The Canonical Amoebot Model: Algorithms and Concurrency Control

Joshua J. Daymude and **Andréa W. Richa** (Arizona State University)

Christian Scheideler (Universität Paderborn)

DISC 2021, Freiburg, Germany (Hybrid Event) — October 5–7, 2021

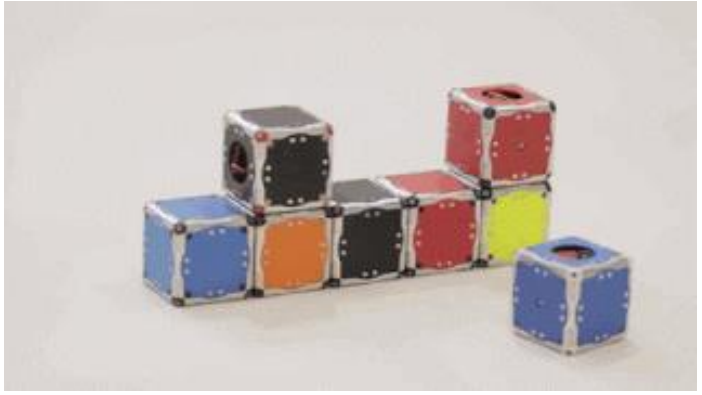
Programmable Matter

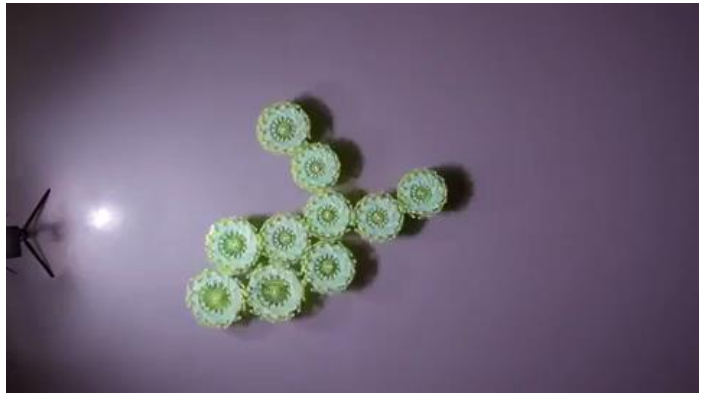
Programmable matter is a substance that can change its physical properties autonomously based on user input or environmental stimuli.

"Catoms"
PB 2018

"Kilobots"
RCN 2014

"M-Blocks"
RGR 2013



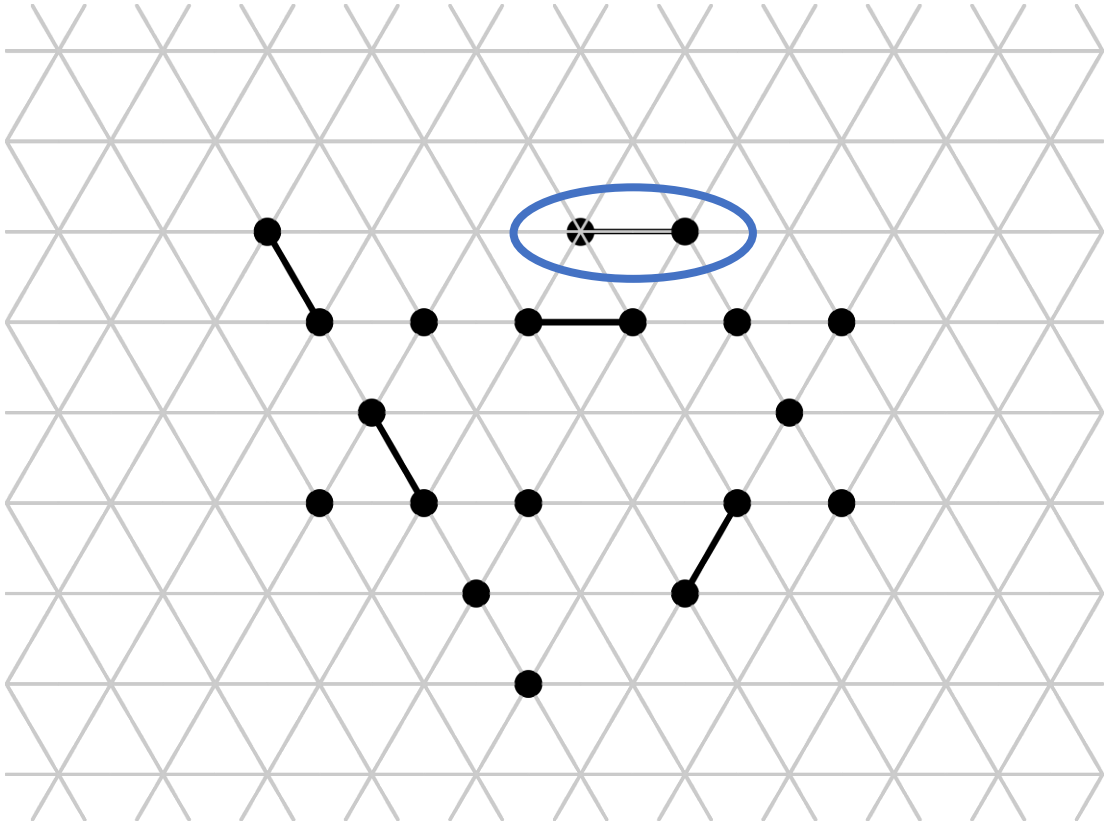


"Particle Robots"
LBBCRHRL 2019

The Geometric Amoebot Model (Before This Paper)

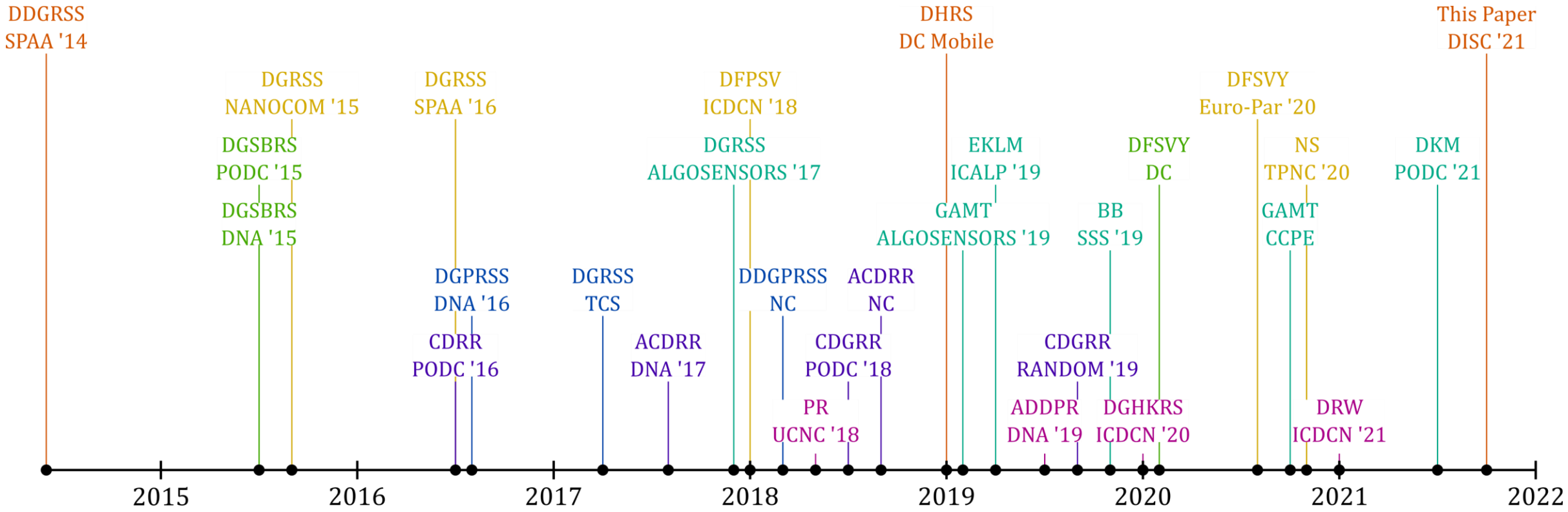
The (geometric) **amoebot model** is an abstraction of programmable matter.

- Space: triangular lattice G_Δ.
- Amoebots can be contracted (one node) or expanded (two adjacent nodes).
- Amoebots are anonymous, have only constant-size memories, communicate with immediate neighbors, and have no global compass.
- Self-actuated movements via expansions, contractions, and handovers.
- Sequential, weakly fair adversary: one amoebot acts per time, every amoebot acts infinitely often.



A Short History of the Amoebot Model

~30 papers: Models, shape formation, leader election, object coating, Markov chains, and more!



Related Models

The amoebot model considers active entities in space, unlike non-spatial, passive models like population protocols, network constructors, and tile-based self-assembly.

The most closely related model is **autonomous mobile robots** (see [FPS 2019] for a recent overview), specifically those using discrete, graph-based models of space.

Similarities

- Anonymous individuals.
- Active movements w/o global orientation.
- Limited compute/sensing capabilities.
- Persistent memory ($\mathcal{F}$-state). [BDS 2008]
- Limited communication (luminous robots). [DFPSY 2012, DFPSY 2016, DFCPSV 2017]

Differences

- Amoebots respect physical exclusion.
- Mobile robots use look-compute-move cycles that are difficult to reconcile with traditional message-passing.
- Mobile robots have a hierarchy of adversarial schedulers; amoebots usually assume sequential activations.

Overview of Results

1) The Canonical Amoebot Model

- Provides a standardized hierarchy of assumption variants to better facilitate comparisons.
- Models communication via message passing to address concurrency using standard adversarial activation models.

2) Sufficient Conditions for Concurrent Correctness

- Identifies sufficient conditions for amoebot algorithm correctness under any adversary.
- Shows that an algorithm for hexagon formation exists that satisfies these conditions.

3) A General Framework for Concurrency Control

- Converts algorithms that terminate under a sequential adversary and satisfy certain conventions into algorithms that exhibit equivalent behavior under an asynchronous adversary.

The Canonical Amoebot Model

The standardized hierarchy of assumption variants:

	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.

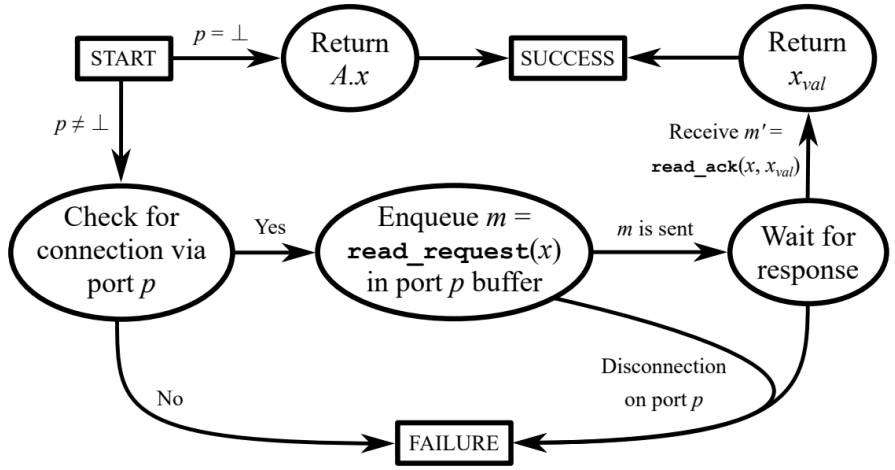
* have been considered in previous work; † are the focus of algorithmic results in this work.

The Canonical Amoebot Model

In the canonical amoebot model, amoebot functionality is partitioned into:

- A higher-level **application layer** where algorithms are defined in terms of operations.
- A lower-level **system layer** that executes an amoebot's operations via message passing.

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 $c_p = \text{FALSE}$ otherwise
READ(p, x)	The value of x in the public memory of this amoebot if $p = \perp$ or of the neighbor incident to port p otherwise
WRITE(p, x, x_{val})	Confirmation that the value of x was updated to x_{val} in the public memory of this amoebot if $p = \perp$ or of the neighbor incident to port p otherwise
CONTRACT(v)	Confirmation of the contraction out of node $v \in \{\text{HEAD}, \text{TAIL}\}$
EXPAND(p)	Confirmation of the expansion into the node incident to port p
PULL(p)	Confirmation of the pull handover with the neighbor incident to port p
PUSH(p)	Confirmation of the push handover with the neighbor incident to port p
LOCK()	Local identifiers of this amoebot and the neighbors that were locked
UNLOCK($\mathcal{L}$)	Confirmation that the amoebots of $\mathcal{L}$ were unlocked



(a) READ(p, x)

The Canonical Amoebot Model

Algorithms in the canonical amoebot model are specified in terms of **actions**:

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

- $label$ specifies the action's name.
- $guard$ is a Boolean predicate determining whether this action is currently **enabled**.
- $operations$ specifies the computation and sequence of operations to perform if enacted.

Example from Hexagon-Formation:

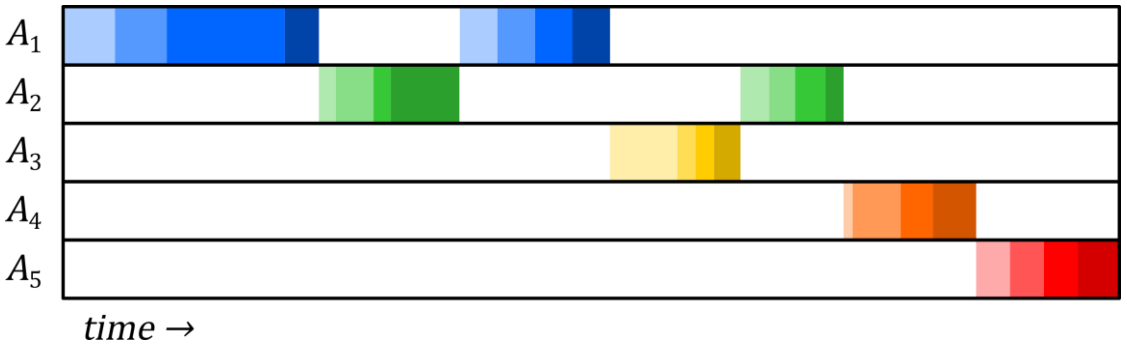
$\alpha_2 : (A.\text{state} = \text{IDLE}) \wedge (\exists B \in N(A) : B.\text{state} \in \{\text{FOLLOWER}, \text{ROOT}\}) \rightarrow$

Find a port p for which $\text{CONNECTED}(p) = \text{TRUE}$ and $\text{READ}(p, \text{state}) \in \{\text{FOLLOWER}, \text{ROOT}\}$.
$\text{WRITE}(\perp, \text{parent}, p)$.
$\text{WRITE}(\perp, \text{state}, \text{FOLLOWER})$.

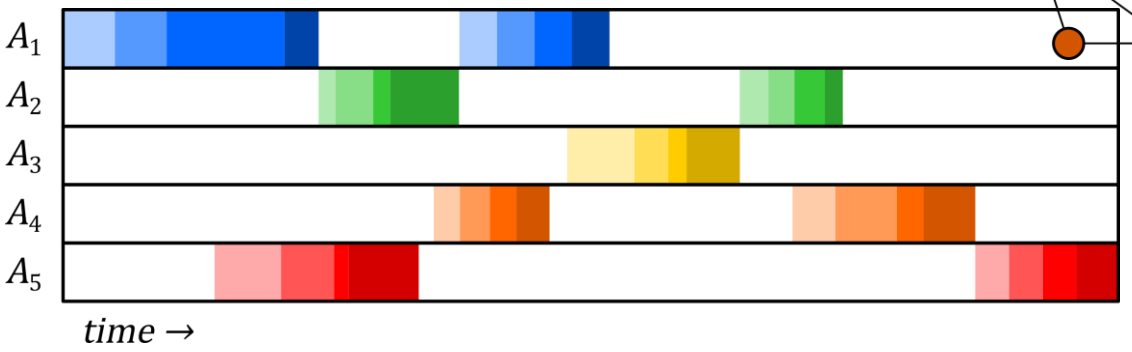
The Canonical Amoebot Model

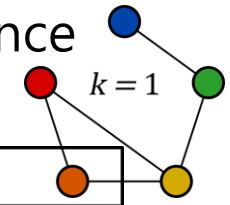
We use an **adversary** to model timing and progress. Four levels of concurrency:

1) Sequential. At most one amoebot can be active at a time.

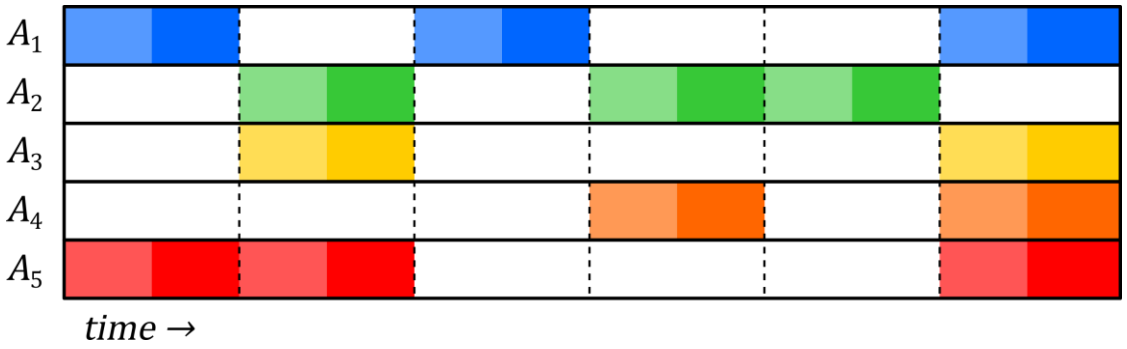


2) k-isolated. No amoebots within distance k can be simultaneously active.

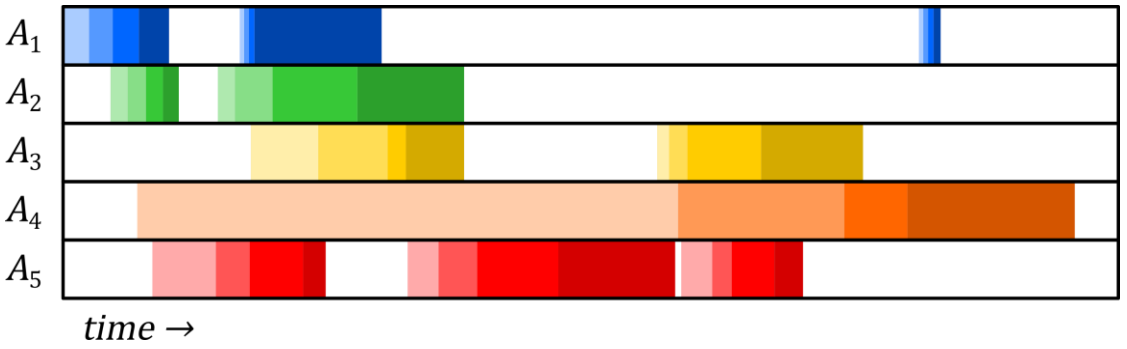




3) Synchronous. Arbitrary sets of amoebots can be active in each discrete round.



4) Asynchronous. Arbitrary sets of amoebots can be simultaneously active.



The Canonical Amoebot Model

The adversary can only activate amoebots with enabled actions. Three levels of fairness:

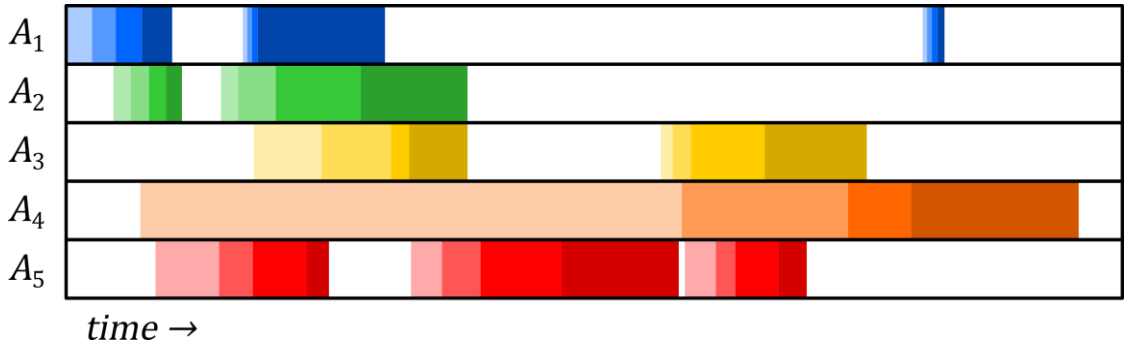
1) **Strongly Fair.** Every amoebot that is enabled infinitely often is activated infinitely often.

2) **Weakly Fair.** Every continuously enabled amoebot is eventually activated.

3) **Unfair.** Some enabled amoebot is eventually activated.

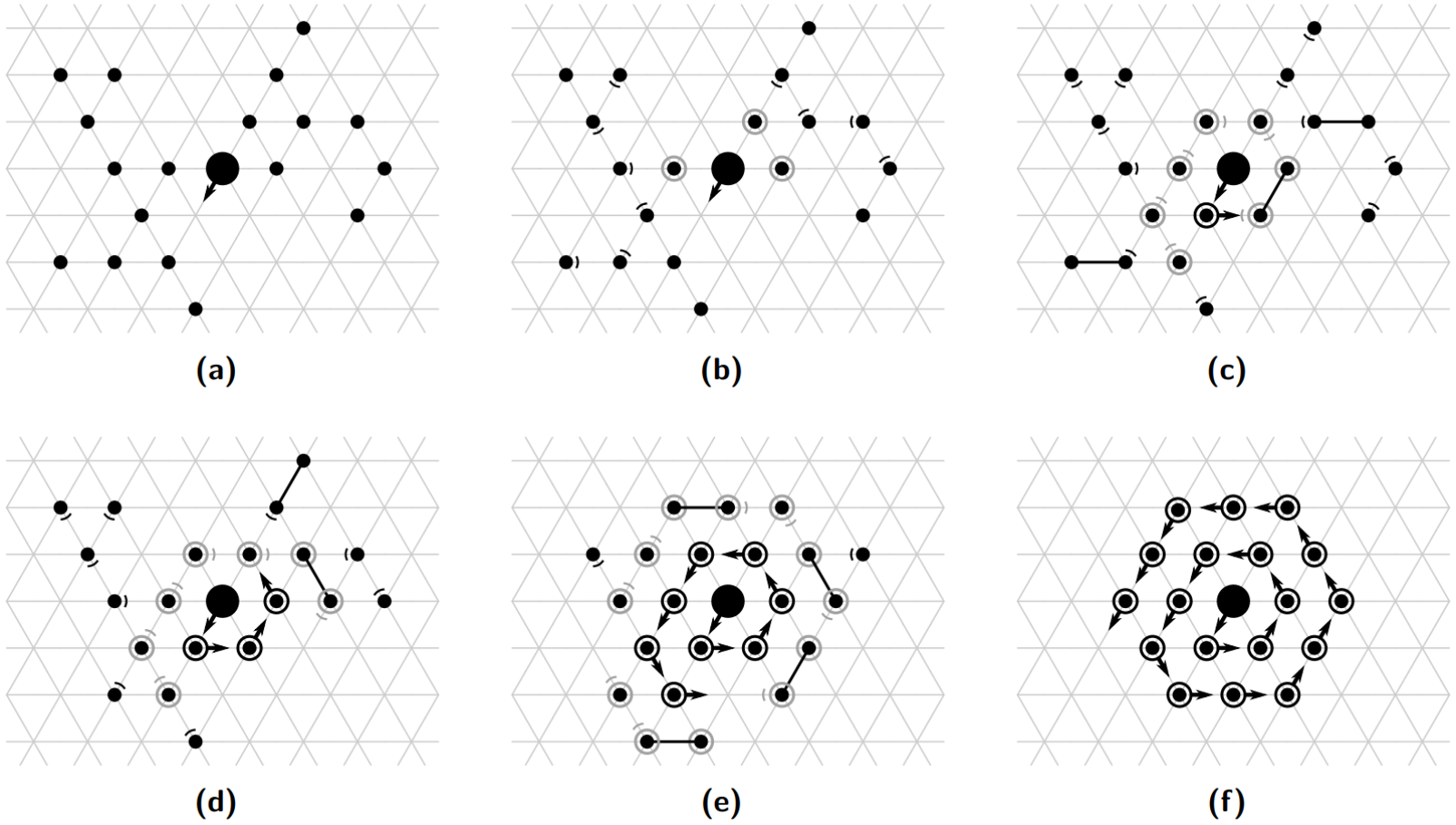
The rest of this talk will primarily focus on unfair asynchronous adversaries, the most general of all adversarial activation models.

Informally: the adversary can activate any amoebot with something to do whenever it wants to.



Asynchronous Hexagon Formation

Problem: Reconfigure any connected amoebot system as a regular hexagon, assuming there is a unique seed amoebot initially in the system.



Asynchronous Hexagon Formation

We formulate a Hexagon-Formation algorithm in terms of actions based on [DGRSS 2015].

Algorithm 1 Hexagon-Formation for Amoebot A

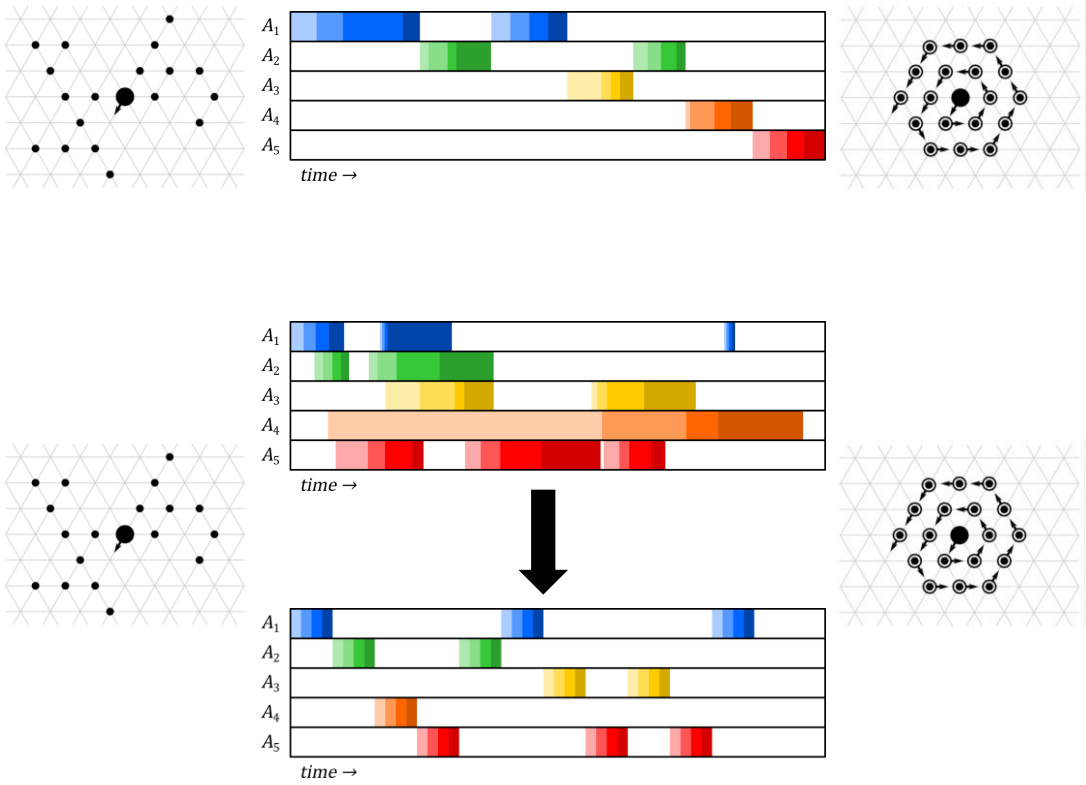
1: $\alpha_1 : (A.\texttt{state} \in \{\text{IDLE}, \text{FOLLOWER}\}) \wedge (\exists B \in N(A) : B.\texttt{state} \in \{\text{SEED}, \text{RETIRED}\}) \rightarrow$
2: WRITE($\bot, \texttt{parent}, \text{NULL}$).
3: WRITE($\bot, \texttt{state}, \text{ROOT}$).
4: WRITE($\bot, \texttt{dir}, \text{GETNEXTDIR}(\text{counter-clockwise})$).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

Asynchronous Hexagon Formation

Theorem. Assuming geometric space, assorted orientations, and constant-size memory, Hexagon-Formation (HF) is correct under any (i.e., an unfair asynchronous) adversary.

Outline of analysis:

- HF is correct under an unfair sequential adversary.
- Enabled actions of HF remain enabled despite concurrent executions.
- Enabled actions of HF are executed identically in sequential and asynchronous executions.
- The action executions of any asynchronous execution of HF can be serialized.
- Any asynchronous execution of HF terminates.



Sufficient Conditions for Concurrent Correctness

Our analysis of Hexagon-Formation shows that any algorithm satisfying

1. correctness under an unfair sequential adversary,
2. enabled actions remaining enabled despite concurrent action executions, and
3. enabled action executions remaining successful/unaffected by concurrent action executions

must also be correct under an unfair asynchronous adversary, the most general of all.

Notably, this is true without using the Lock/Unlock operations, demonstrating that while useful, locks are not always necessary for designing correct concurrent amoebot algorithms.

A General Framework for Concurrency Control

We've shown how individual algorithms can be designed with actions that are ambivalent to the effects of concurrency, but this can be difficult to achieve in general.

Here, we use **locks** to lift correct sequential algorithms to the asynchronous setting.

This is the best of both worlds: the ease of designing algorithms in the sequential setting and the relevance of correct execution in the more realistic concurrent setting.

This is too optimistic and may be impossible to guarantee in general, so instead we only consider algorithms $\mathcal{A}$ that obey certain conventions.

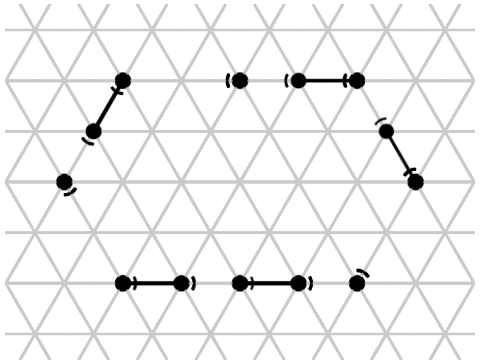
A General Framework for Concurrency Control

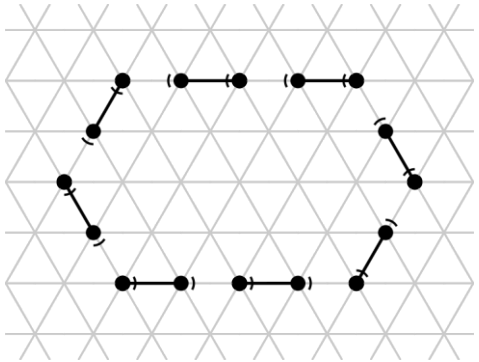
Key Ideas:

- On activation, an amoebot A first attempts to lock its neighborhood.
- If successful, its locked neighbors cannot move or change their memory contents.
- So A can evaluate its guards and perform its actions as if things were sequential (sort of).
- Failed locking attempts and expansions have no effect on the rest of the system.

Key Issue: Locks can't stop amoebots from expanding into an acting amoebot's neighborhood!

Example. "If I have no neighbors, then expand in the 'forward' direction."





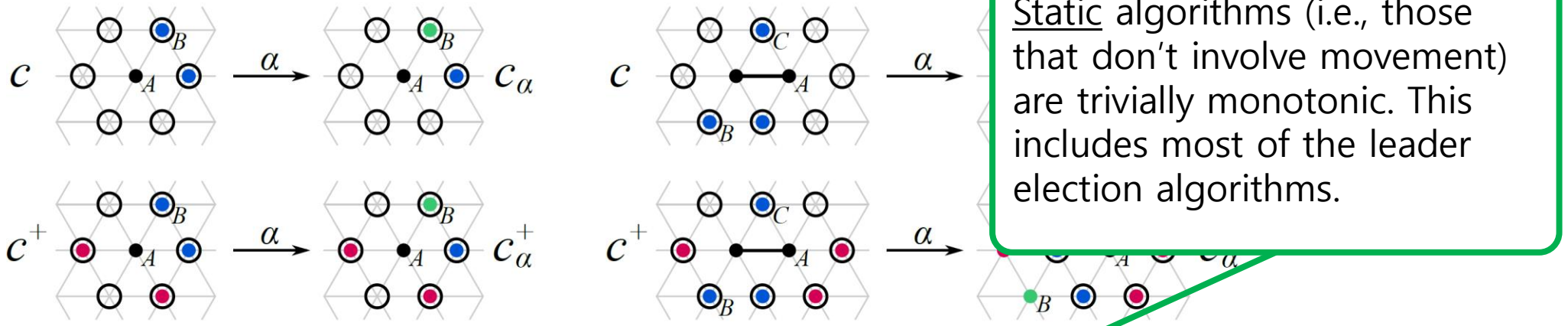
A General Framework for Concurrency Control

We introduce a set of conventions that must be satisfied for the framework to apply.

Convention 1: Any execution of an enabled action must succeed in the sequential setting.

Convention 2: All compute operations must precede at most one movement operation.

Convention 3: **Monotonicity**. Action executions cannot be affected by (unlocked) amoebots that concurrently enter the acting amoebot's neighborhood.



Static algorithms (i.e., those that don't involve movement) are trivially monotonic. This includes most of the leader election algorithms.

Open Question: What amoebot algorithms satisfy monotonicity?

A General Framework for Concurrency Control

Algorithm 2 Concurrency Control Framework for Amoebot A

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

1: Set $g' \leftarrow (A.\texttt{act} = \text{TRUE})$ and $ops' \leftarrow$ "Do:
2: **try:** Set $\mathcal{L} \leftarrow$ LOCK() to attempt to lock A and its 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** $\mathcal{A}' = \{[\alpha' : g' \to ops']\}$.

1. [Failure Point 1] Attempt to Lock.
2. Evaluate all guards of actions in $\mathcal{A}$.
3. If there is an enabled action, emulate its compute phase (Connected, Read, and Write operations) in private memory.
4. [Failure Point 2] If there's an Expand operation, attempt to do so.
5. Perform any Write operations decided on in the compute phase.
6. Handle any other movement types.

Other bookkeeping throughout: Unlocks, setting act/awaken bits, etc.

A General Framework for Concurrency Control

1. Validity. Any execution of an enabled action succeeds in the sequential setting.
2. Computing Before Moving. Compute operations precede movement operations.
3. Monotonicity. Action executions are not affected by (unlocked) amoebots that concurrently enter the acting amoebot's neighborhood.

Theorem. Consider any algorithm $\mathcal{A}$ satisfying Conventions 1-3 and let $\mathcal{A}'$ be the algorithm obtained by the concurrency control framework. If $\mathcal{A}$ terminates under any sequential execution, then every asynchronous execution of $\mathcal{A}'$ terminates in an outcome that some sequential execution of $\mathcal{A}$ also terminates in.

Open Questions

- When existing amoebot algorithms are standardized according to the canonical model's hierarchy of assumptions and described in action semantics, how do they compare?
- How should the canonical model be extended to address fault tolerance? To date, crash faults have been considered for specific problems [DFPSV 2018, DRW 2021].
- We know static algorithms satisfy monotonicity — what about algorithms with movement?
- 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 asynchronous correctness?

Thank you!

sops.engineering.asu.edu

jdaymude.github.io

@joshdaymude