The Canonical Amoebot Model: Algorithms and Concurrency Control

Joshua J. Daymude
Biodesign Center for Biocomputing, Security and Society,
Arizona State University, Tempe, AZ, USA

Andréa W. Richa
School of Computing and Augmented Intelligence, Arizona State University, Tempe, AZ, USA

Christian Scheideler
Department of Computer Science, Universität Paderborn, Germany

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 (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*. Using *hexagon formation* as a case study, 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. Together, the canonical amoebot model and these complementary approaches to concurrent algorithm design open new directions for distributed computing research on programmable matter.

2012 ACM Subject Classification Theory of computation → Distributed computing models; Theory of computation → Concurrency; Theory of computation → Self-organization

Keywords and phrases Programmable matter, self-organization, distributed algorithms, concurrency

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.20

Related Version *Full Version*: `https://arxiv.org/abs/2105.02420` [17]

Funding *Joshua J. Daymude*: NSF (CCF-1733680), U.S. ARO (MURI W911NF-19-1-0233), and the ASU Biodesign Institute.
Andréa W. Richa: NSF (CCF-1733680) and U.S. ARO (MURI W911NF-19-1-0233).
Christian Scheideler: DFG Project SCHE 1592/6-1.

Acknowledgements We thank Nicola Santoro and Paola Flocchini for their constructive feedback and Kristian Hinnenthal for his contributions to a preliminary version of this work.

© Joshua J. Daymude, Andréa W. Richa, and Christian Scheideler;
licensed under Creative Commons License CC-BY 4.0

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

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 [38]. 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, 8, 9, 10, 29, 35, 36, 37, 39]. We focus on the *amoebot model* [16, 19] which is motivated by micro- and nano-scale robotic systems with strictly limited computational and locomotive capabilities [32, 33, 34, 40, 41]. 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 in 2014, the amoebot model has been used to study both fundamental problems – such as leader election [5, 15, 23, 25, 28, 31] and shape formation [7, 20, 21, 24, 25] – as well as more complex behaviors including object coating [13, 22], convex hull formation [14], bridging [2], spatial sorting [6], and fault tolerance [18, 27].

With this growing body of amoebot model literature, it is evident that the model has evolved – and, to some extent, 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] and [28]), let alone between amoebot model results and those of related models (e.g., those from the established *autonomous mobile robots* literature [29]). 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. [24, 25, 27] 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 algorithm design in the canonical amoebot model: 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., [16, 20]) 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 the full generality of the canonical amoebot model which is limited by the framework's conventions.

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 (Section 2).
- General *sufficient conditions* for amoebot algorithm correctness under any adversary and an algorithm for *hexagon formation* that satisfies these conditions (Section 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 (Section 4).

1.1 Related Work

There are many theoretical models of programmable matter, ranging from the non-spatial *population protocols* [3] and *network constructors* [35] to the tile-based models of *DNA computing* and *molecular self-assembly* [8, 36, 39]. 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 [29] 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, 30] and limited communication capabilities in *luminous robots* [11, 12, 26].

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 [32, 33, 34, 40, 41]. 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 [24, 25, 27], 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 [16, 19]. 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 (Table 1), 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.

In the canonical amoebot model, programmable matter consists of individual, homogeneous computational elements called *amoebots*. Any structure that an amoebot system can form 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 potential variants with respect to space; the most common is the *geometric* variant that assumes $G = G_\Delta$, the triangular lattice (Figure 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 (Figure 1b). For a contracted amoebot, the unique node it occupies is considered to be its *head*; for an expanded amoebot, the node it has most recently come to occupy (due to movements) 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 *orientation*. An amoebot's orientation is persistent and depends on its *direction* – i.e., what 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 (see Table 1); Figure 1c gives an example of the *common chirality* variant where amoebots share a sense of clockwise rotation but have different directions.

Table 1 Summary of assumption variants in the canonical amoebot model, each organized from most to least general. Variants marked with $*$ have been considered in existing literature, and variants marked with † are the focus of the algorithmic results in this work.

	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 round. Each round has an evaluation phase and an execution phase.
	k-Isolated	No amoebots within 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.

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 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, but is not so strong so as to collapse the hierarchy of orientation assumptions. More details on an amoebot's anatomy are given in Section 2.1.

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 Section 2.2 and their organization into algorithms is described in Section 2.3. We note that future publications may abstract away from the system layer and focus on the interface to the application layer.

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 Section 2.2), and a volatile *private memory* that is inaccessible

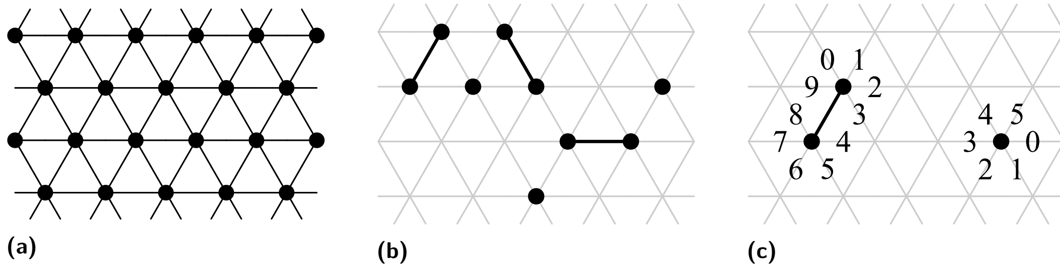

Figure 1 The Canonical Amoebot Model. (a) A section of the triangular lattice G_Δ used in the geometric variant; nodes of V are shown as black circles and edges of E are shown as black lines. (b) Expanded and contracted amoebots; G_Δ is shown in gray, and amoebots are shown as black circles. Amoebots 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 the system layer but read-writeable 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 $A.\texttt{lock} \in \{\text{TRUE}, \text{FALSE}\}$, and (iii) any variables used in the algorithm being run by the application layer.

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 execute one operation at a time. We summarize the communication, movement, and concurrency control operations below and in Table 2; see the full version [17] for pseudocode and execution details, including failure handling and contention resolution.

- The Connected operation checks for the presence of neighbors. Connected(p) returns TRUE iff there is a neighbor connected via port p. Connected() snapshots current port connectivity, including whether any neighbor is connected via multiple consecutive ports.
- The Read and Write operations exchange information in public memory. Read(p, x) issues a request to read the value of a variable x in the public memory of the neighbor connected via port p while Write(p, x, x_{val}) issues a request to update its value to x_{val}. If $p = \bot$, an amoebot's own public memory is accessed instead of a neighbor's.
- 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 neighbor it is vacating.

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()	$[c_0, \ldots, c_{k-1}] \in \{N_1, \ldots, N_8, \text{FALSE}\}^k$ where $c_p = N_i$ if N_i is the locally identified neighbor connected via port p and $c_p =$ FALSE otherwise
READ(p, x)	The value of x in the public memory of this amoebot if $p = \bot$ 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 $p = \bot$ 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()	Local identifiers of this amoebot and the neighbors that were locked
UNLOCK($\mathcal{L}$)	Confirmation that the amoebots of $\mathcal{L}$ were unlocked

- The LOCK operation encapsulates a variant of the mutual exclusion problem where an amoebot attempts to gain exclusive control over itself and its neighbors. A LOCK operation attempts to set the amoebot's and its neighbors' `lock` states from FALSE to TRUE but may fail if the amoebot or some neighbor is already locked or if there is a lock conflict. The UNLOCK operation releases locks obtained in a successful LOCK operation.

Each operation's message passing implementation is carefully designed so that (*i*) any operation execution terminates – either successfully or in failure – in finite time, and (*ii*) 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 operation executions. Combined with the blocking assumption, these design principles prohibit outgoing message buffer overflow and deadlocks in operation executions; see [17] for details.

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:

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

An action's *label* specifies its name. Its *guard* is a Boolean predicate determining whether an amoebot A can execute it based on the ports A has connections on – 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 private 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 – as an assumption for this work – *reliably*, meaning there are no crash or Byzantine faults.[1] An amoebot is said to be *active* if its application layer is executing an action and is *idle* otherwise. An amoebot can begin executing an action if and only if it is idle; 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 $\alpha_i : g_i \to ops_i$ 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 idle; otherwise, A chooses the highest-priority enabled action $\alpha_i : g_i \to ops_i$ – where action priorities are set by the algorithm – and executes the operations and private computation specified by ops_i. Recall from Section 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 idle. An action execution *fails* if any of its operations' executions result in a failure that is not addressed with error handling and *succeeds* otherwise.

As is standard in the distributed computing literature (see, e.g., [1]), we assume an *adversary* (or *daemon*) controls the timing of amoebot activations and the resulting 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; *k-isolated*, in which any set of amoebots except those occupying nodes of G_Δ within distance k can be simultaneously active; *synchronous*, in which time is discretized into "rounds" and in each round any set of amoebots can simultaneously execute one action each; and *asynchronous*, in which any amoebot can be active at any time. For synchronous concurrency, we further assume that each round 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 an amoebot may never be activated unless it is the only one with an enabled action. An algorithm execution is said to *terminate* if eventually all amoebots are idle and no amoebot is enabled; note that since an amoebot can only become enabled if something changes in its neighborhood, termination is permanent. Formal notions of algorithm runtime should be defined on a per-adversary basis; e.g., a standard synchronous round would apply to synchronous adversaries, while a strongly fair sequential adversary could use rounds that complete once every amoebot has been activated at least once.

In this paper, we focus on unfair sequential and asynchronous adversaries. In the sequential setting, an active amoebot knows that all other amoebots are idle; thus, its guard evaluations must be correct since the corresponding Connected and Read operations cannot fail or have their results be outdated due to concurrent changes in the system. In the asynchronous setting, however, these issues may lead to guards being evaluated incorrectly, causing disabled actions to be executed or enabled actions to be skipped. We address these issues in two ways, justifying the formulation of algorithms in terms of actions. In Section 3, we present an algorithm whose actions are carefully designed so that their guards are always evaluated correctly under any adversary. In Section 4, we present a concurrency control framework that uses locks to ensure that guards can be evaluated correctly even in the asynchronous setting.

[1] As we discuss in Section 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.

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 formulated in terms of actions as specified in Section 2.3. Our analysis establishes a general set of sufficient conditions for amoebot algorithm correctness under unfair asynchronous adversaries: (*i*) correctness under any 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 any 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*.[2] Following the sequential algorithm given by Derakhshandeh et al. [16, 20], the basic idea of our Hexagon-Formation algorithm is to form a hexagon by extending a spiral of amoebots counter-clockwise from the seed.

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. The amoebot system first self-organizes as a spanning forest rooted at the seed amoebot using their `parent` ports. Follower amoebots follow their parents until reaching the surface of retired amoebots that have already found their place in the hexagon. They then become roots, traversing the surface of retired amoebots clockwise. Once they connect to a retired amoebot's `dir` port, they also retire and set their `dir` port to the next position of the hexagon. Algorithm 1 describes Hexagon-Formation in terms of actions. We assume that if multiple actions are enabled for an amoebot, the enabled action with smallest index is executed. For conciseness and clarity, we write action guards as logical statements as opposed to their implementation with Read and Connected operations. 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 $B.\texttt{parent}$.

We begin our analysis of the Hexagon-Formation algorithm by showing it is correct under any sequential adversary.[3] All omitted proofs can be found in [17].

▶ **Lemma 1.** *Any unfair sequential execution of the Hexagon-Formation algorithm terminates with the amoebot system forming a hexagon.*

We next consider unfair asynchronous executions, the most general of all possible adversaries. 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.

[2] Note that the assumption of a unique seed amoebot immediately collapses the hierarchy of orientation assumptions since it can impose its own local orientation on the rest of the system via a simple broadcast.

[3] Although the related algorithm of Derakhshandeh et al. has already been analyzed in the sequential setting [16, 20], Hexagon-Formation must be proved correct with respect to its action formulation.

Algorithm 1 Hexagon-Formation for Amoebot A.

1: $\alpha_1 : (A.\texttt{state} \in \{\textsc{idle}, \textsc{follower}\}) \wedge (\exists B \in N(A) : B.\texttt{state} \in \{\textsc{seed}, \textsc{retired}\}) \to$
2: $\quad$ $\textsc{Write}(\bot, \texttt{parent}, \textsc{null})$.
3: $\quad$ $\textsc{Write}(\bot, \texttt{state}, \textsc{root})$.
4: $\quad$ $\textsc{Write}(\bot, \texttt{dir}, \textsc{GetNextDir}(\text{counter-clockwise}))$.
5: $\alpha_2 : (A.\texttt{state} = \textsc{idle}) \wedge (\exists B \in N(A) : B.\texttt{state} \in \{\textsc{follower}, \textsc{root}\}) \to$
6: $\quad$ Find a port p for which $\textsc{Connected}(p) = \textsc{true}$ and $\textsc{Read}(p, \texttt{state}) \in \{\textsc{follower}, \textsc{root}\}$.
7: $\quad$ $\textsc{Write}(\bot, \texttt{parent}, p)$.
8: $\quad$ $\textsc{Write}(\bot, \texttt{state}, \textsc{follower})$.
9: $\alpha_3 : (A.\texttt{shape} = \textsc{contracted}) \wedge (A.\texttt{state} = \textsc{root}) \wedge (\forall B \in N(A) : B.\texttt{state} \neq \textsc{idle})$
10: $\quad \wedge (\exists B \in N(A) : (B.\texttt{state} \in \{\textsc{seed}, \textsc{retired}\}) \wedge (B.\texttt{dir} \text{ is connected to } A)) \to$
11: $\quad$ $\textsc{Write}(\bot, \texttt{dir}, \textsc{GetNextDir}(\text{clockwise}))$.
12: $\quad$ $\textsc{Write}(\bot, \texttt{state}, \textsc{retired})$.
13: $\alpha_4 : (A.\texttt{shape} = \textsc{contracted}) \wedge (A.\texttt{state} = \textsc{root}) \wedge (\text{the node adjacent to } A.\texttt{dir} \text{ is empty}) \to$
14: $\quad$ $\textsc{Expand}(A.\texttt{dir})$.
15: $\alpha_5 : (A.\texttt{shape} = \textsc{expanded}) \wedge (A.\texttt{state} \in \{\textsc{follower}, \textsc{root}\}) \wedge (\forall B \in N(A) : B.\texttt{state} \neq \textsc{idle})$
16: $\quad \wedge (A \text{ has a tail-child } B : B.\texttt{shape} = \textsc{contracted}) \to$
17: $\quad$ **if** $\textsc{Read}(\bot, \texttt{state}) = \textsc{root}$ **then** $\textsc{Write}(\bot, \texttt{dir}, \textsc{GetNextDir}(\text{counter-clockwise}))$.
18: $\quad$ Find a port $p \in \textsc{TailChildren}()$ s.t. $\textsc{Read}(p, \texttt{shape}) = \textsc{contracted}$.
19: $\quad$ Let p' be the label of the tail-child's port that will be connected to p after the pull handover.
20: $\quad$ $\textsc{Write}(p, \texttt{parent}, p')$.
21: $\quad$ $\textsc{Pull}(p)$.
22: $\alpha_6 : (A.\texttt{shape} = \textsc{expanded}) \wedge (A.\texttt{state} \in \{\textsc{follower}, \textsc{root}\}) \wedge (\forall B \in N(A) : B.\texttt{state} \neq \textsc{idle})$
23: $\quad \wedge (A \text{ has no tail-children}) \to$
24: $\quad$ **if** $\textsc{Read}(\bot, \texttt{state}) = \textsc{root}$ **then** $\textsc{Write}(\bot, \texttt{dir}, \textsc{GetNextDir}(\text{counter-clockwise}))$.
25: $\quad$ $\textsc{Contract}(\textsc{tail})$.

▶ **Lemma 2.** *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.*

▶ **Lemma 3.** *For any asynchronous execution of the* Hexagon-Formation *algorithm, any execution of an enabled action is successful and unaffected by any concurrent action executions.*

Combined, Lemmas 1–3 directly imply that the Hexagon-Formation algorithm is both serializable and correct under any unfair asynchronous adversary.

▶ **Lemma 4.** *For any asynchronous execution of the* Hexagon-Formation *algorithm, there exists a sequential ordering of its action executions producing the same final configuration.*

▶ **Lemma 5.** *Any unfair asynchronous execution of the* Hexagon-Formation *algorithm terminates with the amoebot system forming a hexagon.*

Our analysis culminates in the following theorem.

▶ **Theorem 6.** *Assuming geometric space, assorted orientations, and constant-size memory, the* Hexagon-Formation *algorithm solves the hexagon formation problem under any adversary.*

We note that serializability and correctness under any asynchronous adversary (Lemmas 4–5) follow directly from Lemmas 1–3, independent of the specific details of Hexagon-Formation. Thus, these three lemmas establish a set of general sufficient conditions for amoebot algorithm correctness under asynchronous adversaries. We are hopeful that other existing amoebot algorithms can be adapted to satisfy these conditions under the new canonical model.

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, 6, 7, 14, 15, 20, 21, 22, 23, 28, 31].

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. We discuss interesting open problems regarding what algorithms are compatible with this framework and whether it can be extended beyond these conventions in Section 5.

The first of these conventions 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 idle, 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 is TRUE if and only if it has been locked in a LOCK operation. Whenever we talk about a system configuration in the following, we assume that it is legal.

▶ **Convention 1.** *All actions of an amoebot algorithm should be valid, i.e., for all its actions α and 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 idle.*

The second convention keeps an algorithm's actions simple by controlling the order and number of operations they perform.

▶ **Convention 2.** *Each action of an amoebot algorithm should structure its operations as:*

1. *A compute phase, during which an amoebot performs a finite amount of computation in private memory and a finite sequence of CONNECTED, READ, and WRITE operations.*
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.

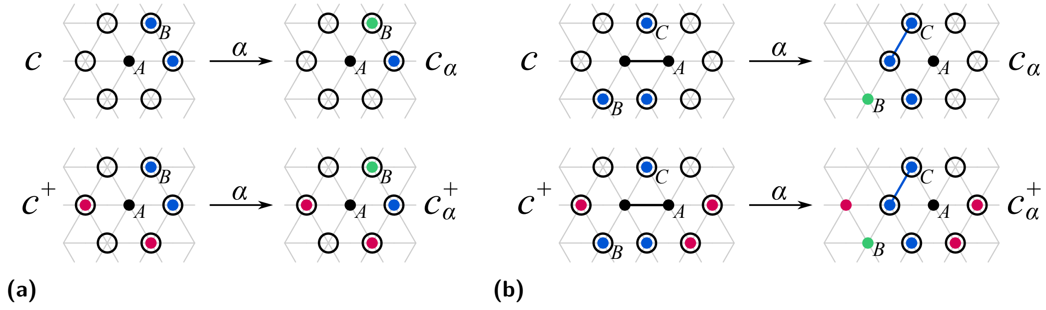

Figure 2 The monotonicity convention. In both examples, we examine the local configuration c of amoebot A, an extension c^+ of c, and the corresponding outcomes c_α and c_α^+ reached by sequential executions of α. The nodes with black circles denote the $N(\cdot)$ sets of nodes adjacent to A. Neighbors occupying the "non-extended neighborhood" $O(c)$ are shown in blue and neighbors occupying the "extended neighborhood" $O(c^+) \setminus O(c)$ are shown in pink. Monotonicity requires that the executions of α make the same updates: in (a), amoebot A updates the public memory of B, shown in green; in (b), amoebot A updates the public memory of B (green) and pulls neighbor C in a handover.

Convention 2 is similar in spirit to the *look-compute-move* paradigm used in the mobile robots literature (see, e.g., [29]), though message passing communication via Read and Write operations adds additional complexity in the amoebot model. 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 (Section 1.1).

Finally, due to our approach of using a serializability argument to show the correctness of algorithms using our framework, we need one last convention. This final convention is significantly more technical and limits the generality of the model more strictly than the first two, which we discuss further in Section 5. Consider any action $\alpha : g \to ops$ of an algorithm $\mathcal{A}$ being executed by an amoebot A. Recall that the guard g is a Boolean predicate based on the *local configuration* of A; i.e., the connected ports of A and the contents of the public memories of A and its neighbors. For a local configuration c of A, let $N(c) = (v_1, \ldots, v_k)$ be the nodes adjacent to A; note that $k = 6$ if A is contracted and $k = 8$ if A is expanded. Let $O(c) \subseteq N(c)$ be the nodes adjacent to A that are occupied by neighboring amoebots. Letting $M_\mathcal{A}$ denote the set of all possible contents of an amoebot's public memory w.r.t. algorithm $\mathcal{A}$, we can write c as a tuple $c = (c_0, c_1, \ldots, c_k) \in M_\mathcal{A} \times (M_\mathcal{A} \cup \{\text{NULL}\})^k$ where $c_0 \in M_\mathcal{A}$ is the public memory contents of A and, for $i \in \{1, \ldots, k\}$, $c_i \in M_\mathcal{A}$ is the public memory contents of the neighbor occupying node $v_i \in N(c)$ if $v_i \in O(c)$ and $c_i = \text{NULL}$ otherwise. Thus, we can express the guard g as a function $g : M_\mathcal{A} \times (M_\mathcal{A} \cup \{\text{NULL}\})^k \to \{\text{TRUE}, \text{FALSE}\}$.

A local configuration c is *consistent* if $c_i = c_j$ whenever nodes $v_i, v_j \in N(c)$ are occupied by the same expanded neighbor. Local configurations c and c' with $N(c) = N(c')$ are said to *agree on a subset* $S \subseteq N(c)$ if for all nodes $v_i \in S$, we have $c_i = c'_i$. A local configuration c^+ is an *extension* of local configuration c if c^+ is consistent and c and c^+ agree on the node A occupies and $O(c)$; intuitively, an extension c^+ has the same amoebots in the same positions with the same public memory contents as c, but may also have additional neighbors occupying nodes of $N(c) \setminus O(c)$. An extension c^+ of c is *expansion-compatible* with an execution of an action on c if any Expand operation by A in this execution would also succeed in c^+. An action $\alpha : g \to ops$ is *monotonic* (see Figure 2) if for any consistent local configuration c with $g(c) = \text{TRUE}$, any local configuration c_α reachable by an (isolated) execution of α on c, and any extension c^+ of c that is expansion-compatible with the execution reaching c_α and is reachable by a sequential execution of $\mathcal{A}$, $g(c^+) = \text{TRUE}$ and there exists a local configuration c_α^+ reachable by an execution of α on c^+ such that:

1. $N(c_\alpha) = N(c_\alpha^+)$; i.e., both executions of α leave A with the same set of adjacent nodes.
2. c_α and c_α^+ agree on the node A occupies and $O(c)$; i.e., both executions of α make identical updates w.r.t. A and its non-extended neighborhood.
3. c^+ and c_α^+ agree on $O(c^+) \setminus O(c)$; i.e., the execution of α on c^+ does not make any updates to the extended neighborhood of A.

▶ **Convention 3.** *All actions of an amoebot algorithm should be monotonic.*

4.1 The Concurrency Control Framework

Our *concurrency control framework* (Algorithm 2) 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 $A.\texttt{act} \in \{\text{TRUE}, \text{FALSE}\}$ 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 $g' = (A.\texttt{act} = \text{TRUE})$, 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 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* $A.\texttt{awaken} \in \{\text{TRUE}, \text{FALSE}\}$ to indicate that they should update their neighbors' activity bits in their next action. Initially, $A.\texttt{act} = \text{TRUE}$ and $A.\texttt{awaken} = \text{FALSE}$ 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 neighbors (Step 2). Given that it locks successfully, there are two cases. If $A.\texttt{awaken} = \text{TRUE}$, then A must have previously been involved in the operation of some acting amoebot that changed the neighborhood of A but could not 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). Otherwise, A obtains the necessary information to evaluate the guards of all actions in algorithm $\mathcal{A}$ (Steps 7–9). If no action from $\mathcal{A}$ 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 α' may also fail during an EXPAND operation if it conflicts with some other concurrent expansion (Step 14). In either case, A releases any locks it obtained (if any) and aborts (Steps 3 and 15). 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).

4.2 Analysis

The full technical analysis of the concurrency control framework is included in [17]. Here, we outline our argument and techniques, concluding with a statement of our main theorem.

Let $\mathcal{A}$ be any amoebot algorithm satisfying Conventions 1–3 and $\mathcal{A}'$ be the algorithm produced from $\mathcal{A}$ by our concurrency control framework (Algorithm 2). Our goal is to show that if any sequential execution of $\mathcal{A}$ terminates, then any asynchronous execution of $\mathcal{A}'$ must also terminate and will do so in a configuration that was reachable by a sequential execution of $\mathcal{A}$. This analysis is broken into two stages: analyzing $\mathcal{A}'$ under sequential executions and then leveraging a serialization argument to analyze $\mathcal{A}'$ under asynchronous executions. In each stage, we show that executions of $\mathcal{A}'$ are finite; i.e., they must terminate. Since all executions of $\mathcal{A}'$ terminate, it suffices to show that the final configurations reachable by asynchronous executions of $\mathcal{A}'$ are contained in those reachable by sequential executions of $\mathcal{A}'$ which in turn are contained in those reachable by sequential executions of $\mathcal{A}$.

The analysis of $\mathcal{A}'$ under sequential executions is relatively straightforward. We first show that every sequential execution of $\mathcal{A}'$ must terminate since otherwise there would exist an infinite sequential execution of $\mathcal{A}$, a contradiction. We then show that the added activity and awaken bits used in $\mathcal{A}'$ never cause it to terminate while there are still amoebots with actions of $\mathcal{A}$ left to perform. Together, these results imply that sequential executions of $\mathcal{A}'$ always terminate in configurations that sequential executions of $\mathcal{A}$ could also have terminated in.

The remainder of the analysis uses a serialization argument to show that asynchronous executions of $\mathcal{A}'$ always terminate in configurations that sequential executions of $\mathcal{A}'$ could also have terminated in. We model an asynchronous execution as a mapping of events (associated with action and operation executions) to ideal wall-clock times defined by an adversary. Given any asynchronous execution of $\mathcal{A}'$, we first *sanitize* it of all events associated with "irrelevant" action executions that do not affect the system configuration. We then argue that the sanitized execution remains valid (i.e., it is possible for some execution of $\mathcal{A}'$ to produce the events in the sanitized version) and that the sanitized execution changes the system configuration in exactly the same way as the original execution does.

Finally, we show that a sanitized execution can be *serialized*: there exists a sequential ordering of the action executions in the sanitized execution that produces the same final configuration as the sanitized execution. To do so, we construct a graph on the action executions in the sanitized schedule where directed edges represent causal relationships. By proving that the resulting graph is directed and acyclic, we obtain a partial order on the action executions. Since Convention 3 is satisfied, this partial order can be harnessed to obtain the desired serialization: a sequential execution of $\mathcal{A}'$ that reaches the same final configuration as the original asynchronous execution did. All together, we obtain the culminating theorem:

▶ **Theorem 7.** *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. Let C_0 be any initial configuration for $\mathcal{A}$ and let C_0' be its extension for $\mathcal{A}'$ with $A.\texttt{act} = \text{TRUE}$ and $A.\texttt{awaken} = \text{FALSE}$ for all amoebots A. If every sequential execution of $\mathcal{A}$ starting in C_0 terminates, then every asynchronous execution of $\mathcal{A}'$ starting in C_0' terminates in a configuration that some sequential execution of $\mathcal{A}$ starting in C_0 also terminates in.*

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 the two state-of-the-art 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. [28] 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.

Another direction would be to extend the canonical amoebot model to address *fault tolerance*. This work assumed that all amoebots are reliable, though crash faults have been previously considered in the amoebot model for specific problems [18, 27]. 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 algorithms for concurrent settings. The amoebot model's inclusion of memory, communication, and movement exacerbates issues of concurrency, ranging from operating based on stale information to conflicts of movement. 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 1–3) 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, monotonicity (Convention 3) is the most restrictive and technically difficult to verify. The serializability argument relies on it to show that when an action execution is removed from its timing in an asynchronous schedule into the future where it is not concurrent with any other execution, it makes exactly the same changes to the 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 trivially satisfy monotonicity. These include many of the existing algorithms for leader election [5, 15, 23, 25, 31] and the recent algorithm for energy distribution [18]. However, many interesting collective behaviors for programmable matter require movement, and it remains an open problem to identify if any of these algorithms satisfy monotonicity.

We emphasize that the monotonicity convention 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 serialization 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 critical open questions. Do there exist algorithms that are not correct under an asynchronous adversary but are compatible with our concurrency control framework? Are there other, less restrictive sufficient conditions for correctness in spite of asynchrony? 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.

References

1. Karine Altisen, Stéphane Devismes, Swan Dubois, and Franck Petit. *Introduction to Distributed Self-Stabilizing Algorithms*, volume 8 of *Synthesis Lectures on Distributed Computing Theory*. Morgan & Claypool Publishers, 2019.
2. Marta Andrés Arroyo, Sarah Cannon, Joshua J. Daymude, Dana Randall, and Andréa W. Richa. A stochastic approach to shortcut bridging in programmable matter. *Natural Computing*, 17(4):723–741, 2018.
3. Dana Angluin, James Aspnes, Zoë Diamadi, Michael J. Fischer, and René Peralta. Computation in networks of passively mobile finite-state sensors. *Distributed Computing*, 18(4):235–253, 2006.
4. Eduardo Mesa Barrameda, Shantanu Das, and Nicola Santoro. Deployment of asynchronous robotic sensors in unknown orthogonal environments. In *Algorithmic Aspects of Wireless Sensor Networks*, ALGOSENSORS 2008, pages 125–140, 2008.
5. Rida A. Bazzi and Joseph L. Briones. Stationary and deterministic leader election in self-organizing particle systems. In *Stabilization, Safety, and Security of Distributed Systems*, SSS 2019, pages 22–37, 2019.
6. Sarah Cannon, Joshua J. Daymude, Cem Gokmen, Dana Randall, and Andréa W. Richa. A local stochastic algorithm for separation in heterogeneous self-organizing particle systems. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, APPROX/RANDOM 2019, pages 54:1–54:22, 2019.
7. Sarah Cannon, Joshua J. Daymude, Dana Randall, and Andréa W. Richa. A Markov chain algorithm for compression in self-organizing particle systems. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, PODC 2016, pages 279–288, 2016.
8. Cameron Chalk, Austin Luchsinger, Eric Martinez, Robert Schweller, Andrew Winslow, and Tim Wylie. Freezing simulates non-freezing tile automata. In *DNA Computing and Molecular Programming*, DNA 2018, pages 155–172, 2018.
9. Gregory S. Chirikjian. Kinematics of a metamorphic robotic system. In *Proceedings of the 1994 IEEE International Conference on Robotics and Automation*, volume 1 of *ICRA 1994*, pages 449–455, 1994.
10. Gianlorenzo D'Angelo, Mattia D'Emidio, Shantanu Das, Alfredo Navarra, and Giuseppe Prencipe. Leader election and compaction for asynchronous silent programmable matter. In *Proceedings of the 19th International Conference on Autonomous Agents and MultiAgent Systems*, AAMAS 2020, pages 276–284, 2020.
11. Shantanu Das, Paola Flocchini, Giuseppe Prencipe, Nicola Santoro, and Masafumi Yamashita. The power of lights: Synchronizing asynchronous robots using visible bits. In *IEEE 32nd International Conference on Distributed Computing Systems*, ICDCS 2012, pages 506–515, 2012.
12. Shantanu Das, Paola Flocchini, Giuseppe Prencipe, Nicola Santoro, and Masafumi Yamashita. Autonomous mobile robots with lights. *Theoretical Computer Science*, 609:171–184, 2016.
13. Joshua J. Daymude, Zahra Derakhshandeh, Robert Gmyr, Alexandra Porter, Andréa W. Richa, Christian Scheideler, and Thim Strothmann. On the runtime of universal coating for programmable matter. *Natural Computing*, 17(1):81–96, 2018.
14. Joshua J. Daymude, Robert Gmyr, Kristian Hinnenthal, Irina Kostitsyna, Christian Scheideler, and Andréa W. Richa. Convex hull formation for programmable matter. In *Proceedings of the 21st International Conference on Distributed Computing and Networking*, ICDCN 2020, pages 2:1–2:10, 2020.

15 Joshua J. Daymude, Robert Gmyr, Andréa W. Richa, Christian Scheideler, and Thim Strothmann. Improved leader election for self-organizing programmable matter. In *Algorithms for Sensor Systems*, ALGOSENSORS 2017, pages 127–140, 2017.

16 Joshua J. Daymude, Kristian Hinnenthal, Andréa W. Richa, and Christian Scheideler. Computing by programmable particles. In *Distributed Computing by Mobile Entities: Current Research in Moving and Computing*, pages 615–681. Springer, 2019.

17 Joshua J. Daymude, Andréa W. Richa, and Christian Scheideler. The canonical amoebot model: Algorithms and concurrency control. Full version available online at `arXiv:2105.02420`, 2021.

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

19 Zahra Derakhshandeh, Shlomi Dolev, Robert Gmyr, Andréa W. Richa, Christian Scheideler, and Thim Strothmann. Brief announcement: Amoebot - a new model for programmable matter. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA 2014, pages 220–222, 2014.

20 Zahra Derakhshandeh, Robert Gmyr, Andréa W. Richa, Christian Scheideler, and Thim Strothmann. An algorithmic framework for shape formation problems in self-organizing particle systems. In *Proceedings of the Second Annual International Conference on Nanoscale Computing and Communication*, NANOCOM 2015, pages 21:1–21:2, 2015.

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

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

23 Zahra Derakhshandeh, Robert Gmyr, Thim Strothmann, Rida A. Bazzi, Andréa W. Richa, and Christian Scheideler. Leader election and shape formation with self-organizing programmable matter. In *DNA Computing and Molecular Programming*, DNA 2015, pages 117–132, 2015.

24 Giuseppe A. Di Luna, Paola Flocchini, Nicola Santoro, Giovanni Viglietta, and Yukiko Yamauchi. Mobile RAM and shape formation by programmable particles. In *Euro-Par 2020: Parallel Processing*, pages 343–358, 2020.

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

26 Giuseppe Antonio Di Luna, Paola Flocchini, Sruti Gan Chaudhuri, Federico Poloni, Nicola Santoro, and Giovanni Viglietta. Mutual visibility by luminous robots without collisions. *Information and Computation*, 254(3):392–418, 2017.

27 Giuseppe Antonio Di Luna, Paola Flocchini, Giuseppe Prencipe, Nicola Santoro, and Giovanni Viglietta. Line recovery by programmable particles. In *Proceedings of the 19th International Conference on Distributed Computing and Networking*, ICDCN 2018, pages 4:1–4:10, 2018.

28 Yuval Emek, Shay Kutten, Ron Lavi, and William K. Moses Jr. Deterministic leader election in programmable matter. In *46th International Colloquium on Automata, Languages, and Programming*, ICALP 2019, pages 140:1–140:14, 2019.

29 Paola Flocchini, Giuseppe Prencipe, and Nicola Santoro, editors. *Distributed Computing by Mobile Entities*. Springer International Publishing, Switzerland, 2019.

30 Paola Flocchini, Nicola Santoro, Giovanni Viglietta, and Masafumi Yamashita. Rendezvous with constant memory. *Theoretical Computer Science*, 621:57–72, 2016.

31 Nicolas Gastineau, Wahabou Abdou, Nader Mbarek, and Olivier Togni. Distributed leader election and computation of local identifiers for programmable matter. In *Algorithms for Sensor Systems*, ALGOSENSORS 2018, pages 159–179, 2019.

32 Lindsey Hines, Kirstin Petersen, Guo Zhan Lum, and Metin Sitti. Soft actuators for small-scale robotics. *Advanced Materials*, 29(13):1603483, 2017.

33 Sam Kriegman, Douglas Blackiston, Michael Levin, and Josh Bongard. A scalable pipeline for designing reconfigurable organisms. *Proceedings of the National Academy of Sciences*, 117(4):1853–1859, 2020.

34 Albert Tianxiang Liu, Jing Fan Yang, Lexy N. LeMar, Ge Zhang, Ana Pervan, Todd D. Murphey, and Michael S. Strano. Autoperforation of two-dimensional materials to generate colloidal state machines capable of locomotion. *Faraday Discussions*, 2021.

35 Othon Michail and Paul G. Spirakis. Simple and efficient local codes for distributed stable network construction. *Distributed Computing*, 29:207–237, 2016.

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

37 Benoit Piranda and Julien Bourgeois. Designing a quasi-spherical module for a huge modular robot to create programmable matter. *Autonomous Robots*, 42:1619–1633, 2018.

38 Tommaso Toffoli and Norman Margolus. Programmable matter: Concepts and realization. *Physica D: Nonlinear Phenomena*, 47(1):263–272, 1991.

39 Damien Woods, Ho-Lin Chen, Scott Goodfriend, Nadine Dabby, Erik Winfree, and Peng Yin. Active self-assembly of algorithmic shapes and patterns in polylogarithmic time. In *Proceedings of the 4th Conference on Innovations in Theoretical Computer Science*, ITCS 2013, pages 353–354, 2013.

40 Hui Xie, Mengmeng Sun, Xinjian Fan, Zhihua Lin, Weinan Chen, Lei Wang, Lixin Dong, and Qiang He. Reconfigurable magnetic microrobot swarm: Multimode transformation, locomotion, and manipulation. *Science Robotics*, 4(28):eaav8006, 2019.

41 Jing Fan Yang, Pingwei Liu, Volodymyr B. Koman, Albert Tianxiang Liu, and Michael S. Strano. Synthetic Cells: Colloidal-sized state machines. In Shawn M. Walsh and Michael S. Strano, editors, *Robotic Systems and Autonomous Platforms*, Woodhead Publishing in Materials, pages 361–386. Woodhead Publishing, 2019.

A Appendix: Concurrency Control Framework Pseudocode

Algorithm 2 Concurrency Control Framework for Amoebot A.

Input: Algorithm $\mathcal{A} = \{[\alpha_i : g_i \to ops_i] : i \in \{1, \dots, m\}\}$ 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 neighbors.
3: **catch** `lock-failure` **do** abort.
4: **if** $A.\texttt{awaken} = \text{TRUE}$ **then**
5: **for all** amoebots $B \in \mathcal{L}$ **do** WRITE $B.\texttt{act} \leftarrow \text{TRUE}$.
6: WRITE $A.\texttt{awaken} \leftarrow \text{FALSE}$, 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 $A.\texttt{act} \leftarrow \text{FALSE}$, 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 $A.\texttt{awaken} \leftarrow \text{TRUE}$.
15: **catch** `expand-failure` **do** UNLOCK($\mathcal{L}$) and abort.
16: **for all** amoebots $B \in \mathcal{L}$ **do** WRITE $B.\texttt{act} \leftarrow \text{TRUE}$.
17: **for all** $(B.x \leftarrow x_{val}) \in W_i$ **do** WRITE $B.x \leftarrow x_{val}$ and WRITE $B.\texttt{awaken} \leftarrow \text{TRUE}$.
18: **if** M_i is NULL or EXPAND **then** UNLOCK each amoebot in $\mathcal{L}$.
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 $A.\texttt{awaken} \leftarrow \text{TRUE}$ and $B.\texttt{awaken} \leftarrow \text{TRUE}$.
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 $B.\texttt{awaken} \leftarrow \text{TRUE}$.
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** $\mathcal{A}' = \{[\alpha' : g' \to ops']\}$.