Energy-Constrained Programmable Matter Under Unfair Adversaries

Jamison W. Weber
School of Computing and Augmented Intelligence, Arizona State University, Tempe, AZ, USA

Tishya Chhabra
School of Computing and Augmented Intelligence, Arizona State University, Tempe, AZ, USA

Andréa W. Richa
School of Computing and Augmented Intelligence & Biodesign Center for Biocomputing, Security and Society, Arizona State University, Tempe, AZ, USA

Joshua J. Daymude
School of Computing and Augmented Intelligence & Biodesign Center for Biocomputing, Security and Society, Arizona State University, Tempe, AZ, USA

Abstract

Individual modules of *programmable matter* participate in their system's collective behavior by expending energy to perform actions. However, not all modules may have access to the external energy source powering the system, necessitating a local and distributed strategy for supplying energy to modules. In this work, we present a general *energy distribution framework* for the *canonical amoebot model* of programmable matter that transforms energy-agnostic algorithms into energy-constrained ones with equivalent behavior and an $\mathcal{O}(n^2)$-round runtime overhead – even under an *unfair adversary* – provided the original algorithms satisfy certain conventions. We then prove that existing amoebot algorithms for *leader election* (ICDCN 2023) and *shape formation* (Distributed Computing, 2023) are compatible with this framework and show simulations of their energy-constrained counterparts, demonstrating how other unfair algorithms can be generalized to the energy-constrained setting with relatively little effort. Finally, we show that our energy distribution framework can be composed with the *concurrency control framework* for amoebot algorithms (Distributed Computing, 2023), allowing algorithm designers to focus on the simpler energy-agnostic, sequential setting but gain the general applicability of energy-constrained, asynchronous correctness.

2012 ACM Subject Classification Theory of computation → Distributed algorithms; Theory of computation → Self-organization

Keywords and phrases Programmable matter, amoebot model, energy distribution, concurrency

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2023.7

Related Version All proofs omitted from this paper due to space constraints can be found in:
Full Version: `https://arxiv.org/abs/2309.04898`

Supplementary Material *Software (AmoebotSim)*: `https://github.com/SOPSLab/AmoebotSim`
archived at `swh:1:dir:af373f8c717556382caba4a3abccc6ad749f03c3`

Funding This work is supported in part by National Science Foundation award CCF-2312537 and by Army Research Office MURI award #W911NF-19-1-0233.

1 Introduction

Programmable matter [34] is often envisioned as a material composed of simple, homogeneous modules that collectively change the system's physical properties based on environmental stimuli or user input. These modules participate in the system's overall collective behavior by expending energy to perform internal computation, communicate with their neighbors, and move. But as the number of modules per collective increases and individual modules

© Jamison W. Weber, Tishya Chhabra, Andréa W. Richa, and Joshua J. Daymude;
licensed under Creative Commons License CC-BY 4.0

are miniaturized from the centimeter/millimeter-scale [20, 22, 32] to the micro- and nano-scale [4, 16, 26], traditional methods of robotic power supply such as internal battery storage and tethering become infeasible. Many programmable matter systems instead make use of an external energy source accessible by at least one module and rely on *module-to-module power transfer* to supply the system with energy [6, 20, 23, 32]. This external energy can be supplied directly to modules in the form of electricity [20] or may be ambiently available as light, heat, sound, or chemical energy in the environment [27, 30]. Since energy may not be uniformly accessible to all modules in the system, a strategy for *energy distribution* – sharing energy among modules such that the system can achieve its desired function – is imperative.

Algorithmic theory for programmable matter – including population protocols [1], the nubot model [36], mobile robots [17], hybrid programmable matter [21], and the amoebot model [10, 12] – has largely ignored energy constraints, focusing instead on characterizing individual modules' necessary and sufficient capabilities for goal collective behaviors. Besides a few notable exceptions [16, 32], this literature only references energy to justify assumptions (e.g., why a system should remain connected [28]) and ignores the impact of energy usage and distribution on an algorithm's efficiency. In contrast, both programmable matter practitioners and the modular and swarm robotics literature incorporate energy constraints as influential aspects of algorithm design [2, 24, 29, 31, 35].

This gap motivated the prior Energy-Sharing algorithm for energy distribution [11] under the *amoebot model* of programmable matter [12]. When amoebots do not move and are activated *sequentially and fairly*, Energy-Sharing distributes any necessary energy to all n amoebots within at most $\mathcal{O}(n)$ rounds. Combined with the Forest-Prune-Repair algorithm introduced in the same work to repair energy distribution networks as amoebots move, it was suggested that any amoebot algorithm could be composed with these two to handle energy constraints, though this was only shown for one algorithm in simulation.

In this work, we introduce a general *energy distribution framework* that provably converts any energy-agnostic amoebot algorithm satisfying certain conventions into an *energy-constrained* version that exhibits the same system behavior while also distributing the energy amoebots need to meet the demands of their actions. In particular, we use the message passing-based *canonical amoebot model* [10] to address the challenges of *unfair* adversarial schedulers – the most general of all fairness assumptions – that can activate any amoebot that is able to perform an action regardless of how long others have been waiting to do the same. Under an unfair adversary, the prior Forest-Prune-Repair algorithm may not terminate, rendering it unusable for maintaining energy distribution networks. In contrast, energy-constrained algorithms produced by our framework not only terminate despite unfairness, but do so within an $\mathcal{O}(n^2)$-round overhead, where n is the number of amoebots in the system.

Our Contributions. We summarize our contributions as follows. We introduce the *energy distribution framework* that transforms any energy-agnostic amoebot algorithm $\mathcal{A}$ satisfying some basic conventions and a demand function δ specifying its energy costs into an energy-constrained algorithm $\mathcal{A}^\delta$ that provably exhibits equivalent behavior to $\mathcal{A}$, even under an unfair adversary, while incurring at most an $\mathcal{O}(n^2)$-round runtime overhead (Section 3). We then prove that both the Leader-Election-by-Erosion algorithm from [5] and the Hexagon-Formation algorithm from [10] satisfy the framework's conventions and show simulations of their energy-constrained counterparts produced by the framework (Section 4).

Finally, we prove that a particular class of "expansion-corresponding" algorithms that are compatible with the established *concurrency control framework* for amoebot algorithms [10] – including Leader-Election-by-Erosion and Hexagon-Formation– remain so after transformation

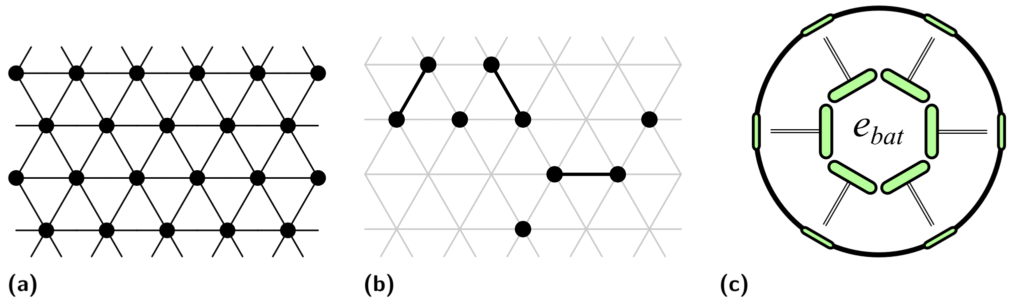


Figure 1 *The Amoebot Model.* (a) A section of the triangular lattice G_Δ used in the geometric space variant; nodes of V are shown as black circles and edges of E are shown as black lines. (b) Expanded and contracted amoebots; G_Δ is shown in gray and amoebots are shown as black circles. Amoebots with a black line between their nodes are expanded. (c) When modeling energy, each amoebot A has a battery $A.e_{bat}$ storing energy for its own use and for sharing with its neighbors.

by our energy distribution framework, establishing a general pipeline for lifting energy-agnostic, non-concurrent amoebot algorithms (which are easier to design and analyze) to the more realistic *energy-constrained, asynchronous setting* (Section 5).

2 Preliminaries

We begin with necessary background on the (canonical) amoebot model in Section 2.1 and our extensions for energy constraints in Section 2.2.

2.1 The Amoebot Model

In the *canonical amoebot model* [10], 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. Here, we adopt the geometric space variant in which $G = G_\Delta$, the triangular lattice (Figure 1a).

An amoebot has two *shapes*: CONTRACTED, meaning it occupies a single node in V, and EXPANDED, meaning it occupies a pair of adjacent nodes in V (Figure 1b). 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*. All results in this work allow for assorted orientations, meaning amoebots may disagree on both direction (which incident edge points "north") and chirality (clockwise vs. counter-clockwise rotation). Two amoebots occupying adjacent nodes are said to be *neighbors*. Although each amoebot is *anonymous*, lacking a unique identifier, an amoebot can locally identify its neighbors using their port labels. In particular, amoebots A and B connected via ports p_A and p_B know each other's orientations and labels for p_A and p_B.

Each amoebot has memory whose size is a model variant; all results in this work assume constant-size memories. An amoebot's memory consists of two parts: a persistent *public memory* that is only accessible to an amoebot algorithm via communication operations (defined next) and a volatile *private memory* that is directly accessible by amoebot algorithms for temporary variables, computation, etc. *Operations* define the programming interface for amoebot algorithms to communicate and move (see [10] for details):

- The CONNECTED operation tests the presence of neighbors. CONNECTED(p) returns TRUE if and only if there is a neighbor connected via port p.
- 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 occurs 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.

Amoebot algorithms are sets of *actions*, each of the form $\langle \textit{label} \rangle : \langle \textit{guard} \rangle \to \langle \textit{operations} \rangle$. 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.

An amoebot is *active* while executing an action and is *inactive* otherwise. An *adversary* controls the timing of amoebot activations and the resulting action executions, whose *concurrency* and *fairness* are assumption variants. In this work, we consider two concurrency variants: sequential, in which at most one amoebot can be active at a time; and asynchronous, in which any set of amoebots can be simultaneously active. We consider the most general fairness variant: unfair, in which the adversary may activate any enabled amoebot.

An amoebot algorithm's time complexity is evaluated in terms of *rounds* representing 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 $(t_i, t_{i+1}]$.

2.2 Extensions for Energy Modeling

In addition to the standard model, we introduce new assumptions and terminology specific to modeling energy in amoebot systems. We consider amoebot systems that are finite, initially connected, and contain at least one *source amoebot* with access to an external energy source. Although system connectivity is not generally required by the (canonical) amoebot model, it is necessary for sharing energy from a single source amoebot to the rest of the system via module-to-module power transfer. Each amoebot A has an *energy battery* denoted $A.e_{bat}$ with capacity $\kappa > 0$ representing energy that A can use to perform actions or share with its neighbors (Figure 1c). In this paper, we assume $\kappa = \Theta(1)$ is a fixed integer constant that does not scale with the number of amoebots n, but all results in this paper would hold even if $\kappa = \mathcal{O}(n)$. Source amoebots can harvest energy directly into their batteries while those without access depend on their neighbors to share with them. In either case, we assume an amoebot transfers at most a single unit of energy per activation.[1] For modeling purposes, we treat $A.e_{bat}$ as a variable stored in the public memory of A. An amoebot A harvesting energy from an external source can be expressed as $\textsc{Write}(\bot, e_{bat}, \textsc{Read}(\bot, e_{bat}) + 1)$ and likewise an amoebot A transferring energy to a neighbor B connected via a port p is a pair of operations $\textsc{Write}(\bot, e_{bat}, \textsc{Read}(\bot, e_{bat}) - 1)$ and $\textsc{Write}(p, e_{bat}, \textsc{Read}(p, e_{bat}) + 1)$.

The energy costs for an amoebot algorithm $\mathcal{A} = \{[\alpha_i : g_i \to ops_i] : i \in \{1, \ldots, m\}\}$ are given by a *demand function* $\delta : \mathcal{A} \to \{1, 2, \ldots, \kappa\}$; i.e., an amoebot must use $\delta(\alpha_i)$ energy to execute action α_i. Energy is incorporated into actions $\alpha_i \in \mathcal{A}$ by (1) including $A.e_{bat} \geq \delta(\alpha_i)$ in each guard g_i and (2) setting $\textsc{Write}(\bot, e_{bat}, \textsc{Read}(\bot, e_{bat}) - \delta(\alpha_i))$ as the first operation of ops_i to spend the corresponding amount of energy. An amoebot A is *deficient* w.r.t. an action $\alpha_i \in \mathcal{A}$ if $A.e_{bat} < \delta(\alpha_i)$. An amoebot algorithm $\mathcal{A}$ is *energy-agnostic* if it is not associated with a demand function δ and is *energy-constrained* (w.r.t. δ) otherwise.

The remainder of this paper is dedicated to transforming amoebot algorithms that were designed for the energy-agnostic setting into algorithms with equivalent behavior in the energy-constrained setting w.r.t. any valid demand function under an unfair adversary.

3 A General Framework for Energy-Constrained Algorithms

Amoebot algorithm designers prove the correctness of their algorithms with respect to a *safety* condition (related to the desired system behavior) and a *liveness* condition (ensuring that until this behavior is achieved, some amoebot can make progress towards it). Moving from energy-agnosticism to respecting energy constraints does not affect safety, but may threaten liveness. Some amoebot that was critical to achieving progress in the energy-agnostic setting may now be deficient under the constraints of actions' energy costs, deadlocking the system until it is provided with sufficient energy. Since not all amoebots have access to an external energy source, simply waiting to recharge is not an option. There must be an active strategy for energy distribution embedded in any energy-constrained algorithm.

Instead of placing the burden on algorithm designers to create bespoke implementations of energy distribution for each algorithm, we introduce a general *energy distribution framework*. This framework transforms energy-agnostic algorithms $\mathcal{A}$ that terminate under an unfair adversary and satisfy certain *conventions* into algorithms $\mathcal{A}^\delta$ that are energy-constrained w.r.t. any valid demand function δ and retain their unfair correctness. We give a narrative description and pseudocode for our framework in Section 3.1 and analyze it in Section 3.2.

3.1 The Energy Distribution Framework

Our *energy distribution framework* (Algorithm 1) takes as input any energy-agnostic amoebot algorithm $\mathcal{A} = \{[\alpha_i : g_i \to ops_i] : i \in \{1, \ldots, m\}\}$ and demand function $\delta : \mathcal{A} \to \{1, 2, \ldots, \kappa\}$ and outputs an energy-constrained algorithm $\mathcal{A}^\delta = \{[\alpha_i^\delta : g_i^\delta \to ops_i^\delta] : i \in \{1, \ldots, m\}\} \cup \{\alpha_{\textsc{EnergyDistribution}}\}$, where actions α_i^δ are energy-constrained versions of the original actions and $\alpha_{\textsc{EnergyDistribution}}$ is a new action that handles energy distribution. Algorithm $\mathcal{A}^\delta$ will achieve the same system behavior as algorithm $\mathcal{A}$ so long as $\mathcal{A}$ satisfies certain conventions:

[1] One could assume that the battery capacity $\kappa > 0$ is any positive real number and that the energy demands are $\delta : \mathcal{A} \to (0, \kappa]$. However, this generality complicates our analysis without meaningfully extending our results, so we make the simplifying assumption that there exists a fundamental unit of energy that divides all action demands $\delta(\alpha_i)$ and the battery capacity κ.

Table 1 Variables used in the Energy Distribution Framework.

Variable	Notation	Domain	Initialization
Forest State	`state`	{SOURCE, IDLE, ACTIVE, ASKING, GROWING, PRUNING}	SOURCE if source amoebot; IDLE otherwise.
Parent Pointer	`parent`	$\{\text{NULL}, 0, \ldots, 9\}$[2]	NULL
Battery Energy	e_{bat}	$\{0, 1, 2, \ldots, \kappa\}$	0

▶ **Definition 1.** *An energy-agnostic amoebot algorithm $\mathcal{A}$ is energy-compatible – i.e., it is compatible with the energy distribution framework – if every (unfair) sequential execution of $\mathcal{A}$ terminates and $\mathcal{A}$ satisfies Conventions 1–3 (defined below).*

Our first two conventions are taken directly from the analogous concurrency control framework for amoebot algorithms [10]. The first convention requires an algorithm's actions to execute successfully in isolation, allowing the framework to ignore invalid actions like attempting to READ on a disconnected port or EXPAND when already expanded. Formally, we define a *system configuration* as the mapping of amoebots to the node(s) they occupy and the contents of each amoebot's public memory. Throughout the remainder of this paper, we assume configurations are *legal*; i.e., they meet the requirements of the amoebot model.

▶ **Convention 1** (Validity). *All actions α of an amoebot algorithm $\mathcal{A}$ should be valid, i.e., for all (legal) 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 their operations, similar to the "look-compute-move" paradigm in the mobile robots literature [17].

▶ **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 the canonical amoebot model's concurrency control operations, LOCK and UNLOCK.*

Our third and final convention is specific to the energy distribution framework. Recall from Section 2.2 that we consider amoebot systems that are initially connected. This last convention requires an algorithm to maintain system connectivity throughout its execution, ensuring that every amoebot has a path to a source amoebot with access to external energy.

▶ **Convention 3** (Connectivity). *All system configurations reachable by any sequential execution of an amoebot algorithm $\mathcal{A}$ starting in a connected configuration must also be connected.*

Framework Overview. With the conventions defined, we now describe how the energy distribution framework (Algorithm 1) transforms an energy-compatible algorithm $\mathcal{A}$ and a demand function $\delta : \mathcal{A} \to \{1, 2, \ldots, \kappa\}$ into an energy-constrained algorithm $\mathcal{A}^\delta$ with

[2] Amoebots maintain one port per incident lattice edge (see Section 2.1), so an expanded amoebot has ten ports despite having a maximum of eight neighbors.

Algorithm 1 Energy Distribution Framework for Amoebot A.

```
    Input: An energy-compatible algorithm $\mathcal{A} = \{[\alpha_i : g_i \to ops_i] : i \in \{1, \ldots, m\}\}$ and a demand
    function $\delta : \mathcal{A} \to \{1, 2, \ldots, \kappa\}$.
 1: for each action $[\alpha_i : g_i \to ops_i] \in \mathcal{A}$ do construct action $\alpha_i^\delta : g_i^\delta \to ops_i^\delta$ as:
 2:     Set $g_i^\delta \leftarrow \big(g_i \wedge (A.e_{bat} \geq \delta(\alpha_i)) \wedge (\forall B \in N(A) \cup \{A\} : B.\texttt{state} \notin \{\textsc{idle}, \textsc{pruning}\})\big)$.
 3:     Set $ops_i^\delta \leftarrow$ "Do:
 4:         $\textsc{Write}(\bot, e_{bat}, \textsc{Read}(\bot, e_{bat}) - \delta(\alpha_i))$.
 5:         Execute the compute phase of $ops_i$.
 6:         if the movement phase of $ops_i$ contains a movement operation $M_i$ then
 7:             if $M_i$ is $\textsc{Contract}()$ or $\textsc{Pull}(p)$ then
 8:                 $\textsc{Write}(\bot, \texttt{parent}, \textsc{null})$ and $\textsc{Prune}()$.
 9:             else if $M_i$ is $\textsc{Push}(p)$ then
10:                 $\textsc{Write}(\bot, \texttt{parent}, \textsc{null})$ and $\textsc{Write}(p, \texttt{parent}, \textsc{null})$.
11:                 $\textsc{Write}(\bot, \texttt{state}, \textsc{pruning})$ and $\textsc{Write}(p, \texttt{state}, \textsc{pruning})$.
12:             Execute $M_i$."
13: Construct $\alpha_{\textsc{EnergyDistribution}} : g_{\textsc{EnergyDistribution}} \to ops_{\textsc{EnergyDistribution}}$ as:
14:     Set $g_{\textsc{EnergyDistribution}} \leftarrow \bigvee_{g \in \mathcal{G}}(g)$, where $\mathcal{G} = \{$
            $g_{\textsc{GetPruned}}$       $= (A.\texttt{state} = \textsc{pruning})$,
            $g_{\textsc{AskGrowth}}$       $= (A.\texttt{state} = \textsc{active}) \wedge (A$ has an $\textsc{idle}$ neighbor or $\textsc{asking}$ child$)$,
            $g_{\textsc{GrowForest}}$      $= (A.\texttt{state} = \textsc{growing}) \vee$
15:                                         $\big((A.\texttt{state} = \textsc{source}) \wedge (A$ has an $\textsc{idle}$ neighbor or $\textsc{asking}$ child$)\big)$,
            $g_{\textsc{HarvestEnergy}}$ $= (A.\texttt{state} = \textsc{source}) \wedge (A.e_{bat} < \kappa)$,
            $g_{\textsc{ShareEnergy}}$     $= (A.\texttt{state} \notin \{\textsc{idle}, \textsc{pruning}\}) \wedge$
                                            $(A.e_{bat} \geq 1) \wedge (A$ has a child $B : B.e_{bat} < \kappa)\}$
16:     Set $ops_{\textsc{EnergyDistribution}} \leftarrow$ "Do:
17:         if $g_{\textsc{GetPruned}}$ then $\textsc{Prune}()$.                                   ▷ GetPruned
18:         if $g_{\textsc{AskGrowth}}$ then $\textsc{Write}(\bot, \texttt{state}, \textsc{asking})$.                  ▷ AskGrowth
19:         if $g_{\textsc{GrowForest}}$ then                                               ▷ GrowForest
20:             for each port $p$ for which $\textsc{Connected}(p) = \textsc{true}$ and $\textsc{Read}(p, \texttt{state}) = \textsc{idle}$ do
21:                 $\textsc{Write}(p, \texttt{parent}, p')$, where $p'$ is any port of the neighbor on port $p$ facing $A$.
22:                 $\textsc{Write}(p, \texttt{state}, \textsc{active})$.
23:             for each port $p \in \textsc{Children}() : (\textsc{Read}(p, \texttt{state}) = \textsc{asking})$ do
24:                 $\textsc{Write}(p, \texttt{state}, \textsc{growing})$.
25:             if $\textsc{Read}(\bot, \texttt{state}) = \textsc{growing}$ then $\textsc{Write}(\bot, \texttt{state}, \textsc{active})$.
26:         if $g_{\textsc{HarvestEnergy}}$ then $\textsc{Write}(\bot, e_{bat}, \textsc{Read}(\bot, e_{bat}) + 1)$.      ▷ HarvestEnergy
27:         if $g_{\textsc{ShareEnergy}}$ then                                              ▷ ShareEnergy
28:             Let port $p \in \textsc{Children}()$ be one for which $\textsc{Read}(p, e_{bat}) < \kappa$.
29:             $\textsc{Write}(\bot, e_{bat}, \textsc{Read}(\bot, e_{bat}) - 1)$.
30:             $\textsc{Write}(p, e_{bat}, \textsc{Read}(p, e_{bat}) + 1)$."
31: return $\mathcal{A}^\delta = \{[\alpha_i^\delta : g_i^\delta \to ops_i^\delta] : i \in \{1, \ldots, m\}\} \cup \{\alpha_{\textsc{EnergyDistribution}}\}$.

32: function $\textsc{Children}()$
33:     return {ports $p : \textsc{Connected}(p) \wedge (\textsc{Read}(p, \texttt{parent})$ points to $A)\}$.
34: function $\textsc{Prune}()$
35:     for each port $p \in \textsc{Children}()$ do
36:         $\textsc{Write}(p, \texttt{state}, \textsc{pruning})$.
37:         $\textsc{Write}(p, \texttt{parent}, \textsc{null})$.
38:     if $\textsc{Read}(\bot, \texttt{state}) \neq \textsc{source}$ then $\textsc{Write}(\bot, \texttt{state}, \textsc{idle})$.
```

"equivalent" behavior (defined formally in Section 3.2). At a high level, $\mathcal{A}^\delta$ works as follows. The amoebot system first self-organizes as a spanning forest $\mathcal{F}$ rooted at source amoebots with access to external energy sources. Energy is harvested by source amoebots and transferred from parents to children in $\mathcal{F}$ as there is need. Amoebots spend energy on enabled actions of algorithm $\mathcal{A}$ until they become deficient, when they will once again need to wait to recharge. This process repeats until termination, which must occur since $\mathcal{A}$ is energy-compatible.

Algorithm $\mathcal{A}^\delta$ comprises two types of actions. First, every action $\alpha_i \in \mathcal{A}$ is transformed into an energy-constrained version $\alpha_i^\delta \in \mathcal{A}^\delta$ (Algorithm 1, Lines 1–12). By including $A.e_{bat} \geq \delta(\alpha_i)$ in its guard g_i^δ and spending $\delta(\alpha_i)$ energy at the start of its operations ops_i^δ, the transformed action α_i^δ is only executed if there is sufficient energy to do so and any such execution spends the corresponding energy. The guard g_i^δ also ensures any amoebot executing an α_i^δ action and all of its neighbors are part of the forest structure $\mathcal{F}$.

Second, there is a singular $\alpha_{\text{EnergyDistribution}}$ action that defines how amoebots self-organize as a spanning forest and distribute energy throughout the system (Algorithm 1, Lines 13–30). Its operations are organized into five blocks – GetPruned, AskGrowth, GrowForest, HarvestEnergy, and ShareEnergy– each of which has a corresponding logical predicate in the set $\mathcal{G}$. These predicates appear in the guard $\bigvee_{g \in \mathcal{G}}(g)$, which ensures that $\alpha_{\text{EnergyDistribution}}$ is only enabled when its execution would progress towards distributing energy to deficient amoebots. The latter is critical for proving that $\mathcal{A}^\delta$ achieves energy distribution even under an unfair adversary, which we show in Section 3.2. The remainder of this section details the five blocks; their local variables are summarized in Table 1.

Forming and Maintaining a Spanning Forest. Recall from Section 2.2 that we consider amoebot systems that are initially connected and contain at least one source amoebot with access to an external energy source. The GetPruned, AskGrowth, and GrowForest blocks (Algorithm 1, Lines 17–25) continuously organize the amoebot system as a spanning forest $\mathcal{F}$ of trees rooted at the source amoebot(s). These trees act as an acyclic resource distribution network for energy transfers, which is important for avoiding non-termination under an unfair adversary.

The well-established *spanning forest primitive* [9] and the recent *feather tree formation* algorithm [25] are both guaranteed to organize an amoebot system as a spanning forest $\mathcal{F}$ under an unfair sequential adversary, assuming no parent–child relationship in $\mathcal{F}$ is ever disrupted after it is formed. However, many amoebot algorithms $\mathcal{A}$ – and by extension, the actions α_i^δ of algorithms $\mathcal{A}^\delta$ – cause amoebots to move, partitioning $\mathcal{F}$ into "unstable" trees whose connections to source amoebots have been disrupted and "stable" trees that remain rooted at sources. This necessitates a protocol for dynamically repairing $\mathcal{F}$ as amoebots move. To this end, the earlier Forest-Prune-Repair algorithm [11] was designed to "prune" unstable trees, allowing their amoebots to rejoin stable trees. Unfortunately, Forest-Prune-Repair requires fairness for termination, which we do not have here. In the following, we describe a new algorithm that dynamically maintains $\mathcal{F}$ under an unfair sequential adversary.

Each amoebot has a `state` variable that is initialized to SOURCE for source amoebots and IDLE for all others. Additionally, each amoebot has a `parent` pointer indicating the port incident to their parent in the forest $\mathcal{F}$; these pointers are initially set to NULL. A source amoebot adopts its IDLE neighbors into its tree by making them ACTIVE and setting their `parent` pointers to itself (GrowForest, Algorithm 1, Lines 19–22). ACTIVE amoebots, however, must ask the source amoebot at the root of their tree for permission before adopting their IDLE neighbors (AskGrowth, Algorithm 1, Line 18). Although indirect, this ensures that IDLE amoebots only join trees that are (or were recently) stable, stopping the unfair adversary from creating non-terminating executions (see Lemma 4). Specifically, an ACTIVE amoebot with an IDLE neighbor becomes ASKING. Any ACTIVE amoebot with an ASKING child also becomes ASKING, propagating this "asking signal" towards the tree's source amoebot. When the source amoebot receives this asking signal, it updates all its ASKING children to GROWING, granting them permission to grow the tree. A GROWING amoebot adopts its IDLE neighbors as ACTIVE children, updates its ASKING children to GROWING, and resets its `state` to ACTIVE. This process repeats until no IDLE amoebots remain.

If an amoebot's movement during an α_i^δ execution would disrupt $\mathcal{F}$, it initiates a pruning process to dissolve disrupted subtrees. Amoebots performing CONTRACT or PULL movements must prune immediately since their movement may disconnect them from their neighbors; PUSH movements instead make the two involved amoebots PRUNING, which will cause them to prune during their next action. When an amoebot prunes, it makes its children PRUNING and resets both its own and its children's `parent` pointers, severing them from their tree (Algorithm 1, Lines 8 and 35–37). If it is not a source, it also becomes IDLE (Algorithm 1, Line 38). The GETPRUNED block ensures that any PRUNING amoebot does the same, dissolving the unstable tree (Algorithm 1, Line 17). These newly IDLE amoebots are then collected into stable trees by the ASKGROWTH and GROWFOREST blocks as described above.

Sharing Energy. The HARVESTENERGY and SHAREENERGY blocks (Algorithm 1, Lines 26–30) define how source amoebots harvest energy from external energy sources and how all non-IDLE, non-PRUNING amoebots transfer energy to their neighbors, respectively. If its battery is not already full, a source amoebot harvests a unit of energy from its external energy source into its own battery. Any non-IDLE, non-PRUNING amoebot with at least one unit of energy to share and a child whose battery is not full will then transfer a unit of energy from its own battery to that of its child.

3.2 Analysis

In this section, we sketch the results building to the following theorem. Informally, it states that an energy-constrained algorithm $\mathcal{A}^\delta$ produced by the energy distribution framework (1) only yields system outcomes that could have been achieved by the original energy-agnostic algorithm $\mathcal{A}$, provided $\mathcal{A}$ is energy-compatible, and (2) incurs an $\mathcal{O}(n^2)$ runtime overhead.

▶ **Theorem 2.** *Consider any energy-compatible amoebot algorithm $\mathcal{A}$ and demand function $\delta : \mathcal{A} \to \{1, 2, \ldots, \kappa\}$, and let $\mathcal{A}^\delta$ be the algorithm produced from $\mathcal{A}$ and δ by the energy distribution framework (Algorithm 1). Let C_0 be any (legal) connected initial configuration for $\mathcal{A}$ and let C_0^δ be its extension for $\mathcal{A}^\delta$ that designates at least one source amoebot and adds the energy distribution variables with their initial values (Table 1) to all amoebots. Then for any configuration C^δ in which an unfair sequential execution of $\mathcal{A}^\delta$ starting in C_0^δ terminates, there exists an unfair sequential execution of $\mathcal{A}$ starting in C_0 that terminates in a configuration C that is identical to C^δ modulo the energy distribution variables. Moreover, if all unfair sequential executions of $\mathcal{A}$ on n amoebots terminate after at most $T_{\mathcal{A}}(n)$ action executions, then any unfair sequential execution of $\mathcal{A}^\delta$ on n amoebots terminates in $\mathcal{O}(n^2 T_{\mathcal{A}}(n))$ rounds.*

Due to space constraints, we highlight only the most important supporting results of this analysis. All omitted lemmas, invariants, and proofs can be found in the full version of this paper (see link in title page).

▶ **Lemma 3.** *Consider any sequential execution $\mathcal{S}^\delta$ of $\mathcal{A}^\delta$ starting in initial configuration C_0^δ and let $\mathcal{S}_\alpha^\delta$ denote its subsequence of α_i^δ action executions. Then the corresponding sequence $\mathcal{S}_\alpha$ of α_i executions is a valid sequential execution of $\mathcal{A}$ starting in initial configuration C_0.*

This lemma implies that any sequential execution $\mathcal{S}^\delta$ of $\mathcal{A}^\delta$ contains a finite number of α_i^δ executions, since the corresponding sequence of α_i executions forms a possible sequential execution of $\mathcal{A}$, which must terminate because $\mathcal{A}$ is energy-compatible. It remains to analyze the *energy runs* in $\mathcal{S}^\delta$, i.e., the maximal sequences of consecutive $\alpha_{\text{EnergyDistribution}}$ executions delineated by α_i^δ executions. Formally, an execution of $\alpha_{\text{EnergyDistribution}}$ by an amoebot A is *g-supported* if predicate $g \in \mathcal{G}$ is satisfied when A is activated and executes $\alpha_{\text{EnergyDistribution}}$. We argue that any predicate $g \in \mathcal{G}$ can support at most a finite number of executions per energy run, implying that all energy runs, and thus all sequential executions of $\mathcal{A}^\delta$, are finite:

▶ **Lemma 4.** *Any energy run of $\mathcal{S}^\delta$ contains at most a finite number of g-supported $\alpha_{\text{EnergyDistribution}}$ executions, for any $g \in \mathcal{G}$.*

Let C^δ be the terminating configuration of $\mathcal{S}^\delta$. We must show that there exists a sequential execution of $\mathcal{A}$ starting in C_0 that terminates in the configuration C obtained from C^δ by removing the energy distribution variables. An obvious candidate is the sequence $\mathcal{S}_\alpha$ of α_i executions corresponding to the α_i^δ executions in $\mathcal{S}^\delta$. Lemma 3 already implies that $\mathcal{S}_\alpha$ reaches C, and a careful argument involving the guard of $\alpha_{\text{EnergyDistribution}}$ shows that it must also terminate there. The remainder of the analysis characterizes the time required for an *uninterrupted* energy run – i.e., one that is not ended early by an α_i^δ execution, which only helps the overall progress argument – to collect all amoebots into stable trees rooted at source amoebots and, once this is achieved, to fully recharge all amoebots' batteries.

▶ **Lemma 5.** *After at most $\mathcal{O}(n^2)$ rounds of any uninterrupted energy run of $\mathcal{S}^\delta$, all n amoebots belong to stable trees.*

▶ **Lemma 6.** *After at most $\mathcal{O}(n)$ rounds of any uninterrupted, stabilized energy run of $\mathcal{S}^\delta$, all n amoebots have full batteries.*

These lemmas imply that every energy run terminates in at most $\mathcal{O}(n^2)$ rounds. The theorem supposes that any sequential execution of $\mathcal{A}$ terminates in $T_\mathcal{A}(n)$ action executions, so we know by Lemma 3 that any sequential execution of $\mathcal{A}^\delta$ contains at most $T_\mathcal{A}(n) + 1$ energy runs. Combining these facts yields the $\mathcal{O}(n^2 T_\mathcal{A}(n))$ runtime bound for $\mathcal{A}^\delta$.

4 Energy-Constrained Leader Election and Shape Formation

With the energy distribution framework defined and its properties analyzed, we now apply it to existing energy-agnostic algorithms for leader election and shape formation and show simulations of their energy-constrained counterparts. We first make a straightforward observation about *stationary* amoebot algorithms, i.e., those in which amoebots do not move. These include simple primitives like spanning forest formation [9] and binary counters [7, 33] as well as the majority of existing algorithms for leader election [3, 5, 8, 14, 15, 18, 19]. It is easily seen that an algorithm that never moves cannot disconnect an initially connected system, and its actions never involve a "move phase". Thus,

▶ **Observation 7.** *All stationary amoebot algorithms satisfy Convention 3, and those that do not use* Lock *or* Unlock *operations also satisfy Convention 2.*

Observation 7 immediately implies the following about stationary algorithms' compatibility with the energy distribution framework.

▶ **Corollary 8.** *Any stationary amoebot algorithm that terminates under every (unfair) sequential execution, comprises only valid actions (i.e., those whose executions always succeed in isolation), and does not use* Lock *or* Unlock *operations is energy-compatible.*

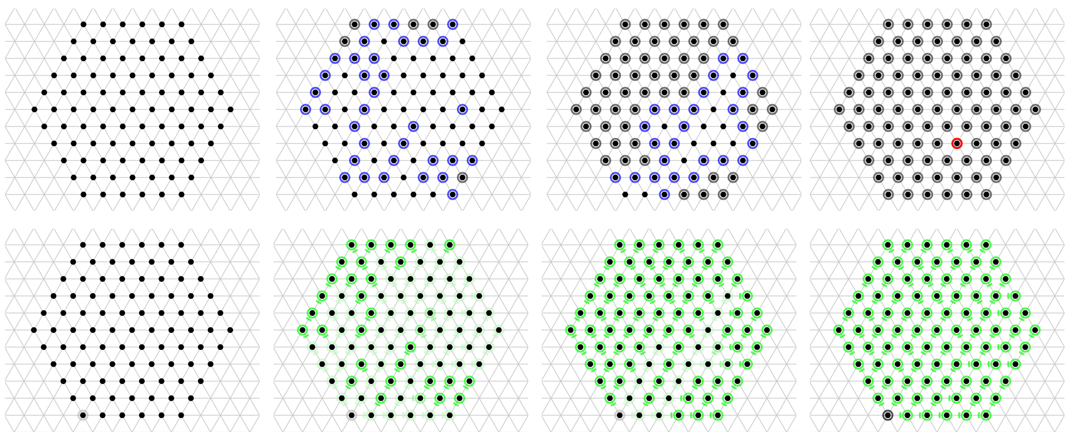

(a) $t = 0$ rounds. **(b)** $t = 100$. **(c)** $t = 250$. **(d)** $t = 350$.

Figure 2 *Simulating* Leader-Election-by-Erosion$^\delta$. A simulation of Leader-Election-by-Erosion$^\delta$ on $n = 91$ amoebots with one source amoebot, capacity $\kappa = 10$, and demand $\delta(\alpha) = 5$ for all actions α. Both rows show the same simulation. Top: For Leader-Election-by-Erosion, amoebots are initially "null candidates" (no color) and eventually declare candidacy (blue); candidates then either erode (dark gray) or become the unique leader (red). Bottom: For energy distribution, color opacity indicates energy levels. All amoebots are initially IDLE (no color) except the source (gray/black); amoebots eventually join the forest $\mathcal{F}$ (green) and distribute energy.

One such algorithm is Leader-Election-by-Erosion, a deterministic leader election algorithm for hole-free, connected amoebot systems introduced by Di Luna et al. [15] and extended to the canonical amoebot model and three-dimensional space by Briones et al. [5]. All amoebots first become leader candidates. When activated, a candidate uses certain rules regarding the number and relative positions of its neighbors to decide whether to "erode", revoking its candidacy without disconnecting or introducing a hole into the remaining set of candidates. The last remaining candidate is necessarily unique and thus declares itself the leader.

▶ **Lemma 9.** *Leader-Election-by-Erosion is energy-compatible.*

Proof. Leader-Election-by-Erosion is clearly stationary – no movement is involved in checking neighbors' positions or revoking candidacy – so it suffices to check the conditions of Corollary 8. Briones et al. [5] have already shown that any unfair sequential execution of this algorithm elects a leader – and thus terminates – in $\mathcal{O}(n)$ rounds. This correctness analysis also confirms that no actions of Leader-Election-by-Erosion are invalid; otherwise, some action executions would fail. Finally, it is easy to verify from the algorithm's pseudocode in [5] that LOCK and UNLOCK are not used, so we are done. ◀

Combining this lemma, the energy distribution framework's guarantees (Theorem 2), and Leader-Election-by-Erosion's correctness and runtime guarantees (Theorem 6.3 of [5]) immediately implies the following theorem.

▶ **Theorem 10.** *For any demand function δ : Leader-Election-by-Erosion $\to \{1, 2, \ldots, \kappa\}$, the algorithm Leader-Election-by-Erosion$^\delta$ produced by the energy distribution framework deterministically solves the leader election problem for hole-free, connected systems of n amoebots in $\mathcal{O}(n^3)$ rounds assuming geometric space, assorted orientations, constant-size memory, and an unfair sequential adversary.*

A simulation of Leader-Election-by-Erosion$^\delta$ successfully electing a unique leader under energy constraints is shown in Figure 2. As the proof of Lemma 9 shows, Corollary 8 sets a very low bar for proving stationary algorithms are energy-compatible. Almost all existing amoebot algorithms are designed to terminate after achieving a desired system behavior, and this property is typically proven as part of their correctness analyses. Invalid actions are avoided, as their executions would always fail.[3] Finally, no existing algorithms use the concurrency control operations Lock and Unlock directly; these are typically reserved for use by the "concurrency control framework" [10] discussed in the next section. The only remaining obstacle is that many existing stationary algorithms predate the canonical amoebot model and have not yet been reformulated in guarded action semantics or analyzed under an unfair adversary. Supposing this obstacle can be overcome without significantly affecting the algorithms' previously proven guarantees, the above discussion shows it is likely that most – if not all – existing stationary amoebot algorithms are energy-compatible.

What about non-stationary amoebot algorithms whose movements make satisfying the phase structure and connectivity conventions (Conventions 2 and 3) non-trivial? Here our example is the Hexagon-Formation algorithm for basic shape formation, originally introduced by Derakhshandeh et al. [13] and carefully reformulated and analyzed under the canonical amoebot model by Daymude et al. [10]. The basic idea of this algorithm is to form a hexagon – or as close to one as is possible with the number of amoebots in the system – by extending a spiral that begins at a (pre-defined or elected) seed amoebot. Thanks to the analysis in [10], it is easy to show Hexagon-Formation is compatible with the energy distribution framework.

▶ **Lemma 11.** *Hexagon-Formation is energy-compatible.*

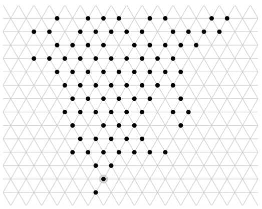
Proof. Every sequential execution of Hexagon-Formation must terminate since Lemma 7 of [10] guarantees that any execution of this algorithm – sequential or concurrent – terminates with the amoebot system forming a hexagon. Theorem 10 of [10] guarantees that Hexagon-Formation satisfies the validity and phase structure conventions (Conventions 1 and 2), as these were the two conventions borrowed directly from that paper's concurrency control framework. Finally, Hexagon-Formation is guaranteed to maintain the connectivity of an initially connected system configuration by Lemma 3 of [10], satisfying Convention 3. ◀

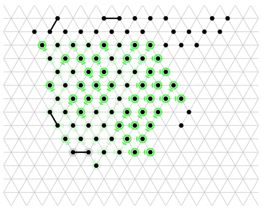
Combining this lemma, the energy distribution framework's guarantees (Theorem 2), Hexagon-Formation's correctness guarantees (Theorem 8 of [10]), and Hexagon-Formation's $\Theta(n^2)$ worst-case work bound [13], we have:

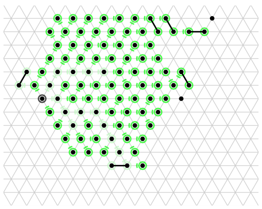
▶ **Theorem 12.** *For any demand function δ : Hexagon-Formation $\to \{1, 2, \ldots, \kappa\}$, the algorithm Hexagon-Formation$^\delta$ produced by the energy distribution framework deterministically solves the hexagon formation problem for connected systems of n amoebots in $\mathcal{O}(n^4)$ rounds assuming geometric space, assorted orientations, constant-size memory, and an unfair sequential adversary.*

Figure 3 depicts a simulation of Hexagon-Formation$^\delta$ forming a hexagon under energy constraints. We emphasize that Leader-Election-by-Erosion and Hexagon-Formation are not cherry-picked examples with particularly straightforward proofs of energy-compatibility. On the contrary, we expect that like our two examples, many algorithms already have the ingredients of energy-compatibility proven in their existing correctness analyses.

[3] The canonical amoebot model introduced error handling for amoebot algorithm design to deal with operation executions that fail due to concurrency (see Section 2.2 of [10]). Although error handling could be used to deal with failed executions of invalid actions, no existing amoebot algorithms have taken such a convoluted approach to designing functional algorithms.


(a) $t = 0$ rounds.


(b) $t = 400$.


(c) $t = 900$.

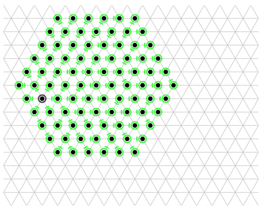
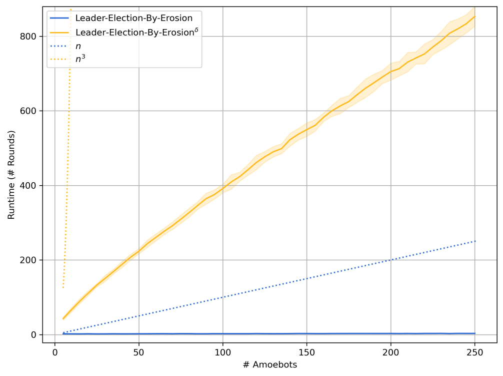
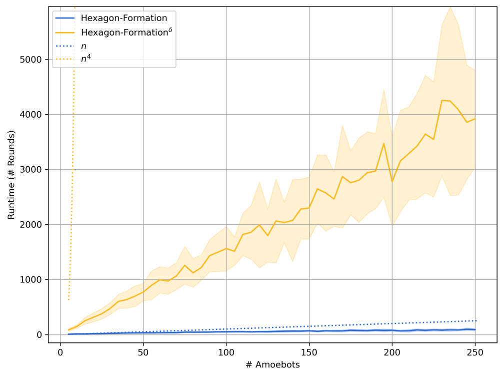

(d) $t = 1200$.

Figure 3 *Simulating Hexagon-Formation$^\delta$.* A simulation of Hexagon-Formation$^\delta$ on $n = 91$ amoebots with one source amoebot, capacity $\kappa = 10$, and demand $\delta(\alpha) = 5$ for all actions α. States from Hexagon-Formation are not visualized. For energy distribution, color opacity indicates energy levels. All amoebots are initially IDLE (no color) except the source (gray/black); amoebots eventually join the forest $\mathcal{F}$ (green) and distribute energy.



(a) Leader-Election-by-Erosion.



(b) Hexagon-Formation.

Figure 4 *Runtime Comparisons.* The energy-constrained (a) Leader-Election-by-Erosion$^\delta$ and (b) Hexagon-Formation$^\delta$ algorithms' runtimes (yellow) and their energy-agnostic counterparts (blue) in terms of sequential rounds. Each algorithm was simulated in 25 independent trials per system size $n \in \{5, 10, \ldots, 250\}$; average runtimes are shown as solid lines and one standard deviation is shown as an error tube. Relevant asymptotic runtime bounds are shown as dotted lines: the energy-agnostic algorithms both terminate in linear rounds (blue) and the energy-constrained algorithms' bounds are given by Theorems 10 and 12 (yellow).

We validate the runtime bounds for Leader-Election-by-Erosion$^\delta$ and Hexagon-Formation$^\delta$ given in Theorems 10 and 12, respectively, by simulating these algorithms and their energy-agnostic counterparts for a range of system sizes n. Figure 4 reports their empirical runtimes. Both energy-constrained algorithms well outperform their theoretical bounds, with Leader-Election-by-Erosion$^\delta$ achieving a near-linear runtime and Hexagon-Formation$^\delta$ remaining sub-quadratic. This suggests that our overhead bound can be optimized further or describes only some pessimistic worst-case scenarios.

5 Asynchronous Energy-Constrained Algorithms

Our energy distribution results thus far consider sequential concurrency, in which at most one amoebot can be active at a time (Section 2.1). This section details a useful extension of these results to *asynchronous concurrency*, in which arbitrary amoebots can be simultaneously active and their action executions can overlap arbitrarily in time.

There are many hazards of asynchrony that complicate amoebot algorithm design, with concurrent movements and memory updates potentially causing operations to fail or action executions to exhibit unintended behaviors. To reduce this complexity, one can use the *concurrency control framework* for amoebot algorithms that – analogous to our own energy distribution framework for energy-agnostic/constrained algorithms – transforms any algorithm $\mathcal{A}$ that terminates under every (unfair) sequential execution and satisfies certain conventions into an algorithm $\mathcal{A}'$ that achieves equivalent behavior under any asynchronous execution [10]. Formally, an amoebot algorithm $\mathcal{A}$ is *concurrency-compatible* if every (unfair) sequential execution of $\mathcal{A}$ terminates and it satisfies the validity, phase structure, and expansion-robustness conventions. The first two conventions are identical to Conventions 1 and 2 of the energy distribution framework. The third convention, *expansion-robustness*, requires actions to be resilient to concurrent expansions into their neighborhood.

We originally aimed to prove that the energy distribution framework preserves any input algorithm's concurrency-compatibility – i.e., if an algorithm $\mathcal{A}$ is concurrency-compatible, then so is $\mathcal{A}^\delta$ – and thus the two frameworks can be composed to obtain energy-constrained, asynchronous versions of all energy-compatible, concurrency-compatible algorithms. But as will become clearer after we formally define expansion-robustness (Definition 13), knowing that $\mathcal{A}$ is expansion-robust is seemingly insufficient for proving that $\mathcal{A}^\delta$ is also expansion-robust: the former only describes terminating configurations for $\mathcal{A}$ while the latter requires analyzing possible amoebot movements in all intermediate configurations reached by $\mathcal{A}^\delta$. Instead, we focus on a special case of expansion-robustness called *expansion-correspondence* (Definition 14) that we can prove is preserved by the energy distribution framework (Lemma 15). Although this restriction may appear limiting, the only algorithm known to be non-trivially expansion-robust (Hexagon-Formation of [10]) was proven to be expansion-robust via expansion-correspondence. Thus, until an algorithm is discovered to be expansion-robust but not expansion-corresponding, our present focus covers all known concurrency-compatible algorithms.

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 an *expand flag* $A.\texttt{flag}_p$ for each of its ports p that is initially FALSE, becomes TRUE whenever A expands to reveal a new port p, and is reset to FALSE whenever A or one of its neighbors executes a later action. These expand flags communicate when an amoebot has newly expanded into another amoebot's neighborhood. Each action $\alpha_i : g_i \to ops_i$ in $\mathcal{A}$ becomes an action $\alpha_i^E : g_i^E \to ops_i^E$ in $\mathcal{A}^E$ (see Algorithm 2 in Appendix A for details). The main difference is that while an amoebot A executes actions with respect to its full neighborhood $N(A)$ in $\mathcal{A}$, it does so only with respect to its *established neighborhood* $N^E(A) = \{B \in N(A) : \exists \text{ port } p \text{ of } B \text{ connected to } A \text{ s.t. } B.\texttt{flag}_p = \text{FALSE}\}$ in $\mathcal{A}^E$, effectively ignoring its newly expanded neighbors until its next action execution.

▶ **Definition 13.** *An amoebot algorithm $\mathcal{A}$ is expansion-robust if for any (legal) initial system configuration C_0 of $\mathcal{A}$, the following conditions hold:*

1. *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. *If a sequential execution of $\mathcal{A}^E$ starting in C_0^E terminates in a configuration C^E, some sequential execution of $\mathcal{A}$ starting in C_0 terminates in C (i.e., C^E without expand flags).*

As alluded to earlier, expansion-robustness only guarantees that sequential executions of $\mathcal{A}^E$ terminate and do so in a configuration that is reachable by a sequential execution of $\mathcal{A}$. This appears to be insufficient to prove $\mathcal{A}^\delta$ is expansion-robust. We instead focus on the following special case of expansion-robustness.

▶ **Definition 14.** *An amoebot algorithm $\mathcal{A}$ is* expansion-corresponding *if for any (legal) initial system configuration C_0 of $\mathcal{A}$, the following conditions hold:*

1. *If an action $\alpha^E_{i\neq 0} \in \mathcal{A}^E$ is enabled for some amoebot A w.r.t. $N^E(A)$, then action $\alpha_i \in \mathcal{A}$ is enabled for A w.r.t. $N(A)$.*
2. *The executions of $\alpha^E_{i\neq 0}$ w.r.t. $N^E(A)$ and α_i w.r.t. $N(A)$ by an amoebot A are identical, except the handling of expand flags.*

The main lemma of this section proves that the energy distribution framework preserves expansion-correspondence. Its proof and supporting results can be found in Appendix A.

▶ **Lemma 15.** *For any energy-compatible, expansion-corresponding algorithm $\mathcal{A}$ and demand function $\delta : \mathcal{A} \to \{1, 2, \dots, \kappa\}$, the algorithm $\mathcal{A}^\delta$ produced from $\mathcal{A}$ and δ by the energy distribution framework is concurrency-compatible.*

Lemma 15 shows that the energy distribution and concurrency control frameworks can be composed to obtain the benefits of both. Specifically, an amoebot algorithm designer should first design their algorithm without energy constraints and perform the usual safety and liveness analyses with respect to an unfair sequential adversary. If the algorithm always terminates, then they need only prove their algorithm satisfies the validity, phase structure, and connectivity conventions and argue that their algorithm is expansion-corresponding to automatically obtain an energy-constrained, asynchronous version of their algorithm with equivalent behavior, courtesy of the two frameworks. The following theorem states this result formally by combining the energy distribution framework's guarantees (Theorem 2), the concurrency control framework's guarantees (Theorem 11 of [10]), and Lemma 15. Note that because the runtime overhead of the concurrency control framework is not known, this theorem does not give any overhead bounds.

▶ **Theorem 16.** *Consider any energy-compatible, expansion-corresponding amoebot algorithm $\mathcal{A}$ and demand function $\delta : \mathcal{A} \to \{1, 2, \dots, \kappa\}$. Let $\mathcal{A}^\delta$ be the algorithm produced from $\mathcal{A}$ and δ by the energy distribution framework (Algorithm 1) and let $(\mathcal{A}^\delta)'$ be the algorithm produced from $\mathcal{A}^\delta$ by the concurrency control framework (Algorithm 4 of [10]). Let C_0 be any (legal) connected initial configuration for $\mathcal{A}$ and let $(C_0^\delta)'$ be its extension for $(\mathcal{A}^\delta)'$ that designates at least one source amoebot and adds the energy distribution and concurrency control variables with their initial values (Table 1 and* `act` *and* `awaken` *of [10]) to all amoebots. Then every asynchronous execution of $(\mathcal{A}^\delta)'$ starting in $(C_0^\delta)'$ terminates. Moreover, if $(C^\delta)'$ is the final configuration of some asynchronous execution of $(\mathcal{A}^\delta)'$ starting in $(C_0^\delta)'$, then there exists a sequential execution of $\mathcal{A}$ starting in C_0 that terminates in a configuration C that is identical to $(C^\delta)'$ modulo the energy distribution and concurrency control variables.*

We conclude this section by applying Theorem 16 to the Leader-Election-by-Erosion and Hexagon-Formation algorithms from Section 4. Those algorithms were shown to be energy-compatible in Lemmas 9 and 11 and expansion-corresponding in Lemma 7.1 of [5] and Theorem 10 of [10], respectively. Therefore,

▶ **Corollary 17.** *There exist energy-constrained amoebot algorithms that deterministically solve the leader election problem (for hole-free, connected systems) and the hexagon formation problem (for connected systems) assuming geometric space, assorted orientations, constant-size memory, and an unfair asynchronous adversary – the most general of all adversaries.*

6 Conclusion

In this work, we introduced the energy distribution framework for amoebot algorithms which transforms any energy-agnostic algorithm into an energy-constrained one with equivalent behavior, provided the original algorithm terminates under an unfair sequential adversary, maintains system connectivity, and follows some basic structural conventions (Theorem 2). We then proved that both the Leader-Election-by-Erosion and Hexagon-Formation algorithms are energy-compatible (Theorems 10 and 12). Perhaps surprisingly, these proofs were not difficult. The algorithms' existing correctness and runtime analyses under an unfair sequential adversary provided nearly all that was needed for energy-compatibility, and we expect this would be true for other algorithms as well. Finally, we proved that if an energy-compatible algorithm is also expansion-corresponding, then its energy-constrained counterpart produced by our framework can be extended to asynchronous concurrency using the concurrency control framework for amoebot algorithms (Theorem 16).

The energy-constrained algorithms produced by our framework have an $\mathcal{O}(n^2)$ round runtime overhead, though our simulations of Leader-Election-by-Erosion$^\delta$ and Hexagon-Formation$^\delta$ suggest that the overhead is much lower in practice. Comparing Lemmas 5 and 6 reveals the spanning forest maintenance algorithm as the performance bottleneck, which uses $\mathcal{O}(n^2)$ rounds in the worst case to prune and rebuild a forest of stable trees. In particular, amoebots getting permission from their (source) root before adopting children is critical for avoiding non-termination under an unfair adversary (Lemma 4), but requires a number of rounds that is linear in the depth of the tree. Improving this bound either requires a new approach to acyclic resource distribution or an optimization of stable tree membership detection. A shortest-path tree – i.e., one that maintains equality between the in-tree and in-system distances from any amoebot to its root – would bound the depth of any tree by the diameter D of the system. This would reduce the overall overhead to $\mathcal{O}(nD)$ rounds, which is still $\mathcal{O}(n^2)$ in the worst case (e.g., a line) but could achieve up to $\mathcal{O}(n^{3/2})$ in the best case (e.g., a regular hexagon). However, the recent feather tree algorithm [25] for forming shortest-path forests in amoebot systems only works in stationary systems. Achieving an algorithm for shortest-path forest maintenance – not just formation – would both improve our present overhead bound and be an interesting contribution in its own right.

References

1 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. doi:10.1007/S00446-005-0138-3.

2 Palina Bartashevich, Doreen Koerte, and Sanaz Mostaghim. Energy-Saving Decision Making for Aerial Swarms: PSO-Based Navigation in Vector Fields. In *2017 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 1–8, 2017. doi:10.1109/SSCI.2017.8285178.

3 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*, volume 11914 of *Lecture Notes in Computer Science*, pages 22–37, 2019. doi:10.1007/978-3-030-34992-9_3.

4 Douglas Blackiston, Emma Lederer, Sam Kriegman, Simon Garnier, Joshua Bongard, and Michael Levin. A Cellular Platform for the Development of Synthetic Living Machines. *Science Robotics*, 6(52):eabf1571, 2021. doi:10.1126/scirobotics.abf1571.

5 Joseph L. Briones, Tishya Chhabra, Joshua J. Daymude, and Andréa W. Richa. Invited Paper: Asynchronous Deterministic Leader Election in Three-Dimensional Programmable Matter. In *Proceedings of the 24th International Conference on Distributed Computing and Networking*, pages 38–47, 2023. doi:10.1145/3571306.3571389.

6 Jason D. Campbell, Padmanabhan Pillai, and Seth Copen Goldstein. The Robot Is the Tether: Active, Adaptive Power Routing for Modular Robots with Unary Inter-Robot Connectors. In *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 4108–4115, 2005. doi:10.1109/IROS.2005.1545426.

7 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*, pages 2:1–2:10, 2020. doi:10.1145/3369740.3372916.

8 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*, volume 10718 of *Lecture Notes in Computer Science*, pages 127–140, 2017. doi:10.1007/978-3-319-72751-6_10.

9 Joshua J. Daymude, Kristian Hinnenthal, Andréa W. Richa, and Christian Scheideler. Computing by Programmable Particles. In Paola Flocchini, Giuseppe Prencipe, and Nicola Santoro, editors, *Distributed Computing by Mobile Entities*, volume 11340 of *Lecture Notes in Computer Science*, pages 615–681. Springer, Cham, 2019. doi:10.1007/978-3-030-11072-7_22.

10 Joshua J. Daymude, Andréa W. Richa, and Christian Scheideler. The Canonical Amoebot Model: Algorithms and Concurrency Control. *Distributed Computing*, 2023. doi:10.1007/s00446-023-00443-3.

11 Joshua J. Daymude, Andréa W. Richa, and Jamison W. Weber. Bio-Inspired Energy Distribution for Programmable Matter. In *International Conference on Distributed Computing and Networking 2021*, pages 86–95, 2021. doi:10.1145/3427796.3427835.

12 Zahra Derakhshandeh, Shlomi Dolev, Robert Gmyr, Andréa W. Richa, Christian Scheideler, and Thim Strothmann. Amoebot – A New Model for Programmable Matter. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 220–222, 2014. doi:10.1145/2612669.2612712.

13 Zahra Derakhshandeh, Robert Gmyr, Andréa W. Richa, Christian Scheideler, and Thim Strothmann. An Algorithmic Framework for Shape Formation Problems in Self-Organizing Particle Systems. In *Proceedings of the Second Annual International Conference on Nanoscale Computing and Communication*, pages 21:1–21:2, 2015. doi:10.1145/2800795.2800829.

14 Zahra Derakhshandeh, Robert Gmyr, Thim Strothmann, Rida Bazzi, Andréa W. Richa, and Christian Scheideler. Leader Election and Shape Formation with Self-Organizing Programmable Matter. In Andrew Phillips and Peng Yin, editors, *DNA Computing and Molecular Programming*, volume 9211 of *Lecture Notes in Computer Science*, pages 117–132, 2015. doi:10.1007/978-3-319-21999-8_8.

15 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. doi:10.1007/S00446-019-00350-6.

16 Shlomi Dolev, Sergey Frenkel, Michael Rosenblit, Ram Prasadh Narayanan, and K. Muni Venkateswarlu. In-Vivo Energy Harvesting Nano Robots. In *2016 IEEE International Conference on the Science of Electrical Engineering (ICSEE)*, pages 1–5, 2016. doi:10.1109/ICSEE.2016.7806107.

17 Paola Flocchini, Giuseppe Prencipe, and Nicola Santoro, editors. *Distributed Computing by Mobile Entities: Current Research in Moving and Computing*, volume 11340 of *Lecture Notes in Computer Science*. Springer, Cham, 2019. doi:10.1007/978-3-030-11072-7.

18 Nicolas Gastineau, Wahabou Abdou, Nader Mbarek, and Olivier Togni. Distributed Leader Election and Computation of Local Identifiers for Programmable Matter. In Seth Gilbert, Danny Hughes, and Bhaskar Krishnamachari, editors, *Algorithms for Sensor Systems*, volume 11410 of *Lecture Notes in Computer Science*, pages 159–179, 2019. doi:10.1007/978-3-030-14094-6_11.

19 Nicolas Gastineau, Wahabou Abdou, Nader Mbarek, and Olivier Togni. Leader Election and Local Identifiers for Three-dimensional Programmable Matter. *Concurrency and Computation: Practice and Experience*, 34(7):e6067, 2022. doi:10.1002/CPE.6067.

20 Kyle Gilpin, Ara Knaian, and Daniela Rus. Robot Pebbles: One Centimeter Modules for Programmable Matter through Self-Disassembly. In *2010 IEEE International Conference on Robotics and Automation*, pages 2485–2492, 2010. doi:10.1109/ROBOT.2010.5509817.

21 Robert Gmyr, Kristian Hinnenthal, Irina Kostitsyna, Fabian Kuhn, Dorian Rudolph, Christian Scheideler, and Thim Strothmann. Forming Tile Shapes with Simple Robots. *Natural Computing*, 19(2):375–390, 2020. doi:10.1007/S11047-019-09774-2.

22 Seth Copen Goldstein, Jason D. Campbell, and Todd C. Mowry. Programmable Matter. *Computer*, 38(6):99–101, 2005. doi:10.1109/MC.2005.198.

23 Seth Copen Goldstein, Todd C. Mowry, Jason D. Campbell, Michael P. Ashley-Rollman, Michael De Rosa, Stanislav Funiak, James F. Hoburg, Mustafa E. Karagozler, Brian Kirby, Peter Lee, Padmanabhan Pillai, J. Robert Reid, Daniel D. Stancil, and Michael P. Weller. Beyond Audio and Video: Using Claytronics to Enable Pario. *AI Magazine*, 30(2):29–45, 2009. doi:10.1609/AIMAG.V30I2.2241.

24 Serge Kernbach, editor. *Handbook of Collective Robotics: Fundamentals and Challenges*. Jenny Stanford Publishing, New York, NY, USA, 2013. doi:10.1201/b14908.

25 Irina Kostitsyna, Tom Peters, and Bettina Speckmann. Brief Announcement: An Effective Geometric Communication Structure for Programmable Matter. In *36th International Symposium on Distributed Computing (DISC 2022)*, volume 246 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 47:1–47:3, 2022. doi:10.4230/LIPICS.DISC.2022.47.

26 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. doi:10.1073/PNAS.1910837117.

27 Bruce J. MacLennan. The Morphogenetic Path to Programmable Matter. *Proceedings of the IEEE*, 103(7):1226–1232, 2015. doi:10.1109/JPROC.2015.2425394.

28 Othon Michail, George Skretas, and Paul G. Spirakis. On the Transformation Capability of Feasible Mechanisms for Programmable Matter. *Journal of Computer and System Sciences*, 102:18–39, 2019. doi:10.1016/J.JCSS.2018.12.001.

29 Sanaz Mostaghim, Christoph Steup, and Fabian Witt. Energy Aware Particle Swarm Optimization as Search Mechanism for Aerial Micro-Robots. In *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 1–7, 2016. doi:10.1109/SSCI.2016.7850263.

30 Nils Napp, Samuel Burden, and Eric Klavins. Setpoint Regulation for Stochastically Interacting Robots. *Autonomous Robots*, 30(1):57–71, 2011. doi:10.1007/S10514-010-9203-2.

31 Daniel Pickem, Paul Glotfelter, Li Wang, Mark Mote, Aaron Ames, Eric Feron, and Magnus Egerstedt. The Robotarium: A Remotely Accessible Swarm Robotics Research Testbed. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1699–1706, 2017. doi:10.1109/ICRA.2017.7989200.

32 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. doi:10.1007/S10514-018-9710-0.

33 Alexandra Porter and Andréa W. Richa. Collaborative Computation in Self-Organizing Particle Systems. In *Unconventional Computation and Natural Computation*, volume 10867 of *Lecture Notes in Computer Science*, pages 188–203, 2018. doi:10.1007/978-3-319-92435-9_14.

34 Tommaso Toffoli and Norman Margolus. Programmable Matter: Concepts and Realization. *Physica D: Nonlinear Phenomena*, 47(1-2):263–272, 1991. doi:10.1016/0167-2789(91)90296-L.

35 Hongxing Wei, Bin Wang, Yi Wang, Zili Shao, and Keith C.C. Chan. Staying-Alive Path Planning with Energy Optimization for Mobile Robots. *Expert Systems with Applications*, 39(3):3559–3571, 2012. doi:10.1016/J.ESWA.2011.09.046.

36 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*, pages 353–354, 2013. doi:10.1145/2422436.2422476.

A Omitted Analysis of Concurrency-Compatibility

This appendix contains the technical material omitted from Section 5 due to space constraints.

Algorithm 2 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 $A : A.\texttt{flag}_p = \text{TRUE}) \to \text{WRITE}(\bot, \texttt{flag}_p, \text{FALSE})$.
 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 $\text{WRITE}(\bot, \texttt{flag}_p, \text{FALSE})$.    ▷ Reset own expand flags.
 6:         for each unique neighbor $B \in \text{CONNECTED}()$ do
 7:             for each port $p$ of $B$ do $\text{WRITE}(B, \texttt{flag}_p, \text{FALSE})$.    ▷ Reset neighbors' expand flags.
 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 $\text{WRITE}(\bot, \texttt{flag}_p, \text{TRUE})$.
11:             for each new port $p$ of $B$ not connected to $A$ do $\text{WRITE}(B, \texttt{flag}_p, \text{TRUE})$.
12:         else if an EXPAND operation was successfully executed then
13:             for each new port $p$ of $A$ do $\text{WRITE}(\bot, \texttt{flag}_p, \text{TRUE})$.
14:         else if an EXPAND operation failed in its execution then undo $ops_i$."
15: return $\mathcal{A}^E = \{[\alpha_i^E : g_i^E \to ops_i^E] : i \in \{0, \ldots, m\}\}$.
```

▶ **Lemma 18.** *If amoebot algorithm $\mathcal{A}$ is expansion-corresponding, it is also expansion-robust.*

Proof. To prove termination, suppose to the contrary that all sequential executions of $\mathcal{A}$ starting in C_0 terminate, but there exists some infinite sequential execution $\mathcal{S}^E$ of $\mathcal{A}^E$ starting in C_0^E. Algorithm $\mathcal{A}$ is expansion-corresponding, so there is a sequential execution $\mathcal{S}$ that is identical to $\mathcal{S}^E$, modulo executions of α_0^E. Execution $\mathcal{S}$ terminates by supposition, so $\mathcal{S}^E$ must contain an infinite number of α_0^E executions after its final $\alpha_{i\neq 0}^E$ execution. But α_0^E executions only reset expand flags, and there are only a finite number of amoebots and a constant number of expand flags per amoebot to reset, a contradiction.

Correctness follows from the same observation. Only $\alpha_{i\neq 0}^E$ executions move amoebots and modify variables of $\mathcal{A}$. Since every sequential execution $\mathcal{S}^E$ of $\mathcal{A}^E$ starting in C_0^E represents an identical sequential execution $\mathcal{S}$ of $\mathcal{A}$ starting in C_0 (after removing the α_0^E executions), and since $\mathcal{S}^E$ terminates whenever $\mathcal{S}$ terminates by the above argument, we conclude that they must terminate in configurations that are identical, modulo expand flags. ◀

Before proving that the energy distribution framework preserves expansion-correspondence, we need one helper lemma characterizing established neighbors in $\mathcal{A}^\delta$.

▶ **Lemma 19.** *During an execution of $(\mathcal{A}^\delta)^E$, if an amoebot A has a neighbor $B \in N(A)$ that is* IDLE, PRUNING, *or a child of A, then $B \in N^E(A)$.*

Proof. Any neighbor $B \in N(A) \setminus N^E(A)$ expanded into $N(A)$ during an EXPAND operation by B, a PUSH operation by B, or a PULL operation by some other amoebot pulling B. Any movement in $(\mathcal{A}^\delta)^E$ occurs in an $(\alpha_i^\delta)^E$ execution, whose guard requires that both the executing amoebot and all its established neighbors are not IDLE or PRUNING. Thus, regardless of whether B is initiating the movement (an EXPAND or PUSH) or is participating in it (a PULL), B cannot be IDLE or PRUNING when it enters $N(A)$. Any subsequent action execution that could make B IDLE or PRUNING must also reset its expand flags (Algorithm 2, Line 7). So there are never IDLE or PRUNING neighbors in $N(A) \setminus N^E(A)$.

Next consider any child B of A. Amoebot B became a child of A when A adopted it during a $g_{\text{GrowForest}}$-supported execution of $\alpha^E_{\text{EnergyDistribution}}$. During this execution, A reset all expand flags of B (Algorithm 2, Line 7). As long as B is a child of A, its expand flags facing A remain reset. Thus, $B \in N^E(A)$. ◀

We can now prove the main lemma of this section.

▶ **Lemma 20.** *For any energy-compatible, expansion-corresponding algorithm $\mathcal{A}$ and demand function $\delta : \mathcal{A} \to \{1, 2, \ldots, \kappa\}$, the algorithm $\mathcal{A}^\delta$ produced from $\mathcal{A}$ and δ by the energy distribution framework is concurrency-compatible.*

Proof. By Theorem 2, we know that every sequential execution of $\mathcal{A}^\delta$ terminates. It remains to show that $\mathcal{A}^\delta$ satisfies the validity, phase structure, and expansion-robustness conventions.

By supposition, every action $\alpha_i \in \mathcal{A}$ in the original algorithm is valid, i.e., its execution is successful whenever it is enabled and all other amoebots are inactive. Since the guard g_i of α_i is a necessary condition for the energy-constrained version α_i^δ to be enabled, we know this validity carries over to the compute and movement phases of α_i. The only new operations added by the energy distribution framework in the α_i^δ and $\alpha_{\text{EnergyDistribution}}$ actions are CONNECTED operations (which never fail) and READ and WRITE operations involving existing neighbors. All of these must succeed, so every action of $\mathcal{A}^\delta$ is valid.

It is easy to see that $\mathcal{A}^\delta$ satisfies the phase structure convention. Its only movements are in the α_i^δ actions, each of which has at most one movement operation that it executes last. Moreover, the energy distribution framework does not add any LOCK or UNLOCK operations.

It remains to show $\mathcal{A}^\delta$ is expansion-robust, and by Lemma 18, it suffices to show $\mathcal{A}^\delta$ is expansion-corresponding. We first show that if some action of $(\mathcal{A}^\delta)^E$ is enabled for an amoebot A w.r.t. $N^E(A)$, then the corresponding action of $\mathcal{A}^\delta$ is enabled for A w.r.t. $N(A)$. We may safely consider only the guard conditions that depend on an amoebot's neighborhood; all others evaluate identically regardless of neighborhood.

- If $(\alpha_i^\delta)^E$ is enabled for an amoebot A, then A must satisfy g_i^E – i.e., A satisfies the guard g_i of $\alpha_i \in \mathcal{A}$ w.r.t. $N^E(A)$ – and neither A nor its established neighbors can be IDLE or PRUNING. Algorithm $\mathcal{A}$ is expansion-corresponding by supposition, so this implies that A must satisfy g_i w.r.t. $N(A)$ as well. Moreover, Lemma 19 ensures that if there are no IDLE or PRUNING neighbors in $N^E(A)$, there are none in $N(A)$ either.
- Suppose $\alpha^E_{\text{EnergyDistribution}}$ is enabled for an amoebot A because A has an IDLE neighbor or an ASKING child $B \in N^E(A)$, a condition in both $g_{\text{AskGrowth}}$ and $g_{\text{GrowForest}}$. We know $N^E(A) \subseteq N(A)$, so $\alpha_{\text{EnergyDistribution}}$ must be enabled for A w.r.t. $N(A)$ as well.
- Suppose $\alpha^E_{\text{EnergyDistribution}}$ is enabled for an amoebot A because A has a child $B \in N^E(A)$ whose battery is not full, a condition in $g_{\text{ShareEnergy}}$. By the same argument as above, we have $N^E(A) \subseteq N(A)$, so $\alpha_{\text{EnergyDistribution}}$ must be enabled for A w.r.t. $N(A)$ as well.

Finally, we show that the executions of any action of $(\mathcal{A}^\delta)^E$ w.r.t. $N^E(A)$ and the corresponding action of $\mathcal{A}^\delta$ w.r.t. $N(A)$ by the same amoebot A are identical. We may safely focus only on the parts of action executions that depend on or interact with an amoebot's neighbors; all others execute identically regardless of neighborhood.

- If A executes an $(\alpha_i^\delta)^E$ action, it emulates the operations of $\alpha_i \in \mathcal{A}$ w.r.t. $N^E(A)$. But algorithm $\mathcal{A}$ is expansion-corresponding by supposition, which immediately implies that an execution of α_i w.r.t. $N(A)$ is identical.
- If A executes an $(\alpha_i^\delta)^E$ action or the GetPruned block of $\alpha^E_{\text{EnergyDistribution}}$, it may update its children's `state` and `parent` variables during Prune( ). By Lemma 19, any child of A in $N(A)$ is also in $N^E(A)$, so the same children are pruned.
- If A executes the GrowForest block of $\alpha^E_{\text{EnergyDistribution}}$, it adopts all its idle neighbors as an active children. Any idle neighbor $B \in N^E(A)$ that A adopts must also be adopted when A executes $\alpha_{\text{EnergyDistribution}}$ since $N^E(A) \subseteq N(A)$. But if there are no idle neighbors in $N^E(A)$ for A to adopt, there cannot be any in $N(A)$ either by Lemma 19. Thus, either the same idle neighbors or no neighbors are adopted.
- If A executes the GrowForest block of $\alpha^E_{\text{EnergyDistribution}}$, it updates any asking children to growing. By Lemma 19, any child of A in $N(A)$ is also in $N^E(A)$, so the same children are updated in $\alpha_{\text{EnergyDistribution}}$.
- If A executes the ShareEnergy block of $\alpha^E_{\text{EnergyDistribution}}$, it transfers an energy unit to one of its children $B \in N^E(A)$ whose battery is not full. We know $N^E(A) \subseteq N(A)$, so B is also a possible recipient of this energy in $\alpha_{\text{EnergyDistribution}}$. ◀