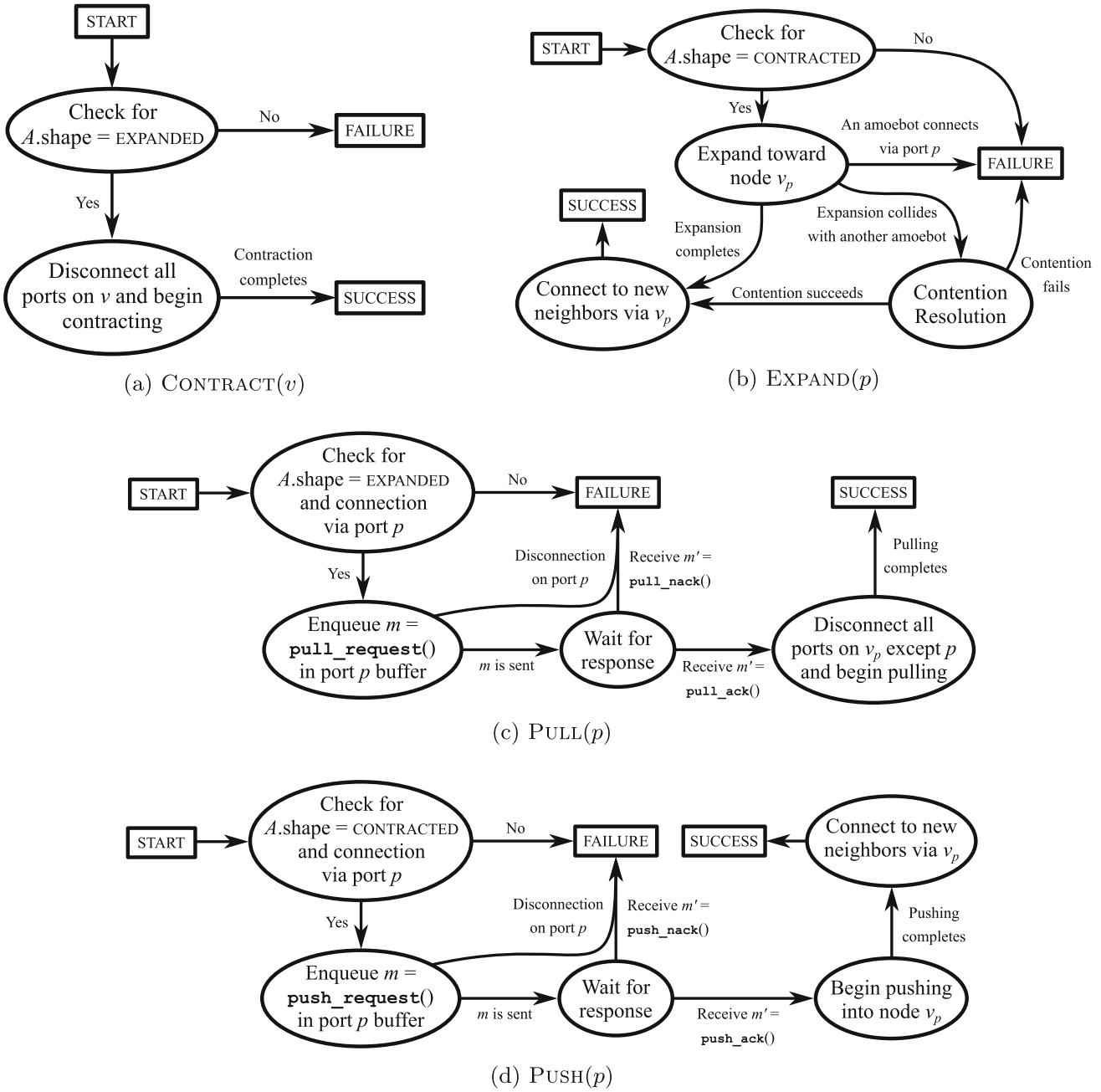


Fig. 4 Execution flows of the movement operations for the calling amoebot A

2.2.2 Movement operations

The application layer can direct the system layer to initiate movements using the four movement operations CONTRACT, EXPAND, PULL, and PUSH. An expanded amoebot can CONTRACT into either node it occupies; a contracted amoebot can EXPAND into an unoccupied adjacent node. Neighboring amoebots can coordinate their movements in a *handover*, which can occur in one of two ways. A contracted amoebot A can PUSH an expanded neighbor B by expanding into a node occupied by B, forcing it to contract. Alternatively, an expanded amoebot B can PULL a contracted neighbor A by contracting, forcing A to expand into the node it is vacating.

Contract. Suppose that the application layer of an amoebot A calls CONTRACT(v), where $v \in \{\text{HEAD}, \text{TAIL}\}$ (see Fig. 4a). The system layer of A first determines if this contraction is valid: if $A.\texttt{shape} \neq \text{EXPANDED}$ or A is currently involved in a handover, this CONTRACT fails. Otherwise, the system layer releases all connections to neighboring amoe-

bots via ports on node v and begins contracting out of node v. Once the contraction completes, the system layer updates $A.\texttt{shape} \leftarrow$ CONTRACTED, successfully completing this CONTRACT.

Expand. Suppose an amoebot A calls EXPAND(p) for one of its ports p (see Fig. 4b); let v_p denote the node A is expanding into. If $A.\texttt{shape} \neq$ CONTRACTED, A is already involved in a handover, or v_p is already occupied by another amoebot, this EXPAND fails. Otherwise, A begins its expansion into node v_p. Once this expansion completes, the system layer establishes connections with all neighbors adjacent to v_p and updates $A.\texttt{shape} \leftarrow$ EXPANDED, successfully completing this EXPAND. However, A may *collide* with other amoebots while expanding into v_p. We assume that the system layer can detect when a collision has occurred and, on collision, performs *contention resolution* such that exactly one contending amoebot succeeds in completing its expansion into v_p while all others fail within finite time. We abstract away from the details of this contention resolution mechanism for the sake of clarity, but give one possible implementation in Appendix B to demonstrate its feasibility.

Pull and Push. Suppose an amoebot A calls PULL(p) for one of its ports p (see Fig. 4c); let v_p denote the node A intends to vacate in this pull handover. If $A.\texttt{shape} \neq$ EXPANDED, A is already involved in a handover, or A is not connected to a neighbor via port p, this PULL fails. Otherwise, the system layer of A enqueues $m = \texttt{pull\_request}()$ in the message buffer on port p. Let B be the neighbor connected to A via port p. Eventually, message m is sent in FIFO order and the system layer of B receives it. If B is not involved in another movement and $B.\texttt{shape} =$ CONTRACTED, its system layer prepares message $m' = \texttt{pull\_ack}()$; otherwise, it sets $m' = \texttt{pull\_nack}()$. In either case, the system layer of B enqueues m' in the message buffer on its port facing A. If A and B are disconnected any time after A enqueues message m but before A receives message m', this PULL fails; otherwise, message m' is eventually sent in FIFO order by B and received by the system layer of A. If $m' = \texttt{pull\_nack}()$, this PULL fails. Otherwise, if $m' = \texttt{pull\_ack}()$, A disconnects from all ports on node v_p (except for p) and A and B begin their coordinated handover of node v_p. When A completes its contraction, it updates $A.\texttt{shape} \leftarrow$ CONTRACTED; analogously, when B completes its expansion, it updates $B.\texttt{shape} \leftarrow$ EXPANDED and establishes connections to its new neighbors adjacent to node v_p. This successfully completes this PULL.

A PUSH(p) operation is executed analogously (see Fig. 4d).

2.2.3 Concurrency control operations

The amoebot model's concurrency control operations LOCK and UNLOCK encapsulate a variant of the classical mutual exclusion problem in which an amoebot attempts to gain exclusive control over itself and the amoebots in its neighborhood. Achieving this behavior in the system layer's setting of asynchronous message passing with dynamic neighbor connections is non-trivial. Daymude et al. recently solved this problem in their algorithm for "local mutual exclusion" [22] where nodes in a dynamic graph seek to acquire exclusive locks over themselves and their "persistent" neighbors, i.e., nodes that remain connected to them over the time interval of the lock request. Here, we focus on the properties that LOCK and UNLOCK must satisfy and refer the interested reader to [22] for one possible implementation.

Each amoebot A stores a variable $A.\texttt{lock} \in \{\bot, -1, \ldots, \Delta - 1\}$, where Δ is the maximum number of neighbors an amoebot can have based on the assumed space variant, that is equal to $\bot$ if A is unlocked, -1 if A has locked itself, and $i \in \{0, \ldots, \Delta - 1\}$ if A is locked by its neighbor connected via port i. An amoebot A calls LOCK() to issue a lock request to itself and the neighbors it has at the start of this execution. To succeed, this LOCK operation must lock A and every *persistent* neighbor of A that remained connected to A throughout its LOCK execution, setting their `lock` variables accordingly. On success, the LOCK operation returns the *lock set* $\mathcal{L}$ of port labels corresponding to the amoebots A has locked. We assume that a LOCK operation either succeeds or fails in finite time. An amoebot calls UNLOCK($\mathcal{L}'$) to release its locks on itself or any neighbors connected via port labels in $\mathcal{L}'$, resetting their `lock` variables to $\bot$; this operation always succeeds.

Any implementation of these operations must ensure that any set of LOCK and UNLOCK executions satisfies: (i) *mutual exclusion*, meaning that the amoebots' lock sets must be disjoint at all times, and (ii) *deadlock freedom*, meaning that if a LOCK operation is initiated at time t, then some LOCK execution succeeds after time t. The local mutual exclusion algorithm of [22] satisfies both of these properties; in fact, it even satisfies the stronger property of *lockout freedom*, guaranteeing that every LOCK execution eventually succeeds.

2.2.4 Operation time and space complexity

With the communication, movement, and concurrency control operations defined, we now briefly characterize their time and space complexity. Recall that we assume amoebots execute reliably, without crash or Byzantine faults. The CONNECTED operations are effectively instantaneous as the system layer has immediate access to the physical information about its port connectivity; moreover, these operations do not involve any messages. The complexity of the LOCK and UNLOCK operations depend on their implementation; e.g., the local mutual exclusion algorithm of [22] guarantees termination in finite time and that at most two messages are in transit between any pair of neighbors at any time. For the

remaining operations, recall from Sects. 2.1 and 2.2.2 that we assume (i) messages pending in an outgoing message buffer are each sent to the connected neighbor in FIFO order in finite time and are immediately flushed on disconnection, and (ii) every physical movement completes in finite time. We first consider the execution of each operation independently.

- A READ operation by an amoebot A from its own public memory is immediate and does not involve any messages. If instead A reads from the public memory of a neighbor B, at most two messages are used—a `read_request` sent from A to B followed by a `read_ack` sent from B to A—that are each delivered in finite time by (i). Successful termination occurs when A receives `read_ack` while a disconnection between A and B results in immediate failure with any related messages being flushed.
- A WRITE operation by an amoebot A to its own public memory is immediate and does not involve any messages. If instead A writes to the public memory of a neighbor B, one message is used: a `write_request` sent from A to B that is delivered in finite time by (i). Successful termination occurs when A sends `write_request` while a disconnection between A and B results in immediate failure with any related messages being flushed.
- A CONTRACT operation does not involve any messages. It either fails at its start or succeeds after releasing its connections and completing its contraction, which must occur in finite time by (ii).
- An EXPAND operation does not involve any messages. It either fails at its start or is able to begin its expansion. If there are no collisions, (ii) guarantees the expansion completes in finite time; otherwise, the contention resolution mechanism guarantees that exactly one contending amoebot succeeds while all others fail within finite time.
- A PULL operation by an amoebot A with a neighbor B involves at most two messages—a `push_request` sent from A to B and either a `push_ack` or a `push_nack` sent from B to A—that are delivered in finite time by (i). The contraction of A and expansion of B must complete in finite time by (ii). Any failures can only happen earlier.
- A PUSH operation is symmetric to a PULL and thus satisfies the same properties.

This immediately reveals the following observation regarding time complexity.

Observation 1 *Any execution of an operation in the canonical amoebot model terminates—either successfully or in failure—in finite time.*

By the blocking assumption, the application layer of each amoebot can execute at most one operation per time. The above discussion shows that each operation has at most one message in transit per time. Thus, there can be at most two messages in any outgoing message buffer at any time, e.g., in the situation where an amoebot A is executing an operation that involves sending a message m_1 to a neighbor B while B is concurrently executing an operation that requires A to send a message m_2 back to B in response to some prior message sent from B. This yields the following observation, demonstrating that constant-size buffers suffice to avoid overflow.

Observation 2 *At any time, there are at most a constant number of messages in transit (i.e., being sent or received) between any pair of neighboring amoebots as a result of any set of operation executions.*

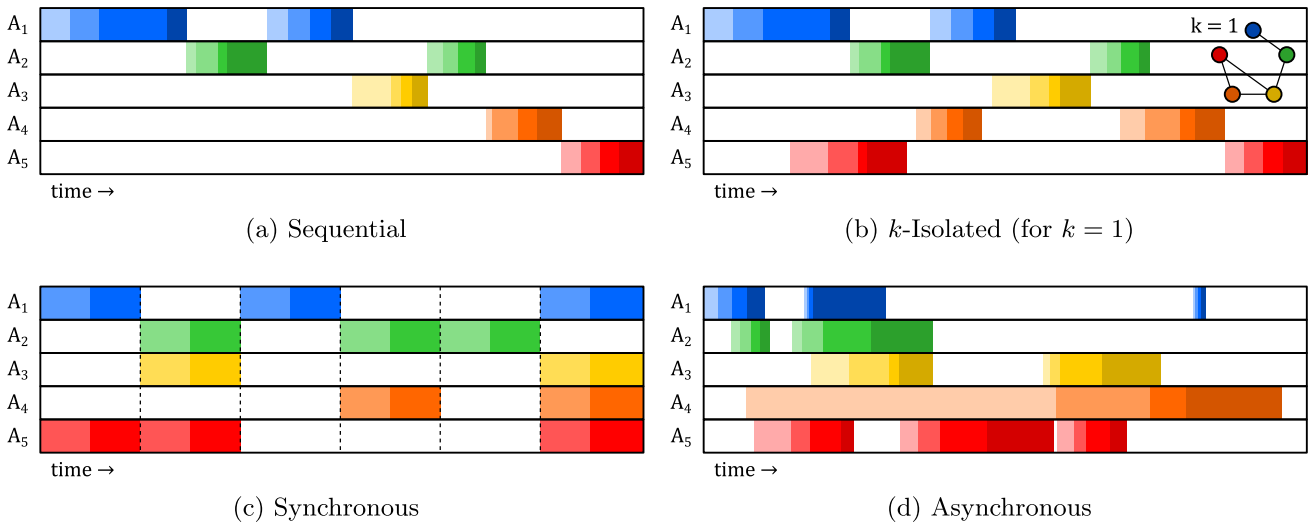
2.3 Amoebot actions, algorithms and executions

Following the message passing literature, we specify distributed algorithms in the amoebot model as sets of *actions* to be executed by the application layer, each of the form:

$$\langle label \rangle : \langle guard \rangle \rightarrow \langle operations \rangle$$

An action's *label* specifies its name. Its *guard* is a Boolean predicate determining whether an amoebot A can execute it based on the connected ports of A—i.e., which nodes adjacent to A are (un)occupied—and information from the public memories of A and its neighbors. An action is *enabled* for an amoebot A if its guard is true for A, and an amoebot is *enabled* if it has at least one enabled action. An action's *operations* specify the finite sequence of operations and computation in private memory to perform if this action is executed. The control flow of this computation may optionally include *randomization* to generate random values and *error handling* to address any operation executions resulting in failure.

Each amoebot executes its own algorithm instance independently and reliably, without crash or Byzantine faults. An amoebot is said to be *active* if its application layer is executing an action and is *inactive* otherwise. An amoebot can begin executing an action if and only if it is inactive; i.e., an amoebot can execute at most one action at a time. On becoming active, an amoebot A first evaluates which of its actions $\alpha_i : g_i \rightarrow ops_i$ are enabled. Since each guard g_i is based only on the connected ports of A and the public memories of A and its neighbors, each g_i can be evaluated using the CONNECTED and READ operations. If no action is enabled, A returns to inactive; otherwise, A chooses an enabled action α_i and executes the operations and private computation specified by ops_i. Recall from Sect. 2.2 that each operation is guaranteed to terminate (either successfully or with a failure) in finite time. Thus, since A is reliable and ops_i consists of a finite sequence of operations and finite computation, each action execution is also guaranteed to terminate in finite time after which A returns to inactive. An action execution *fails* if any of its operations' executions result in a failure that is not addressed with error handling and *succeeds* otherwise.



(a) Sequential (b) k-Isolated (for $k = 1$)

(c) Synchronous (d) Asynchronous

Fig. 5 Adversary concurrency variants. Amoebots are shown in rows and their actions over time are shown as colored boxes, subdivided into operations (gradient of colors)

As is standard in the distributed computing literature (see, e.g., [1]), we assume an *adversary* (or *daemon*) controls the timing of amoebot activations, the choice of enabled actions to execute, and the timing of action executions. The power of an adversary is determined by its *concurrency* and *fairness*. We distinguish between four concurrency variants: *sequential*, in which at most one amoebot can be active at a time (Fig. 5a); *k-isolated*, in which no two amoebots occupying nodes of G within hop distance k can be simultaneously active, but any others can (Fig. 5b); *synchronous*, in which time is discretized into "steps" and in each step any set of amoebots can simultaneously execute one action each (Fig. 5c); and *asynchronous*, in which any set of amoebots can be simultaneously active (Fig. 5d). For synchronous concurrency, we further assume that each step is partitioned into an *evaluation phase* when all active amoebots evaluate their guards followed by an *execution phase* when all active amoebots with enabled actions execute the corresponding operations. Fairness restricts how often the adversary must activate enabled amoebots. We distinguish between three fairness variants: *strongly fair*, in which every amoebot that is enabled infinitely often is activated infinitely often; *weakly fair*, in which every continuously enabled amoebot is eventually activated; and *unfair*, in which the adversary may activate any enabled amoebot. An algorithm execution is said to *terminate* if eventually all amoebots are inactive and disabled; note that since an amoebot can only become enabled based on some other amoebot's action, termination is permanent.

We evaluate an amoebot algorithm's time complexity in terms of *rounds*, which informally represent the time for the slowest continuously enabled amoebot to execute a single action. Let t_i denote the time at which round $i \in \{0, 1, 2, \ldots\}$ starts, where $t_0 = 0$, and let $\mathcal{E}_i$ denote the set of amoebots that are enabled or already executing an action at time t_i. Round i completes at the earliest time $t_{i+1} > t_i$ by which every amoebot in $\mathcal{E}_i$ either completed an action execution or became disabled at some time in $(t_i, t_{i+1}]$. Depending on the adversary's concurrency, action executions may span more than one round.

In this paper, we focus on unfair sequential and asynchronous adversaries. In the sequential setting, there is at most one active amoebot per time; thus, its guard evaluations and subsequent operation executions must be correct. In the asynchronous setting, however, concurrent movements and memory updates can cause discrepancies between the adversary's instantaneous view of enabled actions and an amoebot's real-time evaluation of its guards, potentially allowing disabled actions to be executed or enabled actions to be skipped. Moreover, concurrency can cause operations to fail due to conflicts. We address these issues in two ways, justifying the formulation of algorithms in terms of actions: In Sect. 3, we present an algorithm whose actions are carefully designed to ensure correct execution under any adversary; in Sect. 4, we present a concurrency control framework that uses locks to ensure correct guard evaluation and operation execution even in the asynchronous setting.

3 Asynchronous hexagon formation without locks

We use the *hexagon formation* problem as a concrete case study for algorithm design, pseudocode, and analysis in the canonical amoebot model. Our Hexagon-Formation algorithm (Algorithm 1) assumes geometric space, assorted orientation, and constant-size memory (Table 1) and is for-

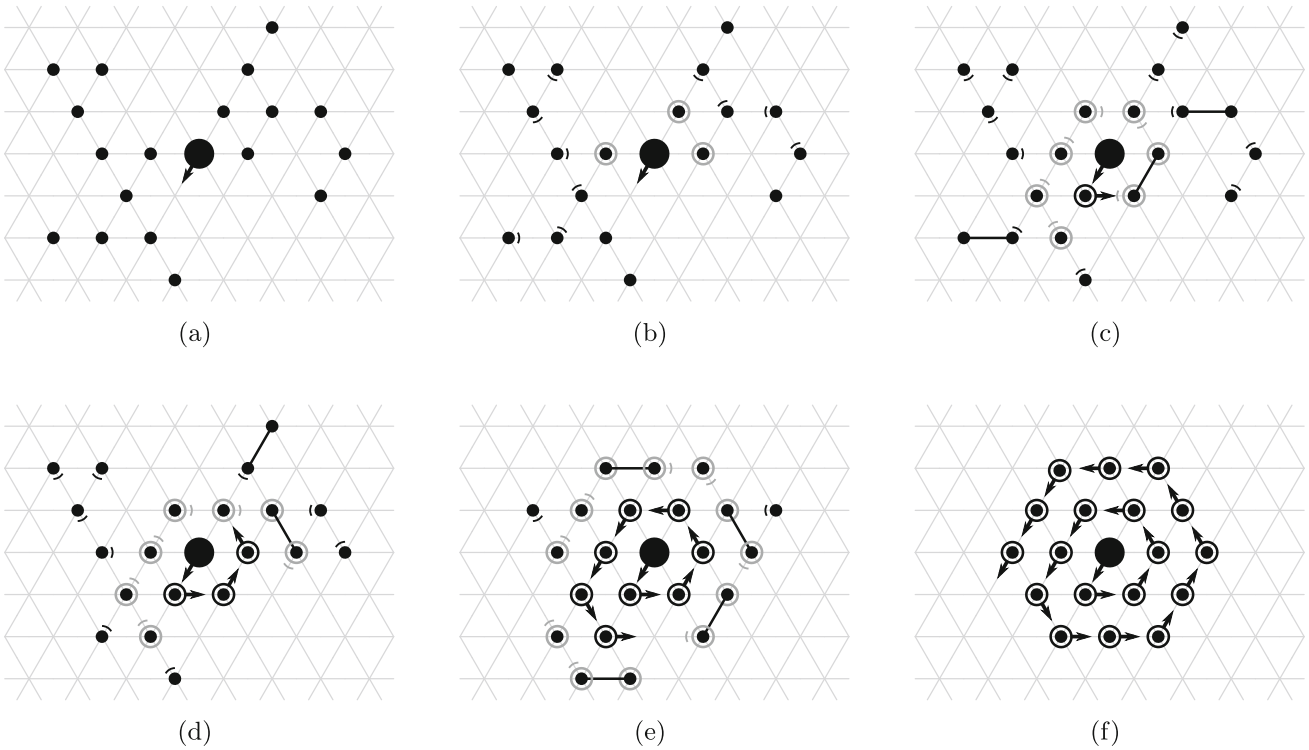


Fig. 6 An example run of Hexagon-Formation with 19 amoebots. **a** All amoebots are initially idle (black dots), with the exception of a unique seed amoebot (large black dot). **b** Amoebots adjacent to the seed become roots (gray circles), and followers form parent–child relationships (black arcs) with roots and other followers. **c–f** Roots traverse the forming hexagon clockwise, becoming retired (black circles) when reaching the position marked by the last retired amoebot

mulated in terms of actions as specified in Sect. 2.3. Our analysis of Hexagon-Formation reveals a set of sufficient conditions for any amoebot algorithm's correctness under an unfair asynchronous adversary: (i) correctness under an unfair sequential adversary, (ii) enabled actions remaining enabled despite concurrent action executions, and (iii) executions of enabled actions remaining successful and unaffected by concurrent action executions. Any concurrent execution of an algorithm satisfying (ii) and (iii) can be shown to be serializable, which combined with sequential correctness establishes correctness under an unfair asynchronous adversary, the most general of all possible adversaries. Notably, we prove that our Hexagon-Formation algorithm satisfies these sufficient conditions without using locks, demonstrating that while locks are useful tools for designing correct amoebot algorithms under concurrent adversaries, they are not always necessary.

The hexagon formation problem tasks an arbitrary, connected system of initially contracted amoebots with forming a regular hexagon (or as close to one as possible, given the number of amoebots in the system). We assume that there is a *unique seed amoebot* in the system and all other amoebots are initially *idle*; note that the seed amoebot immediately collapses the hierarchy of orientation assumptions since it can impose its own local orientation on the rest of the system. Following the sequential algorithm given by Derakhshandeh et al. [20,25], the basic idea of our Hexagon-Formation algorithm is to form a hexagon by extending a spiral of amoebots counter-clockwise from the seed (see Fig. 6).

Algorithm 1 describes Hexagon-Formation in terms of actions. In addition to the shape variable assumed by the amoebot model, each amoebot A keeps variables $A.\texttt{state} \in \{\text{SEED}, \text{IDLE}, \text{FOLLOWER}, \text{ROOT}, \text{RETIRED}\}$, $A.\texttt{parent} \in \{\text{NULL}, 0, \ldots, 9\}$, and $A.\texttt{dir} \in \{\text{NULL}, 0, \ldots, 9\}$ in public memory. W.l.o.g., we assume that if multiple actions are enabled for an amoebot, the enabled action with smallest index is executed.[2] In action guards, we use $N(A)$ to denote the neighbors of amoebot A and say that an amoebot A has a *tail-child* B if B is connected to the tail of A via port $B.\texttt{parent}$.

The amoebot system first self-organizes as a spanning forest rooted at the seed amoebot using their `parent` ports (action α_2). Follower amoebots follow their parents until reaching the surface of retired amoebots that have already found their place in the hexagon (actions α_5 and α_6). They then become roots (action α_1), traversing the surface of retired amoebots clockwise (actions α_4 and α_5). Once they

[2] Observe that any amoebot algorithm could directly implement this assumption by replacing each guard g_i of action α_i with the guard $g_i \wedge \bigwedge_{j=1}^{i-1}(\neg g_j)$.

connect to a retired amoebot's `dir` port, they also retire and set their `dir` port to the next position of the hexagon (action α_3). This process continues until all non-seed amoebots retire, forming a hexagon.

We begin our analysis of the Hexagon-Formation algorithm by showing it is correct under an unfair sequential adversary. Although the related algorithm of Derakhshandeh et al. has already been analyzed in the sequential setting [20,25], Hexagon-Formation must be proved correct with respect to its action formulation.

Lemma 3 *Any unfair sequential execution of the Hexagon-Formation algorithm terminates with the amoebot system forming a hexagon.*

Proof We first show that the system remains connected throughout the execution. Recall that the amoebot system is assumed to be initially connected. A disconnection can only result from a movement, and in particular, a contraction. Expansions only enlarge the set of nodes occupied by the system and handovers only change which amoebot occupies the handover node, not the fact that the node remains occupied. So it suffices to consider α_6, the only action involving a CONTRACT operation. Action α_6 only allows an expanded follower or root amoebot to contract its tail if it has no idle neighbors or neighbors pointing at its tail as their parent. The only other possible tail neighbors are the seed, roots, or retired amoebots; however, all of these neighbors are guaranteed to be connected to the forming hexagon structure. Thus, the system remains connected throughout the algorithm's execution.

Now, suppose to the contrary that the Hexagon-Formation algorithm has terminated—i.e., no amoebot has an enabled action—but the system does not form a hexagon. By inspection of action α_3, the retired amoebots form a hexagon extending counter-clockwise from the seed. Thus, for the system to not form a hexagon, there must exist some amoebot that is neither the seed nor retired.

First of all, there cannot be any idle amoebots remaining in the system; in particular, we argue that so long as there are idle amoebots in the system, there exists an idle amoebot for which α_1 or α_2 is enabled, and thus the algorithm cannot have terminated. Suppose to the contrary that there are idle amoebots in the system but none of them have non-idle neighbors, yielding α_1 and α_2 disabled. Then the idle amoebots must be disconnected from the rest of the system, since we assumed that the system contains a unique seed amoebot initially, a contradiction of connectivity. Thus, if the algorithm has terminated, all idle amoebots must have already become roots or followers.

For all root or follower amoebots to be disabled, we have the following chain of observations:

(a) No follower can have a seed or retired neighbor; otherwise, action α_1 would be enabled for that follower.
(b) Since we have already established that there are no idle amoebots in the system, there must not be a contracted root occupying the next hexagon node; otherwise, action α_3 would be enabled for that root.
(c) Every contracted root amoebot must have its clockwise traversal of the forming hexagon's surface blocked by another amoebot; otherwise, action α_4 would be enabled for some contracted root. Moreover, since there are no followers on the hexagon's surface by (a) and no contracted root has yet reached the next hexagon node by (b), each contracted root must be blocked by another root.
(d) By (c), there must exist at least one expanded root amoebot A. Since actions α_5 and α_6 must be disabled for A by supposition—and, again, there are no idle amoebots remaining in the system—A must have one or more tail-children that are all expanded.
(e) By the same argument, actions α_5 and α_6 can only be disabled for the expanded tail-children of A if they also each have at least one tail-child, all of which are expanded.

The chain of expanded tail-children established by (d) and (e) cannot continue ad infinitum since the amoebot system is finite. There must eventually exist an expanded root or follower amoebot that either has a contracted tail-child or no tail-children, enabling α_5 or α_6, respectively. In all cases, we reach a contradiction: so long as the amoebot system does not yet form a hexagon, there must exist an amoebot with an enabled action. The execution of any enabled action brings the system monotonically closer to forming a hexagon: turning idle amoebots into followers, bringing followers to the hexagon's surface, turning followers into roots, bringing roots closer to their final position, and finally turning roots into retired amoebots. Therefore, regardless of the unfair sequential adversary's choice of enabled amoebot to activate, the system is guaranteed to reach and terminate in a configuration forming a hexagon, as desired. □

We next consider unfair asynchronous executions, the most general of all possible concurrency assumptions. The Hexagon-Formation algorithm maintains the following invariants:

(i) The `state` variable of an amoebot A can only be updated by A itself. This follows from actions α_1, α_2, and α_3.
(ii) Only follower amoebots have non-NULL `parent` variables. An idle amoebot sets its own `parent` variable when it becomes a follower. While an amoebot A is a follower, the only amoebot that can update A.`parent` is the amoebot indicated by A.`parent`. Finally, when a follower becomes a root, it updates its own `parent` variable to NULL, after which its `parent` variable never changes again. This follows from actions α_1, α_2, and α_5.

Algorithm 1 Hexagon-Formation for Amoebot A

1: $\alpha_1 : (A.\texttt{state} \in \{\text{IDLE, FOLLOWER}\}) \wedge (\exists B \in N(A) : B.\texttt{state} \in \{\text{SEED, RETIRED}\}) \rightarrow$
2: WRITE($\perp$, parent, NULL).
3: WRITE($\perp$, state, ROOT).
4: WRITE($\perp$, dir, GETNEXTDIRcounter-clockwise). ▷ See Algorithm 2.
5: $\alpha_2 : (A.\texttt{state} = \text{IDLE}) \wedge (\exists B \in N(A) : B.\texttt{state} \in \{\text{FOLLOWER, ROOT}\}) \rightarrow$
6: Find a port p for which CONNECTED(p) = TRUE and READ(p, state) $\in$ {FOLLOWER, ROOT}.
7: WRITE($\perp$, parent, p).
8: WRITE($\perp$, state, FOLLOWER).
9: $\alpha_3 : (A.\texttt{shape} = \text{CONTRACTED}) \wedge (A.\texttt{state} = \text{ROOT}) \wedge (\forall B \in N(A) : B.\texttt{state} \neq \text{IDLE})$
10: $\wedge\ (\exists B \in N(A) : (B.\texttt{state} \in \{\text{SEED, RETIRED}\}) \wedge (B.\texttt{dir}$ is connected to $A)) \rightarrow$
11: WRITE($\perp$, dir, GETNEXTDIRclockwise).
12: WRITE($\perp$, state, RETIRED).
13: $\alpha_4 : (A.\texttt{shape} = \text{CONTRACTED}) \wedge (A.\texttt{state} = \text{ROOT}) \wedge$ (the node adjacent to $A.\texttt{dir}$ is empty) $\rightarrow$
14: Let $p \leftarrow$ READ($\perp$, dir).
15: EXPAND(p).
16: $\alpha_5 : (A.\texttt{shape} = \text{EXPANDED}) \wedge (A.\texttt{state} \in \{\text{FOLLOWER, ROOT}\}) \wedge (\forall B \in N(A) : B.\texttt{state} \neq \text{IDLE})$
17: $\wedge\ (A$ has a tail-child $B : B.\texttt{shape} = \text{CONTRACTED}) \rightarrow$
18: **if** READ($\perp$, state) = ROOT **then** WRITE($\perp$, dir, GETNEXTDIRcounter-clockwise).
19: Find a port $p \in$ TAILCHILDREN s.t. READ(p, shape) = CONTRACTED. ▷ See Algorithm 2.
20: Let p' be the label of the tail-child's port that will be connected to p after the pull handover.
21: WRITE(p, parent, p').
22: PULL(p).
23: $\alpha_6 : (A.\texttt{shape} = \text{EXPANDED}) \wedge (A.\texttt{state} \in \{\text{FOLLOWER, ROOT}\}) \wedge (\forall B \in N(A) : B.\texttt{state} \neq \text{IDLE})$
24: $\wedge\ (A$ has no tail-children) $\rightarrow$
25: **if** READ($\perp$, state) = ROOT **then** WRITE($\perp$, dir, GETNEXTDIRcounter-clockwise).
26: CONTRACT(TAIL).

Algorithm 2 Helper Functions for Hexagon-Formation

1: **function** GETNEXTDIR(c) ▷ $c \in$ {clockwise, counter-clockwise}
2: Let p be any head port.
3: **try:**
4: **while** $\neg$CONNECTED(p) $\vee$ (READ(p, state) $\notin$ {SEED, RETIRED}) **do**
5: $p \leftarrow$ the next head port in orientation c.
6: **catch** disconnect-failure **do** $p \leftarrow$ the next head port in orientation c; go to Step 4.
7: **try:**
8: **while** CONNECTED(p) $\wedge$ (READ(p, state) $\in$ {SEED, RETIRED}) **do**
9: $p \leftarrow$ the next head port in orientation c.
10: **catch** disconnect-failure **do** $p \leftarrow$ the next head port in orientation c; go to Step 8.
11: **return** p.
12: **function** TAILCHILDREN( )
13: Let $P \leftarrow \emptyset$.
14: **for** each tail port p **do**
15: **try:**
16: **if** CONNECTED(p) $\wedge$ (READ(p, parent) points to A) **then**
17: $P \leftarrow P \cup \{p\}$.
18: **catch** disconnect-failure **do** nothing.
19: **return** P.

(iii) Only root and retired amoebots have non-NULL dir variables. The dir variable of an amoebot A can only be updated by A itself. Once a dir variable is set by a retired amoebot, it never changes again. This follows from actions α_1, α_3, α_5, and α_6.

(iv) Seed, idle, and retired amoebots are always contracted and never move. Moreover, seed and retired amoebots never change their state.

(v) The shape variable of a root or expanded follower A can only be updated by a movement operation initiated by A itself, while the shape variable of a contracted follower A can only be updated by a PULL operation initiated by the neighboring amoebot connected via A.parent. This follows from actions α_4, α_5, and α_6.

(vi) No amoebot can disconnect from an idle neighbor. Moreover, a root will not change its state if it has an idle neighbor. This follows from actions α_3, α_5, and α_6.

(vii) Root amoebots traverse the surface of the forming hexagon clockwise while follower amoebots are pulled by their parents. This follows from actions α_1, α_4, α_5, and α_6.

In general, asynchronous executions may cause amoebots to incorrectly evaluate their action guards. Nevertheless, in the following two lemmas, we show that Hexagon-Formation has the key property that whenever an amoebot thinks an action is enabled, it remains enabled and will execute successfully, even when other actions are executed concurrently.

Lemma 4 *For any asynchronous execution of the Hexagon-Formation algorithm, if an action α_i is enabled for an amoebot A, then α_i stays enabled for A until A executes an action.*

Proof We use the invariants to prove the claim on an action-by-action basis.

α_1 : If A evaluates the guard of α_1 as TRUE, then it must be an idle or follower amoebot with a seed or retired neighbor. Invariant (i) ensures that A remains an idle or follower amoebot, and Invariant (iv) ensures its seed or retired neighbor does not move or change state.

α_2 : If A evaluates the guard of α_2 as TRUE, then it must be an idle amoebot with a follower or root neighbor. Invariant (i) ensures that A remains an idle amoebot, and Invariant (vi) ensures that its neighbors remain connected to A while A is idle. A follower neighbor of A can concurrently change its state to root by α_1; however, a root neighbor of A will not change its state while A is idle by Invariant (vi).

α_3 : If A evaluates the guard of α_3 as TRUE, then it must be a contracted root with no idle neighbors and a seed or retired neighbor that indicates that the node A occupies is the next hexagon node. Invariants (i) and (v) ensure that A remains a contracted root, Invariant (iv) ensures that A cannot gain any idle neighbors, and Invariants (iii) and (iv) ensure that the seed or retired neighbor continues to indicate the node A occupies as the next hexagon node.

α_4 : If A evaluates the guard of α_4 as TRUE, then it must be a contracted root with no neighbor connected via $A.\mathtt{dir}$. Invariants (i) and (v) ensure that A remains a contracted root, and Invariant (vii) ensures that no amoebot but A can move into the node adjacent to $A.\mathtt{dir}$.

α_5 : If A evaluates the guard of α_5 as TRUE, then it must be an expanded follower or root with no idle neighbors and some contracted tail-child. Invariants (i) and (v) ensure that A remains an expanded follower or root, Invariant (iv) ensures that A cannot gain any idle neighbors, and Invariants (ii) and (v) ensure that any contracted tail-child of A remains so.

α_6 : If A evaluates the guard of α_6 as TRUE, then it must be an expanded follower or root with no idle neighbors and no tail-children. Invariants (i) and (v) ensure that A remains an expanded follower or root, and Invariants (ii) and (iv) ensure that A cannot gain any idle neighbors or tail-children.

Therefore, any action that A evaluates as enabled must remain enabled, as claimed. □

Lemma 5 *For any asynchronous execution of the Hexagon-Formation algorithm, any execution of an enabled action is successful and unaffected by any concurrent action executions.*

Proof We once again consider each action individually.

α_1 : Action α_1 first executes two WRITE operations to A's own public memory which cannot fail. It then executes a helper function GETNEXTDIR(counter-clockwise) which involves a sequence of CONNECTED and READ operations. CONNECTED operations always succeed, so it suffices to consider the READ operations. While it is possible that READ operations issued to follower or root neighbors may fail if those neighbors disconnect, these failures are caught by error handling and thus do not cause the action to fail. Moreover, the critical READ operations issued to seed or retired neighbors that the function depends on for calculating the correct direction must succeed by the guard of α_1 and Lemma 4. Once this direction is computed, α_1 then executes a WRITE operation to A's own memory which cannot fail.

α_2 : Action α_2 first executes CONNECTED and READ operations to find a follower or root neighbor. Such a neighbor must exist and the corresponding READ operations must succeed by the guard of α_2 and Lemma 4. Action α_2 then executes two WRITE operations to A's own public memory which cannot fail.

α_3 : Action α_3 first executes helper function GETNEXTDIR (clockwise) which must succeed by an argument analogous to that of α_1. Once this direction is computed, α_3 executes two WRITE operations to A's own public memory which cannot fail.

α_4 : Action α_4 executes an EXPAND operation toward port $A.\mathtt{dir}$ which must succeed because A is contracted and the node adjacent to $A.\mathtt{dir}$ must remain unoccupied, as ensured by the guard of α_4 and Lemma 4.

α_5 : Action α_5 first executes a conditional based on a READ operation issued to A's own public memory which

cannot fail. It then executes helper function GETNEXTDIR(counter-clockwise) which must succeed by an argument analogous to that of α_1. The computed direction is then used in a WRITE operation to A's own public memory which cannot fail. Action α_5 then executes a helper function TAILCHILDREN() which, like GETNEXTDIR, involves CONNECTED and READ operations. It must succeed for similar reasons: any failed READ operations are caught by error handling, and the critical READ operations issued to tail-children must succeed by the guard of α_5 and Lemma 4. Once the ports connected to tail-children are computed, READ operations are executed to find a contracted tail-child B which once again must succeed by the guard of α_5 and Lemma 4. Finally, α_5 executes a WRITE to the public memory of B and performs a PULL handover with B; both operations must succeed because B remains connected to A and cannot be involved in another movement by Invariant (v).

α_6 : Action α_6 first executes the same conditional operation as α_5 and thus succeeds for an analogous reason. It then executes a single CONTRACT operation which must succeed because A is expanded, as ensured by the guard of α_6 and Lemma 4.

Therefore, any execution of an enabled action must be successful and unaffected by concurrent action executions, as claimed. □

We next show that the Hexagon-Formation algorithm is serializable. We denote the execution of an action α by an amoebot A in an execution of the algorithm as a pair (A, α).

Lemma 6 *For any asynchronous execution of the Hexagon-Formation algorithm, there exists a sequential ordering of its action executions producing the same final configuration.*

Proof Argue by induction on i, the number of action executions in the asynchronous execution of Hexagon-Formation. Clearly, if $i = 1$, the asynchronous execution of a single action is also a sequential execution, and we are done. So suppose that any asynchronous execution of Hexagon-Formation consisting of $i \geq 1$ action executions can be serialized, and consider any asynchronous execution $\mathcal{S}$ consisting of $i+1$ action executions. Consider a *wall-clock* that marks the exact time that any amoebot A starts an action execution (A, α); we emphasize that this wall-clock timing is only used for this analysis and is never available to the amoebots. Partially order the action executions (A, α) of $\mathcal{S}$ according to their wall-clock start times. Let (A^*, α^*) be the action execution with the latest activation time; if there are multiple such executions because the asynchronous adversary activated multiple amoebots simultaneously, choose any such execution. If (A^*, α^*) was removed from $\mathcal{S}$ to produce a new asynchronous execution $\mathcal{S}^-$, we have by Lemmas 4 and 5 that the remaining i action executions must still be enabled and successful since all other action executions either terminated before (A^*, α^*) was initiated or were concurrent with it. By the induction hypothesis, there must exist a sequential ordering of the i action executions in $\mathcal{S}^-$ producing the same final configuration as $\mathcal{S}^-$. Append (A^*, α^*) to the end of this sequential execution to produce $\mathcal{S}^*$, a sequential execution of $i+1$ action executions. Any actions that were concurrent with (A^*, α^*) in $\mathcal{S}$ have now terminated before (A^*, α^*) in $\mathcal{S}^*$. However, by Lemmas 4 and 5, this does not change the fact that α^* is enabled for A^* and its execution is successful and produces the same outcome in $\mathcal{S}^*$. Therefore, we conclude that there exists a sequential ordering of the action executions of $\mathcal{S}$ producing the same final configuration. □

Finally, we show that the Hexagon-Formation algorithm is correct under an unfair asynchronous adversary.

Lemma 7 *Any unfair asynchronous execution of the Hexagon-Formation algorithm terminates with the amoebot system forming a hexagon.*

Proof First suppose to the contrary that there exists an asynchronous execution of Hexagon-Formation that does not terminate; i.e., there are an infinite number of executions of enabled actions. By Lemmas 4 and 5, any such action execution must succeed and do exactly what it would have in a sequential execution where there are no other concurrent action executions. But Lemma 3 implies that there can only be a finite number of successful action executions before no amoebot has any enabled actions left, a contradiction. So all asynchronous executions of Hexagon-Formation must terminate.

Now suppose to the contrary that there exists an asynchronous execution of Hexagon-Formation that has terminated but the system does not form a hexagon. By Lemma 6, there must exist a sequential execution that also produces this non-hexagon final configuration. However, this is a contradiction of Lemma 3 which states that every sequential execution of Hexagon-Formation must terminate with the system forming a hexagon. □

Our analysis culminates in the following theorem.

Theorem 8 *Assuming geometric space, assorted orientations, and constant-size memory, the Hexagon-Formation algorithm solves the hexagon formation problem under any adversary.*

Serializability and correctness under an asynchronous adversary (Lemmas 6 and 7) follow directly from Lemmas 3–5, independent of the specific details of Hexagon-Formation. Thus, Lemmas 3–5 establish a set of general sufficient conditions for amoebot algorithm correctness under an

asynchronous adversary. We are optimistic that other existing amoebot algorithms, once translated into action formulations, will also satisfy these conditions.

4 A general framework for concurrency control

In the sequential setting where only one amoebot is active at a time, operation failures are necessarily the fault of the algorithm designer: e.g., attempting to READ on a disconnected port, attempting to EXPAND when already expanded, etc. Barring these design errors, it suffices to focus only on the correctness of the algorithm—i.e., whether the algorithm's actions always produce the desired system behavior under any sequential execution—not whether the individual actions themselves execute as intended. This is the focus of most existing amoebot works [2,9,10,18,19,25–28,33,34,37].

Our present focus is on asynchronous executions, where concurrent action executions can mutually interfere, affect outcomes, and cause failures far beyond those of simple designer negligence. Ensuring algorithm correctness in spite of concurrency thus appears to be a significant burden for the algorithm designer, especially for problems that are challenging even in the sequential setting due to the constraints of constant-size memory, assorted orientation, and strictly local interactions. What if there was a way to ensure that correct, sequential amoebot algorithms could be lifted to the asynchronous setting without sacrificing correctness? This would give the best of both worlds: the relative ease in design from the sequential setting and the correct execution in a more realistic concurrent setting.

In this section, we introduce and rigorously analyze a framework for transforming an algorithm $\mathcal{A}$ that works correctly for every sequential execution into an algorithm $\mathcal{A}'$ that works correctly for every asynchronous execution. We prove that our framework achieves this goal so long as the original algorithms satisfy certain *conventions*. These conventions limit the full generality of the amoebot model in order to provide a common structure to the algorithms. In Sect. 4.1, we define these conventions and prove that they are satisfied by both the Hexagon-Formation algorithm of Sect. 3 and a broad class of *stationary* amoebot algorithms. This implies that these algorithms are immediately compatible with our concurrency control framework, which we detail in Sect. 4.2 and rigorously analyze in Sect. 4.3.

4.1 Algorithm conventions for concurrency control

The first convention requires that all actions of the given algorithm are executed successfully under a sequential adversary. For sequential executions, the *system configuration* is defined as the mapping of amoebots to the node(s) they occupy and the contents of each amoebot's public memory. Certainly, this configuration is well-defined whenever all amoebots are inactive, and we call a configuration *legal* whenever the requirements of our amoebot model are met, i.e., every position is occupied by at most one amoebot, each amoebot is either contracted or expanded, its `shape` variable corresponds to its physical shape, and its `lock` variable corresponds to its lock state. Whenever we talk about a system configuration in the following, we assume that it is legal.

Convention 1 (Validity) *All actions α of an amoebot algorithm $\mathcal{A}$ should be valid, i.e., for all system configurations in which α is enabled for some amoebot A, the execution of α by A should be successful whenever all other amoebots are inactive.*

The second convention defines a common structure for an algorithm's actions by controlling the order and number of operations they perform. This structure is similar in spirit to the *look-compute-move* paradigm used in the mobile robots literature (see, e.g., [35]), though the canonical amoebot model's underlying message passing communication adds additional complexity. Moreover, the instantaneous snapshot performed in the mobile robots' look phase is not trivially realizable by amoebots whose public memories are included in neighborhood configurations (Sect. 1.1).

Convention 2 (Phase Structure) *Each action of an amoebot algorithm $\mathcal{A}$ should structure its operations as: (1) a compute phase, during which an amoebot performs a finite amount of computation and a finite sequence of* CONNECTED, READ, *and* WRITE *operations, and (2) a move phase, during which an amoebot performs at most one movement operation decided upon in the compute phase. In particular, no action should use* LOCK *or* UNLOCK *operations.*

The third and final convention, *expansion-robustness*, allows us to map asynchronous executions of algorithms produced by the concurrency control framework to related sequential executions. A key challenge in achieving this mapping for concurrent executions of amoebot algorithms is the possibility of one or more amoebots expanding into the neighborhood of an amoebot A that has already started executing an action of its own. These newly expanded neighbors were not present when A evaluated its action guard, and thus may cause the execution to exhibit different behavior than in the sequential setting—or worse, fail altogether. Informally, an expansion-robust algorithm is ambivalent to these concurrent expansions, guaranteeing correct behavior regardless.

Formally, let $\mathcal{A}$ be any amoebot algorithm satisfying Conventions 1 and 2 and consider its expansion-robust variant $\mathcal{A}^E$ defined as follows. Each amoebot A executing $\mathcal{A}^E$ additionally stores in public memory *expand flags* $A.\texttt{flag}_p \in \{\text{TRUE}, \text{FALSE}\}$ for each of its ports p that are initially set to FALSE. These expand flags communicate when an

Algorithm 3 Expansion-Robust Variant $\mathcal{A}^E$ of Algorithm $\mathcal{A}$ for Amoebot A

Input: Algorithm $\mathcal{A} = \{[\alpha_i : g_i \to ops_i] : i \in \{1, \ldots, m\}\}$ satisfying Conventions 1 and 2.
1: Set $\alpha_0^E : (\exists$ port p of $A : A.\texttt{flag}_p =$ TRUE$) \to$ WRITE$(\perp, \texttt{flag}_p,$ FALSE$)$.
2: **for** each action $[\alpha_i : g_i \to ops_i] \in \mathcal{A}$ **do**
3: Set $g_i^E \leftarrow g_i$ with $N(A)$ replaced by $N^E(A)$ and connections defined w.r.t. $N^E(A)$.
4: Set $ops_i^E \leftarrow$ "Do:
5: **for** each port p of A **do** WRITE$(\perp, \texttt{flag}_p,$ FALSE$)$. ▷ Reset own expand flags.
6: **for** each unique neighbor $B \in$ CONNECTED() **do** ▷ Reset neighbor's expand flags.
7: **for** each port p of B **do** WRITE$(B, \texttt{flag}_p,$ FALSE$)$.
8: Execute each operation of ops_i with connections defined w.r.t. $N^E(A)$.
9: **if** a PULL or PUSH operation was executed with neighbor B **then**
10: **for** each new port p of A not connected to B **do** WRITE$(\perp, \texttt{flag}_p,$ TRUE$)$.
11: **for** each new port p of B not connected to A **do** WRITE$(B, \texttt{flag}_p,$ TRUE$)$.
12: **else if** an EXPAND operation was successfully executed **then**
13: **for** each new port p of A **do** WRITE$(\perp, \texttt{flag}_p,$ TRUE$)$.
14: **else if** an EXPAND operation failed in its execution **then** undo ops_i."
15: **return** $\mathcal{A}^E = \{[\alpha_i^E : g_i^E \to ops_i^E] : i \in \{0, \ldots, m\}\}$.

amoebot has newly expanded into another amoebot's neighborhood. Each action $\alpha_i : g_i \to ops_i$ in $\mathcal{A}$ translates to an action $\alpha_i^E : g_i^E \to ops_i^E$ in $\mathcal{A}^E$, as detailed in Algorithm 3.[3] The main difference is that while an amoebot A executes actions with respect to its full neighborhood $N(A)$ in algorithm $\mathcal{A}$, it does so only with respect to its *established neighborhood* $N^E(A) = \{B \in N(A) : \exists$ port p of B connected to A s.t. $B.\texttt{flag}_p =$ FALSE$\}$ in algorithm $\mathcal{A}^E$, effectively ignoring its newly expanded neighbors until its next action execution. The expansion-robustness convention can now be stated as follows:

Convention 3 (Expansion-Robustness) *An amoebot algorithm $\mathcal{A}$ should be expansion-robust, meaning that for any (legal) initial system configuration C_0 of $\mathcal{A}$, the following conditions hold:*

1. *Termination. If all sequential executions of $\mathcal{A}$ starting in C_0 terminate, all sequential executions of $\mathcal{A}^E$ starting in C_0^E (i.e., C_0 with all* FALSE *expand flags) also terminate.*
2. *Correctness. If some sequential execution of $\mathcal{A}^E$ starting in C_0^E terminates in a configuration C^E, there exists a sequential execution of $\mathcal{A}$ starting in C_0 that terminates in C (i.e., C^E without expand flags).*

It is worth emphasizing that $\mathcal{A}^E$ is simply a useful artifact for formalizing expansion-robustness and, as such, we are only interested in its behavior in the sequential setting. Several operations in $\mathcal{A}^E$, such as the "undo" on Line 14 of Algorithm 3, may not even be possible in a concurrent setting, but this is inconsequential for our purposes.

[3] For the sake of clarity and brevity, we abuse CONNECTED, READ, and WRITE notation slightly by referring directly to the neighboring amoebots and not to the ports which they are connected to.

We now demonstrate that Conventions 1–3 are not too limiting; i.e., there do exist algorithms that satisfy these conventions and thus are compatible with our concurrency control framework. Of the three conventions, expansion-robustness (Convention 3) is the most technically difficult to verify. However, *stationary* amoebot algorithms $\mathcal{A}$—i.e., those that do not perform any movement operations, including many of the existing algorithms for leader election [5,19,28,32,37] and the recent algorithm for energy distribution [23]—are trivially expansion-robust since no amoebot ever moves and thus $\mathcal{A}$ and $\mathcal{A}^E$ are identical.

Observation 9 *Any stationary amoebot algorithm satisfies Convention 3.*

In the conference version of this work [21], it remained an open question whether any amoebot algorithm involving movement satisfied all three conventions. By replacing the prior version's *monotonicity* convention with expansion-robustness, we identify the Hexagon-Formation algorithm of Sect. 3 as such an algorithm.

Theorem 10 *The Hexagon-Formation algorithm satisfies Conventions 1–3.*

Proof It is easy to verify that Hexagon-Formation satisfies Conventions 1 and 2 by inspection. Hence, it remains to show Hexagon-Formation is expansion-robust (Convention 3). To prove correctness, we will show that whenever an action $\alpha_i^E \in$ Hexagon-Formation^E (other than α_0^E) is enabled for an amoebot A w.r.t. $N^E(A)$, action $\alpha_i \in$ Hexagon-Formation is enabled for A w.r.t. $N(A)$. Moreover, we show that the executions of α_i and α_i^E by A are identical except for the handling of expand flags. This immediately implies that every sequential execution of Hexagon-Formation^E represents an identical sequential execution of Hexagon-Formation (after removing the executions of α_0^E), proving correctness. Observe that expand flags

are only set to TRUE in Hexagon-Formation^E as a result of an EXPAND, PULL, or PUSH operation, which are specific to α_4^E and α_5^E; thus, we need only focus on amoebots executing these actions.

Suppose a contracted root amoebot A executes α_4^E, expanding towards $A.\texttt{dir}$ along the surface of the forming hexagon into the neighborhood of another amoebot B it was not already connected to. If B is a root, then it can be easily verified by inspecting the guards and operations of $\alpha_3, \alpha_4, \alpha_5$, and α_6 that A—which must be "behind" B in the clockwise traversal of the hexagon's surface—has no bearing on which actions are enabled for B nor on their execution. If B is a follower, then its parent must be some amoebot other than A because A only just became its neighbor. Thus, in either of these cases, the fact that $A \in N(B) \setminus N^E(B)$ is inconsequential. Finally, if B is idle, then $A \notin N^E(B)$ prohibits B from choosing A as its parent in α_2^E while the same choice is allowed in α_2. If B chooses some amoebot $C \neq A$ as its parent while executing α_2^E, then certainly an execution of α_2 by B in the same configuration could have made the same choice. Otherwise, if A is the only neighbor of B, then we know α_0^E is continuously enabled for A while $A \notin N^E(B)$. Amoebot A cannot disconnect from B while B is idle by the guards of α_3, α_5, and α_6, so eventually an execution of α_0^E resets the expand flags of A, allowing B to choose A as its parent just as in its corresponding execution of α_2.

Now suppose an expanded follower or root amoebot A executes α_5^E, pulling some follower tail-child B in a handover. Consider any new port p of B for which $B.\texttt{flag}_p =$ TRUE after this handover occurs. If there is no neighboring amoebot connected to port p, then only a contracted root could expand into that position, as already covered in the analysis of α_4^E. So suppose that an amoebot C is connected to B via port p. The guard of α_5 would have prohibited A from performing the pull handover with B if A had any idle neighbors, so C cannot be idle. Inspection of the guards and operations of actions α_1, α_5, and α_6 show that C is irrelevant to B if C is a root. So it remains to consider if C is a follower. If $C.\texttt{parent}$ points to any node other than the new head of B, then $B \in N(C) \setminus N^E(C)$ is inconsequential to C. Otherwise, if $C.\texttt{parent}$ refers to the new head of B, the fact that $B \notin N^E(C)$ is once again inconsequential to C because children never initiate interactions with their parents. Therefore, in all cases, the correctness condition of expansion-robustness follows.

To prove termination, suppose to the contrary that there exists a sequential execution $\mathcal{S}^E$ of Hexagon-Formation^E starting in a legal initial configuration C_0^E that contains an infinite number of action executions. By our correctness analysis, we know that $\mathcal{S}^E$ must correspond to an identical sequential execution $\mathcal{S}$ of Hexagon-Formation, modulo executions of α_0^E. In Sect. 3, we proved that all sequential executions of Hexagon-Formation, $\mathcal{S}$ included, must terminate (Lemma 3). Thus, $\mathcal{S}^E$ must contain an infinite number of executions of α_0^E. But this is impossible, as there are a finite number of amoebots and each of them has a finite number of expand flags to reset with α_0^E, a contradiction. Thus, the termination condition is satisfied, and Hexagon-Formation is expansion-robust. □

With the validity, phase structure, and expansion-robustness conventions established, we now turn to the description and analysis of the concurrency control framework.

4.2 The concurrency control framework

Our *concurrency control framework* (Algorithm 4) takes as input any amoebot algorithm $\mathcal{A} = \{[\alpha_i : g_i \to ops_i] : i \in \{1, \ldots, m\}\}$ satisfying Conventions 1–3 and produces a corresponding algorithm $\mathcal{A}' = \{[\alpha' : g' \to ops']\}$ composed of a single action α'. The core idea of our framework is to carefully incorporate locks in α' as a wrapper around the actions of $\mathcal{A}$, ensuring that $\mathcal{A}'$ only produces outcomes in concurrent settings that $\mathcal{A}$ can produce in the sequential setting. With locks, action guards that in general can only be evaluated reliably in the sequential setting can now also be evaluated reliably in concurrent settings.

To avoid any deadlocks that locking may cause, our framework adds an *activity bit* variable $A.\texttt{act} \in$ {TRUE, FALSE} to the public memory of each amoebot A indicating if any changes have occurred in the memory or neighborhood of A since it last attempted to execute an action. The single action α' of $\mathcal{A}'$ has guard $g' = (A.\texttt{act} =$ TRUE), ensuring that α' is only enabled for an amoebot A if changes in its memory or neighborhood may have caused some actions of $\mathcal{A}$ to become enabled. As will become clear in the presentation of the framework, WRITE and movement operations may enable actions of $\mathcal{A}$ not only for the neighbors of the acting amoebot, but also for the neighbors of those neighbors (i.e., in the 2-neighborhood of the acting amoebot). The acting amoebot cannot directly update the activity bits of amoebots in its 2-neighborhood, so it instead sets its neighbors' *awaken bits* $A.\texttt{awaken} \in$ {TRUE, FALSE} to indicate that they should update their neighbors' activity bits in their next action. Initially, $A.\texttt{act} =$ TRUE and $A.\texttt{awaken} =$ FALSE for all amoebots A.

Algorithm $\mathcal{A}'$ only contains one action $\alpha' : g' \to ops'$ where g' requires that an amoebot's activity bit is set to TRUE (Step 1). If α' is enabled for an amoebot A, A first attempts to LOCK itself and its persistent neighbors (Step 2). Given that it locks successfully, there are two cases. If $A.\texttt{awaken} =$ TRUE, then some amoebot must have previously changed the neighborhood of A without being able to update the corresponding neighbors' activity bits (Steps 14, 17, 24, or 28). So A updates the intended activity bits to TRUE, resets $A.\texttt{awaken}$, releases its locks, and aborts (Steps 4–6).

Algorithm 4 Concurrency Control Framework for Amoebot A

Input: Algorithm $\mathcal{A} = \{[\alpha_i : g_i \to ops_i] : i \in \{1, \ldots, m\}\}$ satisfying Conventions 1–3.

1: Set $g' \leftarrow (A.\texttt{act} = \text{TRUE})$ and $ops' \leftarrow$ "Do:
2: **try:** Set $\mathcal{L} \leftarrow$ LOCK() to attempt to lock A and its persistent neighbors.
3: **catch** `lock-failure` **do** abort.
4: **if** READ($\perp$, `awaken`) = TRUE **then**
5: **for all** amoebots $B \in \mathcal{L}$ **do** WRITE(B, `act`, TRUE).
6: WRITE($\perp$, `awaken`, FALSE), UNLOCK($\mathcal{L}$), and abort.
7: **for all** actions $[\alpha_i : g_i \to ops_i] \in \mathcal{A}$ **do**
8: Perform CONNECTED and READ operations to evaluate guard g_i w.r.t. $\mathcal{L}$.
9: Evaluate g_i in private memory to determine if α_i is enabled.
10: **if** no action is enabled **then** WRITE($\perp$, `act`, FALSE), UNLOCK($\mathcal{L}$), and abort.
11: Choose an enabled action $\alpha_i \in \mathcal{A}$ and perform its compute phase in private memory.
12: Let W_i be the set of WRITE operations and M_i be the movement operation in ops_i based on its compute phase; set $M_i \leftarrow$ NULL if there is none.
13: **if** M_i is EXPAND (say, from node u into node v) **then**
14: **try:** Perform the EXPAND operation and WRITE($\perp$, `awaken`, TRUE).
15: **catch** `expand-failure` **do** UNLOCK($\mathcal{L}$) and abort.
16: **for all** amoebots $B \in \mathcal{L}$ **do** WRITE(B, `act`, TRUE).
17: **for all** WRITE(B, x, x_{val}) $\in W_i$ **do** WRITE(B, x, x_{val}) and WRITE(B, `awaken`, TRUE).
18: **if** M_i is NULL or EXPAND **then** UNLOCK($\mathcal{L}$).
19: **else if** M_i is CONTRACT (say, from nodes u, v into node u) **then**
20: UNLOCK each amoebot in $\mathcal{L}$ that is adjacent to node v but not to node u.
21: Perform the CONTRACT operation.
22: UNLOCK each remaining amoebot in $\mathcal{L}$.
23: **else if** M_i is PUSH (say, A is pushing B) **then**
24: WRITE($\perp$, `awaken`, TRUE) and WRITE(B, `awaken`, TRUE).
25: Perform the PUSH operation.
26: UNLOCK($\mathcal{L}$).
27: **else if** M_i is PULL (say, A in nodes u, v is pulling B into node v) **then**
28: WRITE(B, `awaken`, TRUE).
29: UNLOCK each amoebot in $\mathcal{L}$ (except B) that is adjacent to node v but not to node u.
30: Perform the PULL operation.
31: UNLOCK each remaining amoebot in $\mathcal{L}$."
32: **return** $\mathcal{A}' = \{[\alpha' : g' \to ops']\}$.

Otherwise, A obtains the necessary information to evaluate the guards of all actions in algorithm $\mathcal{A}$ (Steps 7–9). If no action of $\mathcal{A}$ is enabled for A, A sets $A.\texttt{act}$ to FALSE, releases its locks, and aborts; this disables α' for A until some future change occurs in its neighborhood (Step 10). Otherwise, A chooses any enabled action and executes its compute phase in private memory (Step 11) to determine which WRITE and movement operations, if any, it wants to perform (Step 12).

Before enacting these operations (thereby updating the system's configuration) amoebot A must be certain that no operation of α' will fail. It has already passed its first point of failure: the LOCK operation in Step 2. But the EXPAND operation of α' may also fail if it conflicts with some other concurrent expansion (Step 14). In either case, A handles the failure, releases any locks it obtained (if any), and aborts (Steps 3 and 15). As we will show in Lemma 13, these are the only two operations of α' that can fail. Provided neither of these failures occur, A can now perform operations that—without locks on its neighbors—could otherwise interfere with its neighbors' actions or be difficult to undo. This begins with A setting the activity bits of all its locked neighbors to TRUE since it is about to cause activity in its neighborhood (Step 16). It then enacts the WRITE operations it decided on during its computation, writing updates to its own public memory and the public memories of its neighbors. Since writes to its neighbors can change what amoebots in its 2-neighborhood see, it must also set the awaken bits of the neighbors it writes to to TRUE (Step 17).

The remainder of the framework handles movements and releases locks. If A did not want to move or it intended to EXPAND—which, recall, it already did in Step 14—it can simply release all its locks (Step 18). If A wants to contract, it must first release its locks on the neighbors it is contracting away from; it can then CONTRACT and, once contracted, release its remaining locks (Step 20–22). If A wants to perform a PUSH handover, it does so and then releases all its locks (Steps 24–26). Finally, pull handovers are handled similarly to contractions: A first releases its locks on the neighbors it is disconnecting from; it can then PULL and, once contracted, release its remaining locks (Steps 28–31).

4.3 Analysis

In this section, we prove the following result regarding the concurrency control framework.

Theorem 11 *Let $\mathcal{A}$ be any amoebot algorithm satisfying Conventions 1–3 and $\mathcal{A}'$ be the amoebot algorithm produced from $\mathcal{A}$ by the concurrency control framework (Algorithm 4). Let C_0 be any (legal) initial system configuration for $\mathcal{A}$ and let C_0' be its extension for $\mathcal{A}'$ that adds $A.\texttt{act} = \text{TRUE}$ and $A.\texttt{awaken} = \text{FALSE}$ for all amoebots A. If every sequential execution of $\mathcal{A}$ starting in C_0 terminates, every asynchronous execution of $\mathcal{A}'$ starting in C_0' also terminates. Moreover, if C' is the final configuration of some asynchronous execution of $\mathcal{A}'$ starting in C_0', then there exists a sequential execution of $\mathcal{A}$ starting in C_0 with final configuration C that is identical to C', modulo amoebots' activity and awaken bits.*

Informally, this theorem shows that the concurrency control framework only permits asynchronous outcomes that could have occurred in the sequential setting, provided algorithm $\mathcal{A}$ always terminates in the sequential setting and satisfies the three conventions.

This analysis has three parts. First, we show that asynchronous executions of $\mathcal{A}'$ can be "sanitized" of "irrelevant" events without changing the system's final configuration (Observation 12–Lemma 15). Second, we show that any sanitized asynchronous execution of $\mathcal{A}'$ can be transformed into a sequential execution of $(\mathcal{A}^E)'$, the framework-applied expansion-robust version of $\mathcal{A}$, again without changing the final configuration (Lemmas 16–18). Finally, we leverage the expansion-robustness of $\mathcal{A}$ (Convention 3) to show that any final configuration reached by a sequential execution of $(\mathcal{A}^E)'$ is also reachable by a sequential execution of $\mathcal{A}$ (Lemmas 19–21). Combining these results after showing asynchronous executions of $\mathcal{A}'$ terminate (Lemma 22) yields the theorem.

We first analyze algorithm $\mathcal{A}'$ under asynchronous executions. Recall from Sect. 2.3 that although each amoebot executes at most one action at a time and executes that action's operations sequentially to completion, asynchronous executions allow arbitrarily many amoebots to execute actions simultaneously. An *asynchronous schedule* is an assignment of precise timing by a wall-clock to every *event* in an asynchronous execution; i.e., every message sending and receipt, variable update in public memory, movement start and end, and operation failure. We emphasize that this wall-clock timing is only used for this analysis and is unavailable to the amoebots. In keeping with Sects. 2.2 and 2.3, we make no assumptions on timing other than (i) the delay between every message's sending and receipt as well as every movement's start and end must be positive, and (ii) the time taken by every operation execution—and, by extension, every action execution—must be finite. W.l.o.g., we may assume that any two events either occur simultaneously or are at least one time unit apart. We also assume, w.l.o.g., that at any time before the asynchronous schedule has terminated, there is at least one active amoebot; note that any positive delay during which all amoebots are inactive could be truncated so that the last action execution's end time coincides with the next action execution's start time without changing the system configuration. In addition to timing, an asynchronous schedule specifies the operations executed, all messages' contents, and variable values accessed and updated; i.e., all details except private computations.

To ensure that an asynchronous schedule captures the actual system behavior of an amoebot system under an asynchronous adversary, we introduce the concept of validity. An asynchronous schedule is *valid* if there is an asynchronous execution of (enabled) actions producing the same events (w.r.t. timing and content) as in the given asynchronous schedule. In the remainder of our analysis, whenever we refer to an asynchronous schedule, we assume its timing is in the control of an adversary constrained only by validity.

We begin with an observation that follows immediately from $\mathcal{A}'$ and Convention 2.

Observation 12 *Whenever an amoebot B is locked by an amoebot A in an execution of $\mathcal{A}'$, only A can initiate a movement with or update the public memory of B.*

Next, we identify the points of failure in action α' of $\mathcal{A}'$.

Lemma 13 *In an execution of action α', only the* LOCK *and* EXPAND *operations can fail.*

Proof The first operation A executes is the LOCK operation which may fail, as claimed. If it fails, α' catches the `lock-failure` and aborts, so no further operations are executed. Supposing the initial LOCK operation succeeds, let $\mathcal{L}_A$ denote the set of amoebots locked by A. Recall from Sect. 2.2.1 that a READ or WRITE operation by A can only fail if A is accessing a variable in the public memory of an amoebot $B \neq A$ that is disconnected from A during that operation's execution. By Observation 12, no amoebot in $\mathcal{L}_A$ can change its shape outside of a movement operation initiated by A. By inspection of α', A only executes READ and WRITE operations involving amoebots in $\mathcal{L}_A$ and does so before its movement operation; thus, they must succeed. Finally, UNLOCK operations cannot fail because they only involve locked amoebots, and CONNECTED operations always succeed.

It remains to consider the movement operations, all of which are determined by the execution of an enabled action $\alpha \in \mathcal{A}$. An EXPAND operation may fail, as claimed. A CONTRACT operation by A only fails if $A.\texttt{shape} \neq$ EXPANDED or A is already involved in a handover. By Convention 1, this contraction would succeed if all other amoebots were inactive, so A must have been expanded when

it evaluated the guard of α. Action α' does not contain any operations that change the shape of A between the guard evaluations and this CONTRACT operation, and by Observation 12 no other action executions could have involved A in a handover and changed its shape since $A \in \mathcal{L}_A$. Thus, A is expanded and cannot be involved in a handover when starting this contraction, so the CONTRACT operation succeeds.

A PULL operation by A with a neighbor B only fails if $A.\texttt{shape} \neq$ EXPANDED, $B.\texttt{shape} \neq$ CONTRACTED, A and B are not connected, or A or B is already involved in another handover. By Convention 1, this pull handover would succeed if all other amoebots were inactive, so A must have been expanded, B must have been contracted, and A and B must have been connected when A evaluated the guard of α. Once again, α' does not contain any operations that change the shape of A between the guard evaluations and this PULL operation, and by Observation 12 neither A nor B can be involved in another handover or could have changed their shape since $A, B \in \mathcal{L}_A$. So A is expanded, B is contracted, A and B are neighbors, and neither A nor B are involved in another handover when starting this pull handover, so the PULL operation succeeds. An analogous argument holds for PUSH operations.

Therefore, in an execution of α', only the LOCK or EXPAND operations can fail. □

We say that an amoebot is $\mathcal{A}$-*enabled* if it has at least one enabled action $\alpha \in \mathcal{A}$ and is $\mathcal{A}$-*disabled* otherwise. An execution of α' by an amoebot A is *relevant* in an asynchronous schedule of $\mathcal{A}'$ if all its operations succeed and either $A.\texttt{awaken} =$ TRUE or A is $\mathcal{A}$-enabled in α'. The next two lemmas show that we can *sanitize* any asynchronous schedule of $\mathcal{A}'$ by removing all events associated with *irrelevant* executions of α'—i.e., those with at least one failed operation or that have $A.\texttt{awaken} =$ FALSE and are $\mathcal{A}$-disabled—without changing the system's final configuration.

Lemma 14 *Let $\mathcal{S}$ be any asynchronous schedule of $\mathcal{A}'$ and let $\mathcal{S}_L$ be the asynchronous schedule obtained from $\mathcal{S}$ by removing all events except those associated with* LOCK *and* UNLOCK *operations and successful movements. Then $\mathcal{S}_L$ is valid w.r.t. its* LOCK *operations. Moreover, for any set X of successful* LOCK *operations in $\mathcal{S}_L$, the asynchronous schedule $\mathcal{S}_X$ obtained from $\mathcal{S}_L$ by removing all events associated with* LOCK *operations not in X is valid and all* LOCK *operations of X are successful and lock the same amoebots they did in $\mathcal{S}_L$.*

Proof First consider the asynchronous schedule $\mathcal{S}_L$ containing the events of $\mathcal{S}$ associated with all LOCK, UNLOCK, and successful movement operations. The only movements included in $\mathcal{S}$ that are not present in $\mathcal{S}_L$ are failed expansions. However, a failed expansion does not introduce any new connections or disconnections. Thus, since $\mathcal{S}$ is valid w.r.t. its LOCK operations and the validity of a LOCK operation depends only on other LOCK operations and amoebots' connections, $\mathcal{S}_L$ must also be valid w.r.t. its LOCK operations.

Consider any set X of successful LOCK operations in $\mathcal{S}_L$ and let $\mathcal{S}_X$ be the asynchronous schedule obtained from $\mathcal{S}_L$ by removing all events associated with LOCK operations not in X. Since any LOCK operation in X was successful in $\mathcal{S}_L$, then by the LOCK operation's mutual exclusion property all amoebots A it attempted to lock must have had $A.\texttt{lock} = \bot$. Removing other LOCK operations cannot cause amoebots' $\texttt{lock}$ variables to be $\neq \bot$. Thus, LOCK operations in X remain successful in $\mathcal{S}_X$. Any LOCK operation in X must also lock the same amoebots in $\mathcal{S}_X$ as it did in $\mathcal{S}_L$ since this depends only on connectivity—not on other LOCK operations—and the movement operations that control connectivity are identical in $\mathcal{S}_X$ and $\mathcal{S}_L$. Therefore, since $\mathcal{S}_L$ is valid w.r.t. its LOCK operations, so must $\mathcal{S}_X$. □

Lemma 15 *Let $\mathcal{S}$ be any asynchronous schedule of $\mathcal{A}'$ and let $\mathcal{S}^*$ be its sanitized version keeping only the events associated with relevant executions of α' in $\mathcal{S}$. Then $\mathcal{S}^*$ is a valid asynchronous schedule that changes the system configuration exactly as $\mathcal{S}$ does except w.r.t. amoebots' activity bits, which have the property that the set of amoebots A with $A.\texttt{act} =$ TRUE in $\mathcal{S}^*$ is always a superset of those in $\mathcal{S}$.*

Proof By Lemma 13, only LOCK or EXPAND operations can fail in an execution of α', implying three types of irrelevant executions of α' by an amoebot A: those whose LOCK operation fails, those whose LOCK operation succeeds but that have $A.\texttt{awaken} =$ FALSE and are $\mathcal{A}$-disabled, and those whose LOCK operation succeeds but whose EXPAND operation fails. By Lemma 14, when removing events associated with irrelevant executions of α' from $\mathcal{S}$ to obtain $\mathcal{S}^*$, all successful LOCK operations in $\mathcal{S}$ remain valid and successful in $\mathcal{S}^*$ and lock the same amoebots as they did in $\mathcal{S}$. Thus, the only change an irrelevant execution of α' could have made is setting an $\mathcal{A}$-disabled amoebot's activity bit to FALSE. This implies that the set of amoebots A with $A.\texttt{act} =$ TRUE in $\mathcal{S}^*$ is always a superset of those in $\mathcal{S}$ and thus any relevant action execution of α' in $\mathcal{S}$ remains enabled in $\mathcal{S}^*$.

Since relevant executions of α' only issue READ and WRITE operations to the executing amoebot or its locked neighbors, the success and identical outcome of all LOCK operations in $\mathcal{S}^*$ ensures that all READ and WRITE operations in $\mathcal{S}^*$ also succeed. Moreover, because irrelevant executions of α' never perform WRITE operations, all READ and WRITE operations in $\mathcal{S}^*$ must access or update the same variable values as they did in $\mathcal{S}$ since the event timing is preserved. CONNECTED operations in $\mathcal{S}^*$ are also guaranteed to return the same results as in $\mathcal{S}$ since failed EXPAND operations discarded from $\mathcal{S}$ do not change amoebot connectivity.

It remains to show that all movement operations in $\mathcal{S}^*$ are successful. Any CONTRACT, PULL, or PUSH operations in $\mathcal{S}^*$ must have succeeded in $\mathcal{S}$, implying that they were unaffected by any failed EXPAND operations in $\mathcal{S}$ that are now removed. So the only movement operations in $\mathcal{S}^*$ that could have interacted with failed EXPAND operations in $\mathcal{S}$ are concurrent EXPAND operations that contended with failed EXPAND operations for the same nodes. But the fact that these EXPAND operations are in $\mathcal{S}^*$ implies that they succeeded in $\mathcal{S}$, and thus must also succeed in $\mathcal{S}^*$ when all contending expansions are removed. □

It thus suffices to study algorithm $\mathcal{A}'$ under sanitized asynchronous schedules. Our next goal is to map any sanitized asynchronous schedule $\mathcal{S}$ of $\mathcal{A}'$ to a sequential schedule that produces the same final system configuration as $\mathcal{S}$. Any asynchronous schedule already totally orders the updates to any single variable in an amoebot's public memory and the occupancy of any single node; here, we focus on ordering entire action executions. Denote the (relevant) executions of α' in $\mathcal{S}$ as pairs (A_i, α_i'), where amoebot A_i executes α_i'. Construct a directed graph D with nodes $\{(A_1, \alpha_1'), \ldots, (A_k, \alpha_k')\}$ representing all executions of α' in $\mathcal{S}$ and directed edges $(A_i, \alpha_i') \to (A_j, \alpha_j')$ for $i \neq j$ if and only if one of the following hold:

1. Both (A_i, α_i') and (A_j, α_j') lock some amoebot B in their LOCK operations and (A_j, α_j') is the first execution to lock B after B is unlocked by (A_i, α_i').
2. The nodes occupied by A_i at the start of α_i' and by A_j at the start of α_j' are adjacent and (A_i, α_i') is the last execution of A_i to execute an action of $\mathcal{A}$ before the LOCK operation of (A_j, α_j') completes.
3. (A_j, α_j') is the first execution to EXPAND into some node v after v is vacated by a CONTRACT operation in (A_i, α_i').

Lemma 16 *The directed graph D corresponding to the executions of α' in a sanitized asynchronous schedule of $\mathcal{A}'$ is a directed, acyclic graph (DAG).*

Proof We will show that for any edge $(A_i, \alpha_i') \to (A_j, \alpha_j')$ in D, (A_i, α_i') completes its LOCK operation before (A_j, α_j') does. This immediately implies that D is acyclic; otherwise, the LOCK operations of any two executions in a cycle of D must complete both before and after each other, a contradiction.

First suppose that $(A_i, \alpha_i') \to (A_j, \alpha_j')$ is an edge in D by Rule 1, i.e., both executions lock an amoebot B and (A_j, α_j') is the first execution to lock B after B is unlocked by (A_i, α_i'). Clearly, A_j can only lock B after A_i has unlocked B and A_i can only unlock B after it locks B in its own LOCK operation. Since these operations all involve message transfers requiring positive time, (A_i, α_i') must complete its LOCK operation before (A_j, α_j') does.

Next suppose that $(A_i, \alpha_i') \to (A_j, \alpha_j')$ is an edge in D by Rule 2, i.e., the nodes occupied by A_i and A_j at the start of their respective actions are adjacent and (A_i, α_i') is the last execution of A_i to execute an action of $\mathcal{A}$ before the LOCK operation of (A_j, α_j') completes. Any execution of an action of $\mathcal{A}$ in (A_i, α_i') must start after its LOCK operation completes; thus, (A_i, α_i') must complete its LOCK operation before (A_j, α_j') does.

Finally, suppose that $(A_i, \alpha_i') \to (A_j, \alpha_j')$ is an edge in D by Rule 3, i.e., (A_j, α_j') is the first execution to EXPAND into some node v after v is vacated by a CONTRACT operation in (A_i, α_i'). It suffices to consider the case where the LOCK operation of (A_i, α_i') does not lock A_j; otherwise, there exists a directed path of Rule 1 edges from (A_i, α_i') to (A_j, α_j') in D and the first case proves the claim. For (A_i, α_i') to not lock A_j, A_j cannot be a neighbor of A_i at the time the LOCK operation of (A_i, α_i') starts. We know that the LOCK operation of (A_i, α_i') is successful, so A_i is locked and occupies v until the start of its CONTRACT operation out of v. But A_j must occupy a node adjacent to v at the start of (A_j, α_j') and must succeed in its own LOCK operation in order to EXPAND into v. Thus, the LOCK operation of (A_j, α_j') cannot complete until after A_i has started contracting out of v, which occurs strictly after the completion of the LOCK operation of (A_i, α_i'). □

The following lemma compares the outcome of any sanitized asynchronous schedule of $\mathcal{A}'$ to a schedule where an execution of α' corresponding to a sink in the DAG D is removed. A property that will become important shortly is whether any EXPAND operation in the removed execution of α' could still be executed if it were placed in a different schedule. Formally, we say a system configuration C is *expansion-compatible* with an execution (A_i, α_i') if either (A_i, α_i') does not perform an EXPAND operation or the EXPAND operation executed by A_i in α_i' would succeed in C.

Lemma 17 *Consider any sanitized asynchronous schedule $\mathcal{S}$ of $\mathcal{A}'$ and let (A_i, α_i') be any sink in the corresponding DAG D. Let $\mathcal{S}_i^-$ be the asynchronous schedule obtained by removing all events associated with (A_i, α_i') from $\mathcal{S}$. Then $\mathcal{S}_i^-$ is valid and the final configuration reached by $\mathcal{S}_i^-$ is expansion-compatible with (A_i, α_i') and identical to that of $\mathcal{S}$ except for the amoebots locked by (A_i, α_i') in $\mathcal{S}$, which appear exactly as they did just after the* LOCK *operation of (A_i, α_i') completed in $\mathcal{S}$.*

Proof If (A_i, α_i') is the only execution in $\mathcal{S}$, then its removal yields an empty schedule $\mathcal{S}_i^-$ that trivially satisfies the lemma. So consider any action execution (A_j, α_j') in $\mathcal{S}$ with $j \neq i$. We first show that (A_j, α_j') must remain enabled

in $\mathcal{S}_i^-$; i.e., $A_j.\texttt{act} = \text{TRUE}$ at the time of this execution. This must have been the case in $\mathcal{S}$, so the only way for (A_j, α'_j) to not be enabled in $\mathcal{S}_i^-$ is if (A_i, α'_i) was responsible for enabling it in $\mathcal{S}$. But (A_i, α'_i) could only have updated $A_j.\texttt{act}$ to TRUE if (A_i, α'_i) locked A_j, implying that (A_j, α'_j) could not have started until after (A_i, α'_i) unlocked A_j. Thus, there must exist a directed path of Rule 1 edges in D from (A_i, α'_i) to (A_j, α'_j), contradicting the assumption that (A_i, α'_i) is a sink.

We next show that (A_j, α'_j) remains valid in $\mathcal{S}_i^-$. Let $\mathcal{L}_j(\mathcal{S})$ (resp., $\mathcal{L}_j(\mathcal{S}_i^-)$) denote the set of amoebots locked by (A_j, α'_j) in $\mathcal{S}$ (resp., in $\mathcal{S}_i^-$); we begin by showing $\mathcal{L}_j(\mathcal{S}) = \mathcal{L}_j(\mathcal{S}_i^-)$. First suppose that there is a directed path in D from (A_j, α'_j) to (A_i, α'_i). By the proof of Lemma 16, (A_j, α'_j) must complete its LOCK operation before (A_i, α'_i) does, implying that (A_j, α'_j) completes its LOCK operation before (A_i, α'_i) completes any operation. Since the timing of (A_j, α'_j) in $\mathcal{S}$ is preserved in $\mathcal{S}_i^-$, it follows that $\mathcal{L}_j(\mathcal{S}) = \mathcal{L}_j(\mathcal{S}_i^-)$. Now suppose that there is no directed path in D from (A_j, α'_j) to (A_i, α'_i). Then the amoebots locked by (A_j, α'_j) and (A_i, α'_i) in $\mathcal{S}$ must be disjoint by Rule 1, so certainly (A_j, α'_j) can lock any amoebot in $\mathcal{S}_i^-$ that it did in $\mathcal{S}$; i.e., $\mathcal{L}_j(\mathcal{S}) \subseteq \mathcal{L}_j(\mathcal{S}_i^-)$. But suppose to the contrary that (A_j, α'_j) is able to lock some additional amoebot B in $\mathcal{S}_i^-$ that it did not lock in $\mathcal{S}$. This is only possible if (A_i, α'_i) caused B to move out of the neighborhood of A_j in $\mathcal{S}$, either directly via a handover or indirectly by enabling some action of B involving a movement. In either case, A_i must have locked B before A_j did, implying the existence of a directed path of Rule 1 edges in D from (A_i, α'_i) to (A_j, α'_j). This once again contradicts the assumption that (A_i, α'_i) is a sink. So in any case, (A_j, α'_j) locks the same set of amoebots in $\mathcal{S}$ and $\mathcal{S}_i^-$.

After completing its LOCK operation, (A_j, α'_j) does one of two things. If $A_j.\texttt{awaken} = \text{TRUE}$, then it updates the activity bits of all the amoebots it locked to TRUE, updates its own awaken bit to FALSE, releases its locks, and aborts. Since $\mathcal{L}_j(\mathcal{S}) = \mathcal{L}_j(\mathcal{S}_i^-)$ and timing is preserved, these updates occur identically in $\mathcal{S}$ and $\mathcal{S}_i^-$.

Otherwise, if $A_j.\texttt{awaken} = \text{FALSE}$, A_j evaluates the guards of actions in $\mathcal{A}$; recall that these depend only on the positions, shapes, and public memories of the locked amoebots. Suppose to the contrary that there is an amoebot B locked by A_j whose position, shape, or public memory is different in $\mathcal{S}_i^-$ than it was in $\mathcal{S}$. Then (A_i, α'_i) must have locked B to perform the corresponding update in $\mathcal{S}$, implying that there is a directed path of Rule 1 edges from (A_i, α'_i) to (A_j, α'_j) in D, contradicting the assumption that (A_i, α'_i) is a sink. So the outcomes of the guard evaluations must be identical in $\mathcal{S}$ and $\mathcal{S}_i^-$.

Since (A_j, α'_j) is in the sanitized schedule $\mathcal{S}$, it must be relevant, and thus A_j must be $\mathcal{A}$-enabled in α'_j. Whichever enabled action of $\mathcal{A}$ is executed, any WRITE, CONTRACT, PULL, or PUSH operations involved must occur identically in $\mathcal{S}$ and $\mathcal{S}_i^-$ since the locked amoebots and their positions, shapes, and public memories are the same in both schedules. The only remaining possibility is that (A_j, α'_j) causes A_j to EXPAND into an adjacent node v in $\mathcal{S}$ that is occupied in $\mathcal{S}_i^-$, causing the EXPAND operation to fail in $\mathcal{S}_i^-$. This implies that (A_i, α'_i) causes A_i to CONTRACT out of v. But then $(A_i, \alpha'_i) \to (A_j, \alpha'_j)$ must be a Rule 3 edge in D, contradicting the assumption that (A_i, α'_i) is a sink. Thus, we conclude that $\mathcal{S}_i^-$ is valid and all action executions (A_j, α'_j) for which $j \neq i$ execute identically in $\mathcal{S}$ and $\mathcal{S}_i^-$.

Next, we show that the final configuration C_i^- reached by $\mathcal{S}_i^-$ is expansion-compatible with (A_i, α'_i). Suppose to the contrary that (A_i, α'_i) performs a successful EXPAND operation into a node v in $\mathcal{S}$ but the same expansion would fail in C_i^-. This is only possible if v is occupied by another amoebot in C_i^-. Since all executions other than (A_i, α'_i) are valid and execute identically in $\mathcal{S}$ and $\mathcal{S}_i^-$, another amoebot can only have come to occupy v in C_i^- if A_i vacated v in some later execution in $\mathcal{S}$. But A_i can only change its shape if it is locked, contradicting the assumption that (A_i, α'_i) is a sink in D by Rule 1. So v must be unoccupied in C_i^- and thus C_i^- is expansion-compatible with (A_i, α'_i).

It remains to show that the amoebots in $\mathcal{L}_i(\mathcal{S})$—i.e., those locked by (A_i, α'_i) in $\mathcal{S}$—appear in $\mathcal{S}_i^-$ exactly as they did after the LOCK operation of (A_i, α'_i) in $\mathcal{S}$. But this follows immediately from the assumption that (A_i, α'_i) is a sink: for an execution (A_j, α'_j) with $j \neq i$ to change the position, shape, or public memory of an amoebot $B \in \mathcal{L}_i(\mathcal{S})$, it would first have to lock B, implying that $(A_i, \alpha'_i) \to (A_j, \alpha'_j)$ is a directed edge in D. □

Lemma 17 allows us to prove the following central result. Here, we consider the expansion-robust variant $\mathcal{A}^E$ of $\mathcal{A}$ and the algorithm $(\mathcal{A}^E)'$ produced from $\mathcal{A}^E$ by the concurrency control framework. We denote the sole action of $(\mathcal{A}^E)'$ as $(\alpha^E)'$. Given the initial configuration C_0 of $\mathcal{A}$, configuration C_0^E is its extension with expand flags $A.\texttt{flag}_p = \text{FALSE}$ for all amoebots A and ports p; the initial configuration $(C_0^E)'$ of $(\mathcal{A}^E)'$ further extends C_0^E by adding $A.\texttt{act} = \text{TRUE}$ and $A.\texttt{awaken} = \text{FALSE}$ for all amoebots A.

Lemma 18 *For any finite sanitized asynchronous schedule $\mathcal{S}$ of $\mathcal{A}'$ starting in C'_0, there exists a sequential schedule of $(\mathcal{A}^E)'$ starting in $(C_0^E)'$ that reaches a final configuration that is identical to that of $\mathcal{S}$, modulo amoebots' expand flags, with the exception that the set of amoebots A with $A.\texttt{act} = \text{TRUE}$ or $A.\texttt{awaken} = \text{TRUE}$ is a superset of those in the final configuration reached by $\mathcal{S}$.*

Proof Consider any finite sanitized asynchronous schedule $\mathcal{S}$ of $\mathcal{A}'$ starting in C_0' and let D be its corresponding DAG (Lemma 16). We argue by induction on k, the number of executions of α' in $\mathcal{S}$, that any sequential ordering of the executions of α' in $\mathcal{S}$ consistent with a topological ordering of D can be extended to a sequential schedule $\bar{\mathcal{S}}$ of $(\mathcal{A}^E)'$ starting in $(C_0^E)'$ satisfying the lemma. Specifically, we construct $\bar{\mathcal{S}}$ by replacing executions of α' in $\mathcal{S}$ that execute some action $\alpha_i \in \mathcal{A}$ with corresponding executions of $(\alpha^E)'$ that execute action $\alpha_i^E \in \mathcal{A}^E$. We then suitably pad $\bar{\mathcal{S}}$ with executions of $(\alpha^E)'$ that execute action α_0^E (as defined in Algorithm 3) so that the set of amoebots A with $A.\texttt{act} = \text{TRUE}$ or $A.\texttt{awaken} = \text{TRUE}$ in the final configuration reached by $\bar{\mathcal{S}}$ is a superset of those in the final configuration reached by $\mathcal{S}$.

The lemma trivially holds for $k = 0$, so suppose the lemma holds for any sanitized asynchronous schedule of $\mathcal{A}'$ with $k \geq 0$ executions of α'. Let $\mathcal{S}$ be any sanitized asynchronous schedule of $\mathcal{A}'$ starting in C_0' consisting of $k+1$ executions of α', let C be the final configuration it reaches, and let (A_i, α_i') be any sink in the corresponding DAG D. By Lemma 17, the sanitized asynchronous schedule $\mathcal{S}_i^-$ obtained by removing all events associated with (A_i, α_i') from $\mathcal{S}$ is valid and reaches a final configuration C_i^- that is expansion-compatible with (A_i, α_i') and is identical to C except for the amoebots locked by (A_i, α_i') in $\mathcal{S}$, which appear exactly as they did just after the LOCK operation of (A_i, α_i') completed in $\mathcal{S}$. By the induction hypothesis, there exists a sequential schedule $\bar{\mathcal{S}}_i$ of $(\mathcal{A}^E)'$ starting in $(C_0^E)'$ that reaches a final configuration $\bar{C}_i$ identical to C_i^- (modulo amoebots' expand flags) with the exception that the set of amoebots A with $A.\texttt{act} = \text{TRUE}$ or $A.\texttt{awaken} = \text{TRUE}$ in $\bar{C}_i$ is a superset of those in C_i^-. This implies that $(A_i, (\alpha^E)_i')$ is enabled in $\bar{C}_i$ since (A_i, α_i') was enabled in C_i^- and they have the same guard: $A_i.\texttt{act} = \text{TRUE}$.

The amoebots $\mathcal{L}_i(\mathcal{S})$ locked by (A_i, α_i') in $\mathcal{S}$ must still be neighbors of A_i in $\bar{C}_i$ (i.e., at the end of $\bar{\mathcal{S}}_i$) by Lemma 17 and the induction hypothesis, but A_i may also have additional neighbors in $\bar{C}_i$ that were not originally present at the time of its LOCK operation in $\mathcal{S}$. Thus, we have $\mathcal{L}_i(\mathcal{S}) \subseteq \mathcal{L}_i(\bar{\mathcal{S}}_i)$. There are three cases for the behavior of (A_i, α_i'); in each, we construct a sequential schedule $\bar{\mathcal{S}}$ by combining $\bar{\mathcal{S}}_i$, the execution $(A_i, (\alpha^E)_i')$, and possibly additional executions of $(\alpha^E)'$ involving α_0^E whose final configuration satisfies the lemma.

Case 1. $A_i.\texttt{awaken} = \text{TRUE}$ both at the start of (A_i, α_i') in $\mathcal{S}$ and at the end of $\bar{\mathcal{S}}_i$. Let $\bar{\mathcal{S}}$ be the sequential schedule obtained by appending $(A_i, (\alpha^E)_i')$ to the end of $\bar{\mathcal{S}}_i$. Then in $\bar{\mathcal{S}}$, $(A_i, (\alpha^E)_i')$ updates $B.\texttt{act}$ to TRUE for all amoebots B that it locks, updates $A_i.\texttt{awaken}$ to FALSE, releases its locks, and aborts—just as (A_i, α_i') does in $\mathcal{S}$. Since $\mathcal{L}_i(\mathcal{S}) \subseteq \mathcal{L}_i(\bar{\mathcal{S}}_i)$, the only difference between the final configurations of $\mathcal{S}$ and $\bar{\mathcal{S}}$ (other than amoebots' expand flags) is that the latter may have additional amoebots with their activity or awaken bits set to TRUE, so the lemma holds.

Case 2. $A_i.\texttt{awaken} = \text{FALSE}$ at the start of (A_i, α_i') in $\mathcal{S}$ but $A_i.\texttt{awaken} = \text{TRUE}$ at the end of $\bar{\mathcal{S}}_i$. Let $\bar{\mathcal{S}}$ be the sequential schedule obtained by activating A_i twice at the end of $\bar{\mathcal{S}}_i$. The first activation has the same effect as Case 1, potentially yielding more amoebots with their activity or awaken bits set to TRUE. It also resets the awaken bit of A_i, yielding $A_i.\texttt{awaken} = \text{FALSE}$ in both $\mathcal{S}$ and $\bar{\mathcal{S}}_i + (A_i, (\alpha^E)_i')$. We address this in the following case.

Case 3. $A_i.\texttt{awaken} = \text{FALSE}$ both at the start of (A_i, α_i') in $\mathcal{S}$ and at the end of $\bar{\mathcal{S}}_i$. Since (A_i, α_i') is an execution of $\mathcal{S}$, a sanitized schedule, we know that (A_i, α_i') is relevant and thus must have an action $\alpha_j \in \mathcal{A}$ in $\mathcal{S}$ that is enabled by the amoebots $\mathcal{L}_i(\mathcal{S})$ locked in α_i'. Intuitively, we would like to construct the sequential schedule $\bar{\mathcal{S}}$ by appending $(A_i, (\alpha^E)_i')$ to the end of $\bar{\mathcal{S}}_i$, where the execution of $(\alpha^E)_i'$ involves the corresponding action $\alpha_j^E \in \mathcal{A}^E$. However, because $\bar{\mathcal{S}}_i$ involves expand flags and $\mathcal{L}_i(\mathcal{S}) \subseteq \mathcal{L}_i(\bar{\mathcal{S}}_i)$, it is not immediately obvious that α_j^E is enabled at the end of $\bar{\mathcal{S}}_i$ and can be executed to satisfy the lemma.

To this end, we first show that any amoebot $A_\ell \in \mathcal{L}_i(\bar{\mathcal{S}}_i) \backslash \mathcal{L}_i(\mathcal{S})$—i.e., those locked by $(A_i, (\alpha^E)_i')$ at the end of $\bar{\mathcal{S}}_i$ but not by (A_i, α_i') in $\mathcal{S}$—would be ignored in any guard evaluation and execution of α_j^E at the end of $\bar{\mathcal{S}}_i$ due to expand flags. Such an A_ℓ can only exist if there was some time t during the LOCK operation of (A_i, α_i') in $\mathcal{S}$ at which A_i and A_ℓ were not connected. But $A_\ell \in \mathcal{L}_i(\bar{\mathcal{S}}_i)$ implies that A_ℓ later became a neighbor of A_i, so consider the first event in $\mathcal{S}$ after time t at which A_i and A_ℓ are connected. This event must correspond to A_ℓ completing an expansion or handover and connecting to A_i, so in the corresponding action execution in $\bar{\mathcal{S}}_i$, A_ℓ must have updated the expand flag of any new port now connected to A_i to TRUE (see Lines 10, 11, and 13 of Algorithm 3). Any such expand flag can only be reset to FALSE in $\bar{\mathcal{S}}_i$ if A_ℓ or A_i execute another action in $\mathcal{S}$ after their connection event (see Line 5 or 7 of Algorithm 3, respectively). But A_ℓ (resp., A_i) cannot execute another action in $\mathcal{S}$ because Rule 2 (resp., Rule 1) of the DAG D would imply (A_i, α_i') is not a sink in D, a contradiction. Thus, any port p of any amoebot $A_\ell \in \mathcal{L}_i(\bar{\mathcal{S}}_i) \backslash \mathcal{L}_i(\mathcal{S})$ connected to A_i must have $A_\ell.\texttt{flag}_p = \text{TRUE}$ at the end of $\bar{\mathcal{S}}_i$.

We have established that if $\alpha_j \in \mathcal{A}$ was enabled in $\mathcal{S}$ for execution (A_i, α_i'), then any additional neighbors $\mathcal{L}_i(\bar{\mathcal{S}}_i) \backslash \mathcal{L}_i(\mathcal{S})$ locked by A_i at the end of $\bar{\mathcal{S}}_i$ cannot cause $\alpha_j^E \in \mathcal{A}^E$ to be disabled because their expand flags are TRUE and they are thus ignored. However, we must show the opposite for any original neighbor $A_\ell \in \mathcal{L}_i(\mathcal{S})$ at the end of $\bar{\mathcal{S}}_i$, i.e., that its expand flags do not cause A_i to ignore it and thus possibly disable α_j^E. This situation can be easily prevented using the $\alpha_0^E \in \mathcal{A}^E$ action as follows. For any port p of any amoebot $A_\ell \in \mathcal{L}_i(\mathcal{S})$ connected to A_i with $A_\ell.\texttt{flag}_p = \text{TRUE}$,

append an execution $(A_\ell, (\alpha^E)'_\ell)$ to the end of $\bar{\mathcal{S}}_i$ that involves an execution of α_0^E resetting $A_\ell.\texttt{flag}_p$ to FALSE. Complete the construction of the desired sequential schedule $\bar{\mathcal{S}}$ by appending the execution $(A_i, (\alpha^E)'_i)$ involving the execution of α_j^E. We have shown that this final execution in $\bar{\mathcal{S}}$ considers exactly the same neighborhood as (A_i, α'_i) did in $\mathcal{S}$, and thus because α_j is enabled in $\mathcal{S}$ so is α_j^E in $\bar{\mathcal{S}}$. This further guarantees that their respective executions make the same updates to the original variables of $\mathcal{A}$ and, by the expansion-compatibility ensured by Lemma 17, the same movement. The only differences between the final configurations reached by $\mathcal{S}$ and $\bar{\mathcal{S}}$ are (i) the added executions $(A_\ell, (\alpha^E)'_\ell)$ in $\bar{\mathcal{S}}$ for executing α_0^E might set additional activity and awaken bits to TRUE, and (ii) the final execution $(A_i, (\alpha^E)'_i)$ involving the execution of α_j^E will set the activity bit of any amoebot in $\mathcal{L}_i(\bar{\mathcal{S}}_i) \backslash \mathcal{L}_i(\mathcal{S})$ to TRUE. But these differences are exactly those allowed by the lemma, completing the induction. □

We now turn to the analysis of $\mathcal{A}'$ under sequential executions. Define a *sequential schedule* $\mathcal{S} = ((A_1, \alpha_1), (A_2, \alpha_2), \ldots)$ as the sequence of actions executed in a sequential execution, where α_i is the i-th action of $\mathcal{A}$ executed by the system and A_i is the amoebot that executed it. For a sequential schedule to be *valid*, α_i must be enabled for A_i in the configuration produced by executions $(A_1, \alpha_1), \ldots, (A_{i-1}, \alpha_{i-1})$, for all $i \geq 1$. Certainly, sequential schedules obfuscate various details that were made explicit in asynchronous schedules; e.g., they ignore the precise timing of message transmissions and movements. Although a single sequential schedule may in fact represent many possible sequential executions, this abstraction suffices for our purposes because the resulting system configurations are well-defined.

We first argue that sequential executions of $\mathcal{A}'$ terminate.

Lemma 19 *If every sequential schedule of $\mathcal{A}$ starting in C_0 is finite, then every sequential schedule of $\mathcal{A}'$ starting in C'_0 is also finite.*

Proof Suppose to the contrary that there exists an infinite sequential schedule $\mathcal{S}$ of $\mathcal{A}'$ starting in configuration C'_0. When ignoring the handling of amoebots' activity and awaken bits, any execution of action α' of $\mathcal{A}'$ either makes no change to the system configuration or makes changes identical to those of some action $\alpha \in \mathcal{A}$. First suppose that $\mathcal{S}$ contains an infinite number of executions of α' executing actions of $\mathcal{A}$. Then by constructing a sequential schedule comprising only these $\mathcal{A}$ action executions, we obtain an infinite schedule of $\mathcal{A}$ starting in C_0, a contradiction.

Suppose instead that $\mathcal{S}$ contains only a finite number of executions of α' executing actions of $\mathcal{A}$. Since there are only a finite number of such executions, there must exist a time t after which no amoebot is $\mathcal{A}$-enabled and the remaining infinite executions of α' only involve updates to amoebots' activity and awaken bits. Any activation of an amoebot A with $A.\texttt{awaken} = \text{TRUE}$ results in A setting the activity bits of its neighbors to TRUE—of which there can be at most a finite number Δ that depends on the assumed space variant—and resetting $A.\texttt{awaken}$ to FALSE (Steps 4–6). Otherwise, an activation of A with $A.\texttt{awaken} = \text{FALSE}$ must result in A resetting $A.\texttt{act}$ to FALSE since it is not $\mathcal{A}$-enabled (Step 10). Then the potential function $\Phi(C) = \sum_A (I_{A.\texttt{act}} + (\Delta + 1) I_{A.\texttt{awaken}})$ over system configurations C where $I_{A.\texttt{act}} \in \{0, 1\}$ (resp., $I_{A.\texttt{awaken}} \in \{0, 1\}$) is equal to 1 if and only if $A.\texttt{act} = \text{TRUE}$ (resp., $A.\texttt{awaken} = \text{TRUE}$) is both lower bounded by 0 and strictly decreasing after time t. Therefore, $\mathcal{S}$ can only contain a finite number of executions of α' only involving updates to amoebots' activity and awaken bits, a contradiction of $\mathcal{S}$ being infinite. □

We next establish a crucial property for characterizing configurations reachable by $\mathcal{A}'$.

Lemma 20 *Consider any sequential schedule $\mathcal{S}$ of $\mathcal{A}'$ starting in C'_0. Any amoebot that is $\mathcal{A}$-enabled in the final configuration reached by $\mathcal{S}$ either (i) is $\mathcal{A}'$-enabled or (ii) has an $\mathcal{A}'$-enabled neighbor B with $B.\texttt{awaken} = \text{TRUE}$.*

Proof Argue by induction on the length of $\mathcal{S} = ((A_1, \alpha_1), \ldots, (A_k, \alpha_k))$. If $k = 0$, then the lemma trivially holds since all amoebots A initially have $A.\texttt{act} = \text{TRUE}$ in C'_0 and thus are all $\mathcal{A}'$-enabled. So suppose the lemma holds for schedules of $\mathcal{A}'$ starting in C'_0 with any length $k \geq 0$, and consider any such schedule $\mathcal{S}_{k+1} = ((A_1, \alpha_1), \ldots, (A_{k+1}, \alpha_{k+1}))$ with length $k + 1$. For $1 \leq i \leq k + 1$, let C'_i be the final configuration reached by the subschedule $\mathcal{S}_i = ((A_1, \alpha_1), \ldots, (A_i, \alpha_i))$ of $\mathcal{S}_{k+1}$. Consider any $\mathcal{A}$-enabled amoebot A in C'_{k+1}.

We first suppose that A was also $\mathcal{A}$-enabled in C'_k. By the induction hypothesis, there are two cases to consider. If A is $\mathcal{A}'$-enabled in C'_k, then the only scenario in which $A.\texttt{act}$ is updated to FALSE is if $A = A_{k+1}$ and A is not $\mathcal{A}$-enabled (Step 10), contrary to our supposition. So A must also be $\mathcal{A}'$-enabled in C'_{k+1}, satisfying (i). Otherwise, A must have an $\mathcal{A}'$-enabled neighbor B with $B.\texttt{awaken} = \text{TRUE}$ in C'_k. The only scenario in which $B.\texttt{awaken}$ is updated to FALSE is if $B = A_{k+1}$ and B sets all of its neighbors' activity bits, including that of A, to TRUE (Steps 4–6). So either B satisfies (ii) by remaining an $\mathcal{A}'$-enabled neighbor with $B.\texttt{awaken} = \text{TRUE}$ or A is $\mathcal{A}'$-enabled in C'_{k+1}, satisfying (i).

Now suppose that A was not $\mathcal{A}$-enabled in C'_k; i.e., the execution of action α_{k+1} by amoebot A_{k+1} causes a change in the neighborhood of A such that A becomes $\mathcal{A}$-enabled in C'_{k+1}. Note that because A was not $\mathcal{A}$-enabled in C'_k, we must have $A_{k+1} \neq A$. If A and A_{k+1} were neighbors in C'_k, then A_{k+1} must update $A.\texttt{act}$ to TRUE during its execution of α_{k+1} (Step 16), satisfying (i). Otherwise, if A and A_{k+1} were not neighbors in C'_k, there are still two ways A_{k+1} could change the neighborhood of A by executing α_{k+1}. First, A_{k+1} could move into the neighborhood of A via an EXPAND or

PUSH; in this case, A_{k+1} remains $\mathcal{A}'$-enabled and updates its own awaken bit to TRUE (Steps 14 and 24), satisfying (ii). Second, A_{k+1} could update the memory of a neighbor B of A via a WRITE; in this case, A_{k+1} must also update $B.\texttt{act}$ and $B.\texttt{awaken}$ to TRUE (Steps 16 and 17), also satisfying (ii). □

The following lemma concludes our analysis of sequential executions.

Lemma 21 *For any configuration C' in which some sequential execution of $\mathcal{A}'$ starting in C'_0 terminates, there exists a sequential execution of $\mathcal{A}$ starting in C_0 that terminates in a configuration C identical to C', modulo activity and awaken bits.*

Proof Consider any valid sequential schedule $\mathcal{S}'$ of $\mathcal{A}'$ starting in C'_0 under which $\mathcal{A}'$ terminates and let C' be the configuration it terminates in. As in the proof of Lemma 19, the executions of action α' in $\mathcal{S}'$ involving $\mathcal{A}$ action executions form a valid sequential schedule $\mathcal{S}$ of $\mathcal{A}$ starting in C_0 that makes the same system configuration changes as $\mathcal{S}'$ w.r.t. the variables used in $\mathcal{A}$. So $\mathcal{S}$ reaches a configuration C that is equivalent to C' modulo amoebots' activity and awaken bits. Moreover, $\mathcal{S}$ must terminate in C; otherwise, there exists an $\mathcal{A}$-enabled amoebot in C that, by Lemma 20, implies there exists an $\mathcal{A}'$-enabled amoebot in C', contradicting our supposition that $\mathcal{A}'$ terminates in C'. □

It remains to show that all asynchronous schedules of $\mathcal{A}'$ are finite in a sense that they only require a finite amount of time.

Lemma 22 *If every sequential schedule of $\mathcal{A}$ starting in C_0 is finite, then every asynchronous schedule of $\mathcal{A}'$ starting in C'_0 is also finite.*

Proof Suppose to the contrary that there exists an infinite asynchronous schedule $\mathcal{S}$ of $\mathcal{A}'$ starting in C'_0.

First suppose that $\mathcal{S}$ contains only a finite number of relevant action executions. Then there exists an earliest time t after which no event associated with a relevant action execution is ever scheduled. Time t is well-defined because (i) every operation—and, by extension, every action execution—terminates in finite time and (ii) there can be at most a finite number of irrelevant action executions initiated before time t due to the fact that there are a finite number of amoebots, each amoebot executes at most one action per time, and any non-simultaneous events in $\mathcal{S}$ are at least one time unit apart. Since $\mathcal{S}$ is infinite and there always exists at least one active amoebot, there must exist an infinite number of action executions initiated after time t and they must all be irrelevant. Recall that, by Lemma 13, there are three types of irrelevant executions: those whose LOCK operation fails, those whose LOCK operation succeeds but that have $A.\texttt{awaken} = \text{FALSE}$ and are $\mathcal{A}$-disabled, and those whose LOCK operation succeeds but whose EXPAND operation fails.

It is easy to see that there must exist an execution of α' initiated after time t whose LOCK operation succeeds; otherwise, all action executions initiated after time t fail in their LOCK operation, a violation of the LOCK operation's deadlock freedom property.

We next argue that some execution of α' initiated after time t whose LOCK operation succeeds has $A.\texttt{awaken} = \text{FALSE}$ and is $\mathcal{A}$-enabled. Certainly, no execution of α' initiated after time t with a successful LOCK operation could have $A.\texttt{awaken} = \text{TRUE}$ as this execution would be relevant, contradicting our assumption on t. Any execution of α' that succeeds in its LOCK operation but is $\mathcal{A}$-disabled sets its amoebot's activity bit to FALSE, disabling α'. With a finite number of amoebots, there cannot be an infinite number of such executions.

So consider any execution (A, α') initiated after time t that succeeds in its LOCK operation, has $A.\texttt{awaken} = \text{FALSE}$, and is $\mathcal{A}$-enabled. This execution is irrelevant by supposition, so by Lemma 13, its EXPAND operation (say, into an adjacent node v) must fail. Convention 1 ensures that A could not have called EXPAND if it was expanded or if v was occupied at the time of the corresponding guard evaluation, and A cannot be involved in a movement initiated by some other amoebot because it is locked. The only way the EXPAND operation of (A, α') could fail is if another amoebot B successfully moves into v during an execution (B, α') that is concurrent with (A, α'). But if (B, α') succeeds in its movement operation, then all its operation executions must succeed by Lemma 13; therefore, (B, α') is a relevant execution with an event occurring after time t, again contradicting our assumption on t.

We conclude that $\mathcal{S}$ must in fact contain an infinite number of relevant action executions. Moreover, when ordering these relevant action executions by the time their LOCK operations complete, there is at most a finite number of time units—and thus a finite number of irrelevant action executions—between any two consecutive relevant action executions. Thus, every relevant action execution has a well-defined, finite start time.

Since $\mathcal{S}$ contains an infinite number of relevant action executions, its sanitized version $\mathcal{S}^*$ is also infinite. By Lemma 15 (which also holds for infinite schedules), $\mathcal{S}^*$ is a valid asynchronous schedule that changes the system configuration exactly as $\mathcal{S}$ does, except w.r.t. amoebots' activity bits. Let D be the infinite DAG corresponding to $\mathcal{S}^*$ (Lemma 16). We argue next that Lemmas 17 and 18 apply to any snapshot of $\mathcal{S}^*$ consistent with D.

Consider the schedule $\hat{\mathcal{S}}$ obtained by selecting the first $T \geq 1$ relevant action executions from $\mathcal{S}^*$ ordered by the time their LOCK operations complete; if multiple action executions complete their LOCK operations simultaneously, we may assume any unique, canonical ordering of these action executions. Since all edges of the DAG D of $\mathcal{S}^*$ are forward

in time w.r.t. completions of LOCK operations, $\hat{\mathcal{S}}$ forms a consistent snapshot of $\mathcal{S}^*$: for any edge $(A_i, \alpha_i') \to (A_j, \alpha_j')$ in D with execution (A_j, α_j') contained in $\hat{\mathcal{S}}$, we must have that (A_i, α_i') is also in $\hat{\mathcal{S}}$. This snapshot property ensures that any memory accesses, contractions, and handovers execute in $\hat{\mathcal{S}}$ in the same way as in $\mathcal{S}^*$ since these only depend on the amoebots locked in the LOCK operations. Moreover, if $\hat{\mathcal{S}}$ contains the first execution (A_j, α_j') to EXPAND into a position after it is vacated during an execution (A_i, α_i') in $\mathcal{S}^*$, then $(A_i, \alpha_i') \to (A_j, \alpha_j')$ is an edge in D by DAG Rule 3 and thus (A_i, α_i') is also contained in $\hat{\mathcal{S}}$ because it is a consistent snapshot. Hence, $\hat{\mathcal{S}}$ is a valid asynchronous schedule of $\mathcal{A}'$ starting in C_0'. By Lemmas 17 and 18, $\hat{\mathcal{S}}$ can be mapped to a valid sequential schedule of $(A^E)'$ starting in $(C_0^E)'$ that contains at least T action executions.

This immediately implies that if there exists an infinite asynchronous schedule of $\mathcal{A}'$ starting in C_0', then there must also exist an infinite sequential schedule of $(A^E)'$ starting in $(C_0^E)'$. Otherwise, there exists a value of T for which the above conversion fails, a contradiction. But this contradicts our original supposition: if every sequential schedule of $\mathcal{A}$ starting in C_0 is finite, then every sequential schedule of $\mathcal{A}^E$ starting in C_0^E is finite by the termination condition of Convention 3, which in turn implies that every sequential schedule of $(A^E)'$ starting in $(C_0^E)'$ is finite by Lemma 19. This concludes the proof. □

We are now ready to prove Theorem 11, concluding our analysis.

Proof of Theorem 11 Every sequential execution of $\mathcal{A}$ starting in C_0 terminates by supposition, so every asynchronous execution of $\mathcal{A}'$ starting in C_0' also terminates by Lemma 22. Consider any asynchronous schedule $\mathcal{S}$ of $\mathcal{A}'$ starting in C_0' and let C' be the configuration it terminates in. By Lemma 15, the sanitized asynchronous schedule $\mathcal{S}^*$ obtained from $\mathcal{S}$ is valid and terminates in a configuration C^* that is identical to C', except C^* may contain additional amoebots with TRUE activity bits. By Lemma 18, there exists a sequential schedule $\bar{\mathcal{S}}$ of $(\mathcal{A}^E)'$ starting in $(C_0^E)'$ that terminates in a configuration $(C^E)'$ that is identical to C^*, except $(C^E)'$ contains amoebots' expand flags and may also have additional amoebots with TRUE activity or awaken bits. Applying Lemma 21 to $\mathcal{A}^E$ implies that there exists some sequential schedule of $\mathcal{A}^E$ starting in C_0^E that terminates in a configuration C^E that is identical to $(C^E)'$, modulo amoebots' activity and awaken bits. Finally, because $\mathcal{A}$ satisfies Convention 3 by supposition, the correctness condition of expansion-robustness states that there exists a sequential execution of $\mathcal{A}$ starting in C_0 that terminates in a configuration C that is identical to C^E, modulo amoebots' expand flags. Therefore, C and C' are identical, modulo amoebots' activity and awaken bits, concluding the proof. □

5 Discussion and future work

An immediate application of the canonical amoebot model and its hierarchy of assumption variants is a systematic comparison of existing amoebot algorithms and their assumptions. For example, when comparing two recent amoebot algorithms for leader election using the canonical hierarchy, we find that among other problem-specific differences, Bazzi and Briones [5] assume an asynchronous adversary and common chirality while Emek et al. [34] assume a sequential adversary and assorted orientations. Such comparisons will provide valuable and comprehensive understanding of the state of amoebot literature and will facilitate clearer connections to related models of programmable matter.

The canonical amoebot model should also be extended to address *fault tolerance* and *self-stabilizing* algorithms. This work assumed that all amoebots are reliable, though crash faults have been previously considered in the amoebot model for specific problems [23,30]. Faulty amoebot behavior is especially challenging for lock-based concurrency control mechanisms which are prone to deadlock in the presence of crash faults. Additional modeling efforts will be needed to introduce a stable family of fault assumptions.

Finally, further study is needed on the design of concurrent amoebot algorithms. Amoebots' communication and movement raise many issues of concurrency, ranging from conflicts of movement to operating based on stale information. Our analysis of the Hexagon-Formation algorithm produced one set of algorithm-agnostic invariants that yield correct asynchronous behavior without the use of locks (Lemmas 3–5) while our concurrency control framework gives another set of sufficient conditions for obtaining correct behavior under an asynchronous adversary when using locks (Conventions 1–3).

Of the three conventions used by the concurrency control framework, expansion-robustness (Convention 3) is the most restrictive and technically difficult to verify, though it is easier to understand and verify than the original "monotonicity" convention [21] that it replaced. The framework's analysis relies on expansion-robustness to show that when an action execution is moved from its timing in an asynchronous schedule into the future where it is not concurrent with any other execution, it produces the same system configuration that it did originally, regardless of any new amoebots that may have moved into its neighborhood in the meantime. In that light, it is easy to see that *stationary* algorithms that do not use movement are trivially expansion-robust (Observation 9). These include many of the existing algorithms for leader election [5,19,28,32,37] and the recent algorithm for energy distribution [23]. However, many interesting collective behaviors for programmable matter require movement. We proved that the Hexagon-Formation algorithm is expansion-robust and compatible with the concurrency con-

trol framework (Theorem 10). Future work should investigate whether this is also true of other existing amoebot algorithms.

We emphasize that expansion-robustness is not simply a technicality of our approach but rather a general phenomenon for asynchronous amoebot systems. Imagine a cycle alternating between contracted amoebots and empty positions and an asynchronous execution where all amoebots, having no neighbors, expand concurrently. This forms a cycle of expanded amoebots. However, any sequence of these expansions would result in at least one amoebot seeing an already expanded neighbor at the start of its action execution, which may prohibit its expansion and stop the system from reaching the original outcome (an expanded cycle).

This discussion highlights two open questions. Do there exist amoebot algorithms that are not correct under an asynchronous adversary but are compatible with our concurrency control framework, establishing the necessity of lock-based approaches to concurrency control? What are the necessary conditions for amoebot algorithm correctness in spite of asynchrony, both with and without locks? We are hopeful that our approaches to concurrent algorithm design combined with answers to these open problems will advance the analysis of existing and future algorithms for programmable matter in the concurrent setting.

Author Contributions All authors contributed equally to the model design and formulation, algorithm design and analysis, and manuscript writing and review.

Funding Funding was provided by National Science Foundation under grand number CCF-1733680, Army Research Office under grand number MURI W911NF-19-1-0233, ASU Biodesign Institute, Momental Foundation under grand number Mistletoe Research Fellowship, Deutsche Forschungsgemeinschaft under grand number SCHE 1592/6-1.

Declarations

Conflict of interest The authors declare no competing interests.

Appendix A: Amoebot operation pseudocode

In this appendix, we give formal distributed pseudocode for the amoebot operations. Algorithm 5 details the communication operations (Sect. 2.2.1) and Algorithms 6 and 7 detail the movement operations (Sect. 2.2.2). One possible implementation of the concurrency control operations (Sect. 2.2.3) is given in [22].

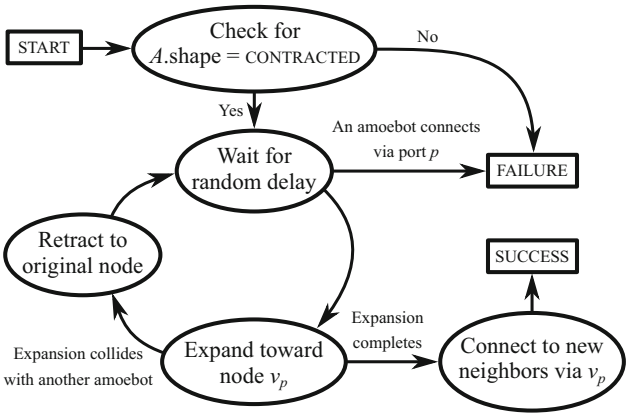


Fig. 7 Execution flow of the EXPAND operation with contention resolution for the calling amoebot A

Appendix B: Expansion contention resolution

Recall that when an amoebot's expansion collides with another movement, it must perform contention resolution such that exactly one contending amoebot succeeds in its expansion while all others fail. In this appendix, we detail and analyze one possible implementation of such a contention resolution scheme inspired by randomized backoff mechanisms for contention resolution in wireless networks [6,8,11]. We need one additional assumption: all amoebots know an upper bound T on the time required for an amoebot to complete any movement. For simplicity, we will assume geometric space (i.e., the triangular lattice G_Δ), though this mechanism would generalize to any bounded degree graph.

The execution flow of our contention resolution mechanism is shown in Fig. 7 and its pseudocode is given in Algorithm 8. When A detects a collision, it retracts to its original node and retries its expansion after waiting for a delay chosen uniformly at random from $[5T, 10T]$, where T is an upper bound on the time required for an amoebot to complete an expansion or retraction. In the remainder of this section, we verify the following claim.

Lemma 23 *Suppose a set of amoebots are contending to expand into the same node of G_Δ. If each amoebot waits for a delay chosen uniformly at random from $[5T, 10T]$ before its expansion attempt, then exactly one contender succeeds in $\mathcal{O}(\log n)$ attempts w.h.p.*[4]

Proof We first bound the probability that two amoebots A_1 and A_2 collide in their respective expansion attempts into the same node. For each amoebot $A_i \in \{A_1, A_2\}$, let t_i denote the start of its expansion attempt, d_i denote its random delay, and

[4] An event occurs *with high probability* (w.h.p.) if it occurs with probability at least $1 - 1/n^c$, where n is the number of amoebots in the system and $c > 0$ is a constant.

Algorithm 5 Communication Operations for Amoebot A

```
1: function CONNECTED(p)
2:     if there is a neighbor connected via port p then return TRUE.
3:     else return FALSE.
1: function CONNECTED( )
2:     Let k be the number of edges incident to the node(s) A occupies.
3:     Snapshot the connectivity status of each port p ∈ {1, ..., k}.
4:     Let c_p ← N_i if neighbor N_i is connected via port p and c_p ← FALSE otherwise.
5:     return [c_0, ..., c_{k−1}] ∈ {N_1, ..., N_8, FALSE}^k.
1: function READ(p, x)
2:     On being called:
3:         if p = ⊥ then return the value of x in the public memory of A; success.
4:         else if CONNECTED p then enqueue read_request(x) in the message buffer on p.
5:         else throw disconnect-failure.
6:     On receiving read_request(x) via port p′:
7:         Let x_val be the value of x in the public memory of A.
8:         Enqueue read_ack(x, x_val) in the message buffer on p′.
9:     On receiving read_ack(x, x_val) via port p:
10:        return x_val; success.
11:    On disconnection via port p:
12:        throw disconnect-failure.
1: function WRITE(p, x, x_val)
2:     On being called:
3:         if p = ⊥ then update the value of x in the public memory of A to x_val; return success.
4:         else if CONNECTED p then enqueue write_request(x, x_val) in the message buffer on p.
5:         else throw disconnect-failure.
6:     On write_request(x, x_val) being sent:
7:         return success.
8:     On disconnection via port p:
9:         throw disconnect-failure.
10:    On receiving write_request(x, x_val) via port p′:
11:        Update the value of x in the public memory of A to x_val.
```

e_i denote the duration of its expansion if it were to succeed. The start time t_i and expansion duration e_i are fixed a priori by the adversary while the delay d_i is chosen uniformly at random from the interval $[5T, cT]$, where $c > 5$ is a constant. So, in summary, amoebot $A_i \in \{A_1, A_2\}$ is waiting in the time interval $[t_i, t_i + d_i)$ and is expanding in the interval $[t_i + d_i, t_i + d_i + e_i]$. Thus, the expansions of amoebots A_1 and A_2 collide if and only if:

$$\begin{aligned}
&[t_1 + d_1, t_1 + d_1 + e_1] \cap [t_2 + d_2, t_2 + d_2 + e_2] \neq \emptyset \\
&\quad \iff (t_1 + d_1 + e_1 \geq t_2 + d_2) \\
&\quad \wedge (t_1 + d_1 \leq t_2 + d_2 + e_2) \\
&\quad \iff t_2 - t_1 - e_1 \leq d_1 - d_2 \leq t_2 - t_1 + e_2
\end{aligned}$$

This implies:

$$\begin{aligned}
&\Pr[A_1 \text{ and } A_2 \text{ collide} \mid t_1, t_2, e_1, e_2] \\
&\quad = \Pr[t_2 - t_1 - e_1 \leq d_1 - d_2 \leq t_2 - t_1 + e_2] \\
&\quad = \Pr[d_1 - d_2 \leq t_2 - t_1 + e_2] \\
&\qquad - \Pr[d_1 - d_2 \leq t_2 - t_1 - e_1]
\end{aligned}$$

Delays d_1 and d_2 are both uniform random variables over the interval $[5T, cT]$, so the difference $d_1 - d_2$ follows the symmetric triangular distribution with lower bound $(5-c)T$, upper bound $(c-5)T$, and mode 0. W.l.o.g., suppose $t_1 < t_2$. There are two cases: when $t_2 - t_1 - e_1 \leq 0$ and when $t_2 - t_1 - e_1 > 0$. If we have $t_2 - t_1 - e_1 \leq 0$, then:

$$\begin{aligned}
&\Pr[d_1 - d_2 \leq t_2 - t_1 + e_2] \\
&\qquad - \Pr[d_1 - d_2 \leq t_2 - t_1 - e_1] \\
&\quad = 1 - \frac{((c-5)T - (t_2 - t_1 + e_2))^2}{((c-5)T - (5-c)T)((c-5)T - 0)} \\
&\qquad - \frac{(t_2 - t_1 - e_1 - (5-c)T)^2}{((c-5)T - (5-c)T)(0 - (5-c)T)} \\
&\quad = \Big[2(c-5)^2T^2 - ((c-5)T - t_2 + t_1 - e_2)^2 \\
&\qquad -((c-5)T + t_2 - t_1 - e_1)^2\Big] / 2(c-5)^2T^2 \\
&\quad = \Big[2(c-5)^2T^2 - 2(c-5)^2T^2 + 2(c-5)Te_2 \\
&\qquad + 2(c-5)Te_1 - 2t_2^2 + 4t_2t_1 - 2t_2e_2 + 2t_2e_1 \\
&\qquad -2t_1^2 + 2t_1e_2 - 2t_1e_1 - e_2^2 - e_1^2\Big] / 2(c-5)^2T^2
\end{aligned}$$

Algorithm 6 Movement Operations for Amoebot A

```
1: function CONTRACT(v)
2:     On being called:
3:         if A.shape ≠ EXPANDED then throw shape-failure.
4:         else if A is involved in a handover then throw handover-failure.
5:         else release all connections via ports on v and begin contracting out of v.
6:     On completing the contraction:
7:         Update A.shape ← CONTRACTED; return success.
1: function EXPAND(p)
2:     Let v_p denote the node that port p faces.
3:     On being called:
4:         if A.shape ≠ CONTRACTED then throw shape-failure.
5:         else if A is involved in a handover then throw handover-failure.
6:         else if CONNECTED p then throw occupied-failure.
7:         else begin expanding into v_p.
8:     On collision with another amoebot:
9:         Perform contention resolution.
10:    On failing contention resolution:
11:        throw occupied-failure.
12:    On completing the expansion or on succeeding in contention resolution:
13:        Establish connections with any new neighbors adjacent to v_p.
14:        Update A.shape ← EXPANDED; return success.
```

$$
\begin{aligned}
&= [2(c-5)T(e_1+e_2) - 2(t_2-t_1)(e_2-e_1) \\
&\quad -2(t_2-t_1)^2 - e_1^2 - e_2^2\Big]/2(c-5)^2T^2 \\
&< \Big[4(c-5)T^2 + 2(c+1)T^2\Big]/2(c-5)^2T^2 \\
&= 3(c-3)/(c-5)^2,
\end{aligned}
$$

which is a constant probability whenever $c > \frac{13+\sqrt{33}}{2} \approx 9.373$. The upper bound follows from:

- Since T is the upper bound on the time required for an expansion, $e_1 + e_2 \leq 2T$.
- We assumed that $t_1 < t_2$, but we also have that if $t_2 > t_1 + d_1 + e_1$, then there cannot be a collision. Thus, $t_2 - t_1 \leq d_1 + e_1 \leq cT + T$ is a necessary condition for a collision. We also have that $-T \leq e_2 - e_1 \leq T$, so we conclude that $-2(t_2 - t_1)(e_2 - e_1) \leq 2(c+1)T^2$.
- The last three numerator terms are all nonpositive, and thus can be upper bounded by 0.

In the second case, if we have $t_2 - t_1 - e_1 > 0$, then:

$$
\begin{aligned}
&\Pr[d_1 - d_2 \leq t_2 - t_1 + e_2] \\
&\quad - \Pr[d_1 - d_2 \leq t_2 - t_1 - e_1] \\
&= 1 - \frac{((c-5)T - (t_2 - t_1 + e_2))^2}{((c-5)T - (5-c)T)((c-5)T - 0)} \\
&\quad - 1 + \frac{((c-5)T - (t_2 - t_1 - e_1))^2}{((c-5)T - (5-c)T)((c-5)T - 0)} \\
&= \Big[((c-5)T - t_2 + t_1 + e_1)^2 \\
&\quad -((c-5)T - t_2 + t_1 + e_2)^2\Big]/2(c-5)^2T^2 \\
&= [2(c-5)Te_1 - 2(c-5)Te_2 - 2t_2e_1 + 2t_2e_2 \\
&\quad +2t_1e_1 - 2t_1e_2 + e_1^2 - e_2^2\Big]/2(c-5)^2T^2 \\
&= [2(c-5)T(e_1-e_2) - 2(t_2-t_1)(e_1-e_2) \\
&\quad +e_1^2 - e_2^2\Big]/2(c-5)^2T^2 \\
&< \Big[2(c-5)T^2 + 2(c+1)T^2 + T^2\Big]/2(c-5)^2T^2 \\
&= (4c-7)/2(c-5)^2,
\end{aligned}
$$

which is a constant probability whenever $c > 6 + \sqrt{15/2} \approx 8.739$. Therefore, in any case, the probability that the expansions of A_1 and A_2 collide when their delays are drawn uniformly at random from the interval $[5T, cT]$ is at most a constant $p \in (0, 1)$ when $c > 9.373$.

Due to the structure of the triangular lattice G_Δ, at most six amoebots may be concurrently expanding into the same node. We now establish that pairwise collisions of any of these amoebots' expansions are independent. Given each expansion attempt's starting time and expansion duration—which are fixed by the asynchronous execution—the interval of expansion is entirely determined by the delay. Since each delay is drawn independently and uniformly from $[5T, cT]$, each pair of expansions' time intervals and thus also their collision is independent. So, fixing an amoebot A_1,

$$
\begin{aligned}
&\Pr\big[\text{an expansion of } A_1 \text{ succeeds} \mid t_1, e_1\big] \\
&\quad = \Pr[A_1 \text{ and } A_i \text{ do not collide} \mid t_1, t_i, e_1, e_i, \forall i \neq 1]
\end{aligned}
$$

Algorithm 7 Movement Operations for Amoebot A (cont.)

```
1: function PULL(p)
2:     Let v_p denote the node that port p faces.
3:     On being called:
4:         if A.shape ≠ EXPANDED then throw shape-failure.
5:         else if A is involved in a handover then throw handover-failure.
6:         else if ¬CONNECTED p then throw disconnect-failure.
7:         else enqueue pull_request() in the message buffer on p.
8:     On receiving pull_request() via port p′:
9:         if A.shape = CONTRACTED and A is not involved in a move then set m′ ← pull_ack().
10:        else set m′ ← pull_nack().
11:        Enqueue m′ in the message buffer on p′.
12:    On sending pull_ack():
13:        Begin expanding into v_p.
14:    On completing the expansion into v_p:
15:        Establish connections with any new neighbors adjacent to v_p.
16:        Update A.shape ← EXPANDED.
17:    On receiving pull_ack() via port p:
18:        Release all connections via ports on v_p except p and begin contracting out of v_p.
19:    On receiving pull_nack() via port p or on a disconnection via port p:
20:        throw nack-failure.
21:    On completing the contraction out of v_p:
22:        Update A.shape ← CONTRACTED; return success.

1: function PUSH(p)
2:     Let v_p denote the node that port p faces.
3:     On being called:
4:         if A.shape ≠ CONTRACTED then throw shape-failure.
5:         else if A is involved in a handover then throw handover-failure.
6:         else if ¬CONNECTED p then throw disconnect-failure.
7:         else enqueue push_request() in the message buffer on p.
8:     On receiving push_request() via port p′:
9:         if A.shape = EXPANDED and A is not involved in a move then set m′ ← push_ack().
10:        else set m′ ← push_nack().
11:        Enqueue m′ in the message buffer on p′.
12:    On sending push_ack():
13:        Release all connections via ports on v_p except p and begin contracting out of v_p.
14:    On completing the contraction out of v_p:
15:        Update A.shape ← CONTRACTED.
16:    On receiving push_ack() via port p:
17:        Begin expanding into v_p.
18:    On receiving push_nack() via port p or on a disconnection via port p:
19:        throw nack-failure.
20:    On completing the expansion into v_p:
21:        Establish connections with any new neighbors adjacent to v_p.
22:        Update A.shape ← EXPANDED; return success.
```

$$= \prod_{i \neq 1} (1 - \Pr[A_1 \text{ and } A_i \text{ collide} \mid t_1, t_i, e_1, e_i])$$
$$> (1-p)^5,$$

which is a constant probability since p is a constant probability.

In order to amplify this success probability for the desired w.h.p. result, we must establish independence of expansion attempts. We have already shown that pairwise collisions of amoebots' expansions are independent, but this is insufficient to establish the independence of subsequent expansion attempts. In particular, A_1 and A_2 may collide while concurrently attempting to expand, causing them both to retract before reattempting their expansions. A third amoebot A_3 could then expand and collide with A_1 or A_2 while they are retracting, causing A_3 to also retract; a fourth amoebot A_4 could then expand and collide with A_3 while it retracts, and so on. In the worst case, if the expansions of A_1 and A_2 collide at time t, these cascading expansion-retraction collisions can continue until time $t + 5T$; this occurs if all

Algorithm 8 EXPAND Operation with Contention Resolution for Amoebot A

```
1: function EXPAND(p)
2:     Let v_p denote the node that port p faces.
3:     On being called:
4:         if A.shape ≠ CONTRACTED then throw shape-failure.
5:         else if A is involved in a handover then throw handover-failure.
6:         else wait for a delay of 0.
7:     After waiting for a delay:
8:         if ¬CONNECTED p then begin expanding into v_p.
9:         else throw occupied-failure.
10:    On collision with another amoebot:
11:        Retract back out of v_p and wait for a delay chosen u.a.r. from [5T, 10T].
12:    On connection via port p:
13:        throw occupied-failure.
14:    On completing the expansion:
15:        Establish connections with any new neighbors adjacent to v_p.
16:        Update A.shape ← EXPANDED; return success.
```

retractions take the maximum time T and each amoebot A_i (for $i = 3, \ldots, 6$, since there are at most six competing amoebots) collides with retracting amoebot A_{i-1} at the last possible moment. However, it is impossible for these cascading collisions to continue after $t + 5T$: the earliest an amoebot could reattempt its expansion is after time $t + 5T$ if A_1 or A_2 immediately retracted after colliding at time t and then sampled the minimum possible delay, $5T$. Therefore, the expansion attempt of an amoebot A_i is independent of any of its previous attempts. So we have:

$$\begin{aligned}
&\Pr\left[\text{no amoebot successfully expands after } k \text{ attempts}\right] \\
&\quad \leq \Pr\left[A_1 \text{ collides in all } k \text{ expansion attempts}\right] \\
&\quad = \prod_{i=1}^{k} \left(1 - \Pr\left[A_1\text{'s } i\text{-th expansion attempt succeeds} \mid t_1^i, e_1^i\right]\right) \\
&\quad < \left(1 - (1-p)^5\right)^k
\end{aligned}$$

Setting $k = \ln n/(1-p)^5$, we have the probability that no amoebot successfully expands after k attempts is at most

$$\left(1 - (1-p)^5\right)^k \leq e^{-(1-p)^5 \cdot \frac{\ln n}{(1-p)^5}} = \frac{1}{n}$$

Once an amoebot's expansion succeeds, it connects to all its new neighbors causing any contending expansions to immediately fail. Therefore, we conclude that exactly one amoebot will successfully expand in at most $\ln n = \mathcal{O}(\log n)$ attempts with high probability. □

References

1. Altisen, K., Devismes, S., Dubois, S., Petit, F.: Introduction to Distributed Self-Stabilizing Algorithms, volume 8 of Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers (2019). https://doi.org/10.2200/S00908ED1V01Y201903DCT015
2. Andrés Arroyo, M., Cannon, S., Daymude, J.J., Randall, D., Richa, A.W.: A stochastic approach to shortcut bridging in programmable matter. Nat. Comput. **17**(4), 723–741 (2018). https://doi.org/10.1007/s11047-018-9714-x
3. Angluin, D., Aspnes, J., Diamadi, Z., Fischer, M.J., Peralta, R.: Computation in networks of passively mobile finite-state sensors. Distrib. Comput. **18**(4), 235–253 (2006). https://doi.org/10.1007/s00446-005-0138-3
4. Barrameda, E.M., Das, S., Santoro, N.: Deployment of asynchronous robotic sensors in unknown orthogonal environments. In: Algorithmic Aspects of Wireless Sensor Networks, volume 5389 of Lecture Notes in Computer Science, pp 125–140 (2008). https://doi.org/10.1007/978-3-540-92862-1_11
5. Bazzi, R.A., Briones, J.L.: Stationary and deterministic leader election in self-organizing particle systems. In: Stabilization, Safety, and Security of Distributed Systems, volume 11914 of Lecture Notes in Computer Science, pp 22–37 (2019). https://doi.org/10.1007/978-3-030-34992-9_3
6. Bender, M.A., Farach-Colton, M., He, S., Kuszmaul, B.C., Leiserson, C.E.: Adversarial contention resolution for simple channels. In: Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures, pp 325–332 (2005). https://doi.org/10.1145/1073970.1074023
7. Blackiston, D., Lederer, E., Kriegman, S., Garnier, S., Bongard, J., Levin, M.: A cellular platform for the development of synthetic living machines. Sci. Robot. **6**(52), eabf1571 (2021). https://doi.org/10.1126/scirobotics.abf1571
8. Cali, F., Conti, M., Gregori, E.: IEEE 802.11 protocol: design and performance evaluation of an adaptive backoff mechanism. IEEE J. Sel. Areas Commun. **18**(9), 1774–1786 (2000). https://doi.org/10.1109/49.872963
9. Cannon, S., Daymude, J.J., Gökmen, C., Randall, D., Richa, A.W.: A Local stochastic algorithm for separation in heterogeneous self-organizing particle systems. In: Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques (APPROX/RANDOM 2019), volume 145 of Leibniz International Proceedings in Informatics (LIPIcs), pp 54:1–54:22 (2019). https://doi.org/10.4230/LIPIcs.APPROX-RANDOM.2019.54
10. Cannon, S., Daymude, J.J., Randall, D., Richa, A.W.: A Markov chain algorithm for compression in self-organizing particle systems. In: Proceedings of the 2016 ACM Symposium on Principles

of Distributed Computing, pp 279–288 (2016). https://doi.org/10.1145/2933057.2933107
11. Capetanakis, J.: Tree algorithms for packet broadcast channels. IEEE Trans. Inform. Theory **25**(5), 505–515 (1979). https://doi.org/10.1109/TIT.1979.1056093
12. Chalk, C., Luchsinger, A., Martinez, E., Schweller, R., Winslow, A., Wylie, T.: Freezing simulates non-freezing tile automata. DNA Comput. Mol. Programm. **11145**, 155–172 (2018). https://doi.org/10.1007/978-3-030-00030-1_10
13. Chirikjian, G.S.: Kinematics of a metamorphic robotic system. In: Proceedings of the 1994 IEEE International Conference on Robotics and Automation, pp 449–455 (1994). https://doi.org/10.1109/ROBOT.1994.351256
14. D'Angelo, G., D'Emidio, M., Das, S., Navarra, A., Prencipe, G.: Asynchronous silent programmable matter achieves leader election and compaction. IEEE Access **8**, 207619–207634 (2020). https://doi.org/10.1109/ACCESS.2020.3038174
15. Das, S., Flocchini, P., Prencipe, G., Santoro, N., Yamashita, M.: The power of lights: synchronizing asynchronous robots using visible bits. In: 2012 IEEE 32nd International Conference on Distributed Computing Systems, pages 506–515, (2012). https://doi.org/10.1109/ICDCS.2012.71
16. Das, S., Flocchini, P., Prencipe, G., Santoro, N., Yamashita, M.: Autonomous Mobile Robots with Lights. Theoret. Comput. Sci. **609**(1), 171–184 (2016). https://doi.org/10.1016/j.tcs.2015.09.018
17. Daymude, J.J., Derakhshandeh, Z., Gmyr, R., Porter, A., Richa, A.W., Scheideler, C., Strothmann, T.: On the runtime of universal coating for programmable matter. Nat. Comput. **17**(1), 81–96 (2018). https://doi.org/10.1007/s11047-017-9658-6
18. Daymude, J.J., Gmyr, R., Hinnenthal, K., Kostitsyna, I., Scheideler, C., Richa, A.W.: Convex Hull Formation for Programmable Matter. In: Proceedings of the 21st International Conference on Distributed Computing and Networking, pp 2:1–2:10 (2020). https://doi.org/10.1145/3369740.3372916
19. Daymude, J.J., Gmyr, R., Richa, A.W., Scheideler, C., Strothmann, T.: Improved leader election for self-organizing programmable matter. In: Algorithms for Sensor Systems, volume 10718 of Lecture Notes in Computer Science, pp 127–140 (2017). https://doi.org/10.1007/978-3-319-72751-6_10
20. Daymude, J.J., Hinnenthal, K., Richa, A.W., Scheideler, C.: Computing by Programmable Particles. In: Flocchini, P., Prencipe, G., Santoro, N., (eds.) Distributed computing by mobile entities, volume 11340 of Lecture Notes in Computer Science, pp 615–681. Springer International Publishing, Cham (2019). https://doi.org/10.1007/978-3-030-11072-7_22
21. Daymude, J.J., Richa, A.W., Scheideler, C.: The canonical amoebot model: algorithms and concurrency control. In: 35th International Symposium on Distributed Computing (DISC 2021), volume 209 of Leibniz International Proceedings in Informatics (LIPIcs), pages 20:1–20:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik (2021). https://doi.org/10.4230/LIPIcs.DISC.2021.20
22. Daymude, J.J., Richa, A.W., Scheideler, C.: Local mutual exclusion for dynamic, anonymous, bounded memory message passing systems. In: 1st Symposium on Algorithmic Foundations of Dynamic Networks (SAND 2022), volume 221 of Leibniz International Proceedings in Informatics (LIPIcs), pages 12:1–12:19. Schloss Dagstuhl—Leibniz-Zentrum für Informatik (2022). https://doi.org/10.4230/LIPIcs.SAND.2022.12
23. Daymude, J.J., Richa, A.W., Weber, J.W.: Bio-inspired energy distribution for programmable matter. In: International Conference on Distributed Computing and Networking 2021, pages 86–95 (2021). https://doi.org/10.1145/3427796.3427835
24. Derakhshandeh, Z., Dolev, S., Gmyr, R., Richa, A.W., Scheideler, C., Strothmann, T.: Amoebot: a new model for programmable matter. In: Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures, pp 220–222, (2014). https://doi.org/10.1145/2612669.2612712
25. Derakhshandeh, Z., Gmyr, R., Richa, A.W., Scheideler, C., Strothmann, T.: 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 21:1–21:2 (2015). https://doi.org/10.1145/2800795.2800829
26. Derakhshandeh, Z., Gmyr, R., Richa, A.W., Scheideler, C., Strothmann, T.: Universal shape formation for programmable matter. In: Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures, pp 289–299 (2016). https://doi.org/10.1145/2935764.2935784
27. Derakhshandeh, Z., Gmyr, R., Richa, A.W., Scheideler, C., Strothmann, T.: Universal coating for programmable matter. Theoret. Comput. Sci. **671**, 56–68 (2017). https://doi.org/10.1016/j.tcs.2016.02.039
28. Derakhshandeh, Z., Gmyr, R., Strothmann, T., Bazzi, R., Richa, A.W., Scheideler, C.: Leader election and shape formation with self-organizing programmable matter. In: DNA Computing and Molecular Programming, volume 9211 of Lecture Notes in Computer Science, pp 117–132 (2015). https://doi.org/10.1007/978-3-319-21999-8_8
29. Di Luna, G.A., Flocchini, P., Gan Chaudhuri, S., Poloni, F., Santoro, N., Viglietta, G.: Mutual visibility by luminous robots without collisions. Inf. Comput. **254**(3), 392–418 (2017). https://doi.org/10.1016/j.ic.2016.09.005
30. Di Luna, G.A., Flocchini, P., Prencipe, G., Santoro, N., Viglietta, G.: Line recovery by programmable particles. In: Proceedings of the 19th International Conference on Distributed Computing and Networking, pp 4:1–4:10 (2018). https://doi.org/10.1145/3154273.3154309
31. Di Luna, G.A., Flocchini, P., Santoro, N., Viglietta, G., Yamauchi, Y.: Mobile RAM and shape formation by programmable particles. In: Euro-Par 2020: Parallel Processing, volume 12247 of Lecture Notes in Computer Science, pp 343–358 (2020). https://doi.org/10.1007/978-3-030-57675-2_22
32. Di Luna, G.A., Flocchini, P., Santoro, N., Viglietta, G., Yamauchi, Y.: Shape formation by programmable particles. Distrib. Comput. **33**(1), 69–101 (2020). https://doi.org/10.1007/s00446-019-00350-6
33. Dufoulon, F., Kutten, S., Moses Jr., W.K.: Efficient deterministic leader election for programmable matter. In: Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing, pp 103–113 (2021). https://doi.org/10.1145/3465084.3467900
34. Emek, Y., Kutten, S., Lavi, R., Moses Jr, W.K.: Deterministic leader election in programmable matter. In: 46th International Colloquium on Automata, Languages, and Programming (ICALP 2019), Leibniz International Proceedings in Informatics (LIPIcs), pp 140:1–140:14 (2019). https://doi.org/10.4230/LIPICS.ICALP.2019.140
35. Flocchini, P., Prencipe, G., Santoro, N., (eds.): Distributed Computing by Mobile Entities: Current Research in Moving and Computing, volume 11340 of Lecture Notes in Computer Science. Springer, Cham, (2019). https://doi.org/10.1007/978-3-030-11072-7
36. Flocchini, P., Santoro, N., Viglietta, G., Yamashita, M.: Rendezvous with constant memory. Theor. Comput. Sci. **621**, 57–72 (2016). https://doi.org/10.1016/j.tcs.2016.01.025
37. Gastineau, N., Abdou, W., Mbarek, N., Togni, O.: Distributed leader election and computation of local identifiers for programmable matter. In: Algorithms for Sensor Systems, volume 11410 of Lecture Notes in Computer Science, pp 159–179 (2019). https://doi.org/10.1007/978-3-030-14094-6_11
38. Gastineau, N., Abdou, W., Mbarek, N., Togni, O.: Leader election and local identifiers for three-dimensional programmable matter.

Concurr. Comput. Pract. Exp. (2020). https://doi.org/10.1002/cpe.6067
39. Hines, L., Petersen, K., Lum, G.Z., Sitti, M.: Soft actuators for small-scale robotics. Adv. Mater. **29**(13), 1603483 (2017). https://doi.org/10.1002/adma.201603483
40. Kriegman, S., Blackiston, D., Levin, M., Bongard, J.: A scalable pipeline for designing reconfigurable organisms. Proc. Natl. Acad. Sci. **117**(4), 1853–1859 (2020). https://doi.org/10.1073/pnas.1910837117
41. Liu, A.T., Yang, J.F., LeMar, L.N., Zhang, G., Pervan, A., Murphey, T.D., Strano, M.S.: Autoperforation of two-dimensional materials to generate colloidal state machines capable of locomotion. Faraday Discuss. **227**, 213–232 (2021). https://doi.org/10.1039/D0FD00030B
42. Michail, O., Spirakis, P.G.: Simple and efficient local codes for distributed stable network construction. Distrib. Comput. **29**(3), 207–237 (2016). https://doi.org/10.1007/s00446-015-0257-4
43. Nokhanji, N., Santoro, N.: Line Reconfiguration by programmable particles maintaining connectivity. In: Theory and Practice of Natural Computing, volume 12494 of Lecture Notes in Computer Science, pp 157–169 (2020). https://doi.org/10.1007/978-3-030-63000-3_13
44. Patitz, M.J.: An introduction to tile-based self-assembly and a survey of recent results. Nat. Comput. **13**(2), 195–224 (2014). https://doi.org/10.1007/s11047-013-9379-4
45. Piranda, B., Bourgeois, J.: Designing a quasi-spherical module for a huge modular robot to create programmable matter. Auton. Robot. **42**, 1619–1633 (2018). https://doi.org/10.1007/s10514-018-9710-0
46. Toffoli, T., Margolus, N.: Programmable matter: concepts and realization. Phys. D **47**(1–2), 263–272 (1991). https://doi.org/10.1016/0167-2789(91)90296-L
47. Woods, D., Chen, H.-L., Goodfriend, S., Dabby, N., Winfree, E., Yin, P.: Active self-assembly of algorithmic shapes and patterns in polylogarithmic time. In: Proceedings of the 4th Conference on Innovations in Theoretical Computer Science, pp 353–354, (2013). https://doi.org/10.1145/2422436.2422476
48. Xie, H., Sun, M., Fan, X., Lin, Z., Chen, W., Wang, L., Dong, L., He, Q.: Reconfigurable magnetic microrobot swarm: multimode transformation, locomotion, and manipulation. Sci. Robot. **4**(28), eaav8006 (2019). https://doi.org/10.1126/scirobotics.aav8006
49. Yang, J.F., Liu, P., Koman, V.B., Liu, A.T., Strano, M.S.: Synthetic cells: colloidal-sized state machines. In: Walsh, S.M., Strano, M.S. (eds.) Robotic Systems and Autonomous Platforms, Woodhead Publishing in Materials, pp. 361–386. Woodhead Publishing (2019). https://doi.org/10.1016/B978-0-08-102260-3.00015-9