**Dana Angluin · James Aspnes · Zoë Diamadi ·
Michael J. Fischer · René Peralta**

Computation in networks of passively mobile finite-state sensors

Abstract The computational power of networks of small resource-limited mobile agents is explored. Two new models of computation based on pairwise interactions of finite-state agents in populations of finite but unbounded size are defined. With a fairness condition on interactions, the concept of stable computation of a function or predicate is defined. Protocols are given that stably compute any predicate in the class definable by formulas of Presburger arithmetic, which includes Boolean combinations of threshold-k, majority, and equivalence modulo m. All stably computable predicates are shown to be in *NL*. Assuming uniform random sampling of interacting pairs yields the model of conjugating automata. Any counter machine with $O(1)$ counters of capacity $O(n)$ can be simulated with high probability by a conjugating automaton in a population of size n. All predicates computable with high probability in this model are shown to be in *P*; they can also be computed by a randomized logspace machine in exponential time. Several open problems and promising future directions are discussed.

Keywords Diffuse computation · Finite-state agent · Intermittent communication · Mobile agent · Sensor net · Stable computation

1 Scenario: A flock of birds

Suppose we have equipped each bird in a particular flock with a sensor that can determine whether the bird's temper-

D. Angluin · J. Aspnes · Z. Diamadi · M. J. Fischer (✉) · R. Peralta
Department of Computer Science, Yale University, New Haven, CT 06520-8285, USA
E-mail: fischer-michael@cs.yale.edu

ature is elevated or not, and we wish to know whether at least five birds in the flock have elevated temperatures. We assume that the sensors are quite limited: each sensor has a constant number of bits of memory and can respond to a global start signal, and two sensors can communicate only when they are sufficiently close to each other.

In this scenario, the sensors are mobile but have no control over how they move, that is, they are *passively mobile*. Initially, we assume that the underlying pattern of movement gives rise to a communication pattern that guarantees a fairness condition on the resulting computation. Intuitively, it is useful to imagine that every pair of birds in the flock repeatedly come sufficiently close to each other for their sensors to communicate, but it will turn out that this condition is neither necessary nor sufficient for our results. While this intuition is sufficient for understanding the protocol that follows, the reader is urged to read carefully the formal definitions in Sect. 3.

Under these assumptions, there is a simple protocol ensuring that every sensor eventually contains the correct answer. At the global start signal, each sensor makes a measurement, resulting in a 1 (elevated temperature) or 0 (not elevated temperature) in a counter that can hold values from 0 to 4. When two sensors communicate, one of them sets its counter to the sum of the two counters, and the other one sets its counter to 0. If two counters ever sum to at least 5, the sensors go into a special alert state, which is then copied by every sensor that encounters them. The output of a sensor is 0 if it is not in the alert state, and 1 if it is in the alert state. If we wait a sufficient interval after we issue the global start signal, we can retrieve the correct answer from any of the sensors.

Now consider the question of whether at least 5% of the birds in the flock have elevated temperatures. Is there a protocol to answer this question in the same sense, without assumptions about the size of the flock? In Sect. 4, we show that such a protocol exists. More generally, we are interested in fundamental questions about the computational power of this and related models of interactions among members of a distributed population of finite-state agents.

2 A wider view

Most work in distributed algorithms assumes that agents in a system are computationally powerful, capable of storing non-trivial amounts of data and carrying out complex calculations. But in systems consisting of massive amounts of cheap, bulk-produced hardware, or of small mobile agents that are tightly constrained by the systems they run on, the resources available at each agent may be severely limited. Such limitations are not crippling if the system designer has fine control over the interactions between agents; even finite-state agents can be regimented into cellular automata [20] with computational power equivalent to linear space Turing machines. But if the system designer cannot control these interactions, it is not clear what the computational limits are.

Sensor networks are a prime example of this phenomenon. Each sensing unit is a self-contained physical package including its own power supply, processor and memory, wireless communication capability, and one or more sensors capable of recording information about the local environment of the unit. Constraints on cost and size translate into severe limitations on power, storage, processing, and communication. Sensing units are designed to be deployed in large groups, using local low-power wireless communication between units to transmit information from the sensors back to a base station or central monitoring site.

Research in sensor networks has begun to explore the possibilities for using distributed computation capabilities of such networks in novel ways to reduce communication costs. Aggregation operations, such as count, sum, average, extrema, median, or histogram, may be performed on the sensor data in the network as it is being relayed to the base station [14, 16]. Flexible groups of sensors associated with targets in spatial target tracking can conserve resources in inactive portions of the tracking area [8, 24]. Though sensors are usually assumed to be stationary or nearly so, permitting strategies based on relatively stable routing, this assumption is not universal in the sensor-network literature. For example, an assumption of random mobility and packet relay dramatically increases the throughput possible for communication between source-destination pairs in a wireless network [11].

The flock of birds scenario illustrates the question of characterizing what computations are possible in a cooperative network of passively mobile finite-state sensors. The assumptions we make about the motion of the sensors are that it is passive (not under the control of the sensors), too rapid and unpredictable for stable routing strategies to be feasible, and that specified pairs of sensors, given by an *interaction graph*, will repeatedly be close enough to communicate using a low-power wireless signal.

There is a global start signal transmitted by the base station to all the sensors simultaneously to initiate a computation. When they receive the global start signal, the sensors take a reading (one of a finite number of possible input values) and attempt to compute some function or predicate of all the sensor values. This provides a "snapshot" of the sensor values, rather than the continuous stream of sensor values more commonly considered. Sensors communicate in pairs and do not have unique identifiers; thus, they update their states based strictly on the pair of their current states and on the role each plays in the interaction—one acting as initiator and the other as responder.

In Sect. 3, we define a model of computation by pairwise interactions in a population of identical finite-state agents. Assuming a fairness condition on interactions, we define the concept of stable computation of a function or predicate by a population protocol.

In Sect. 4, we consider the question of what predicates can be stably computed when interactions can occur between all pairs of agents. We show how to construct protocols for any Presburger-definable predicate. This is a rich class of arithmetic predicates that includes threshold-k, parity, majority, and simple arithmetic relations. We show that stably computable predicates are closed under the Boolean operations. We also show that every predicate computable in this model is in nondeterministic log space. An open problem is to give an exact characterization of the computational power of stable computation in this model.

In Sect. 5, we show that the all-pairs case is the weakest for stably computing predicates by showing that it can be simulated by any population that cannot be separated into non-interacting subpopulations. The questions of what additional predicates can be computed for reasonable restrictions on the interactions and what properties of the underlying interaction graph can be stably computed by a population are considered in [1].

In Sect. 6, we obtain the model of conjugating automata by adding a uniform sampling condition on interactions to the assumption that interactions are enabled between all pairs of agents. This allows us to consider computations that are correct with high probability and to address questions of expected resource use. We show that this model has sufficient power to simulate, with high probability, a counter machine with $O(1)$ counters of capacity $O(n)$. We further show that Boolean predicates computable with high probability in this model are in P, and can be computed by a randomized logspace machine in exponential time. This gives a partial characterization of the set of predicates computable by such machines, but finding an exact characterization is still open.

In Sect. 7, we describe other related work, and in Sect. 8 we discuss some of the many intriguing questions raised by these models.

3 A formal model

We define a model that generalizes the flock of birds scenario from Sect. 1.

3.1 Population protocols

A *population protocol* $\mathcal{A}$ consists of finite *input and output alphabets* X and Y, a finite set of *states* Q, an *input function*

$I : X \rightarrow Q$ mapping inputs to states, an *output function* $O : Q \rightarrow Y$ mapping states to outputs, and a *transition function* $\delta : Q \times Q \rightarrow Q \times Q$ on pairs of states. If $\delta(p, q) = (p', q')$, we call $(p, q) \mapsto (p', q')$ a *transition*, and we define $\delta_1(p, q) = p'$ and $\delta_2(p, q) = q'$.

Example As a simple illustration, we formalize a version of the count-to-five protocol from Sect. 1. The six states are $q_0, \ldots, q_5$. The input and output alphabets are $X = Y = \{0, 1\}$. The input function I maps 0 to q_0 and 1 to q_1. The output function O maps all states except q_5 to 0 and the state q_5 to 1. The transition function $\delta(q_i, q_j)$ is defined as follows: if $i + j \geq 5$, then the result is (q_5, q_5); if $i + j < 5$ then the result is (q_{i+j}, q_0).

A *population* $\mathcal{P}$ consists of a set A of n *agents*[1] together with an irreflexive relation $E \subseteq A \times A$ that we interpret as the directed edges of an *interaction graph*. E describes which agents may interact during the computation. Intuitively, an edge $(u, v) \in E$ means that u and v are able to interact, with u playing the role of *initiator* and v playing the role of *responder* in the interaction. Note that the distinct roles of the two agents in an interaction is a fundamental assumption of asymmetry in our model; symmetry-breaking therefore does not arise as a problem within the model. Though most of the present paper concerns the case in which E consists of all ordered pairs of distinct elements from A, which is termed the *complete interaction graph*, we give definitions appropriate for general E.

When a population protocol $\mathcal{A}$ runs in a population $\mathcal{P}$, we think of each agent in $\mathcal{P}$ as having a state from $\mathcal{A}$'s state set $Q_{\mathcal{A}}$. Pairs of agents interact from time to time and change their states as a result. Each agent also has a current output value determined by its current state. The collection of all agents' current outputs is deemed to be the current output of the computation. These concepts are made more precise below.

A *population configuration* is a mapping $C : A \rightarrow Q$ specifying the state of each member of the population. Let C and C' be population configurations, and let u, v be distinct agents. We say that C goes to C' via *encounter* $e = (u, v)$, denoted $C \xrightarrow{e} C'$, if

$C'(u) = \delta_1(C(u), C(v))$
$C'(v) = \delta_2(C(u), C(v))$
$C'(w) = C(w)$ for all $w \in A - \{u, v\}$.

We say that C can go to C' in one step, denoted $C \rightarrow C'$, if $C \xrightarrow{e} C'$ for some encounter $e \in E$, and we call $C \rightarrow C'$ a *transition*. We write $C \xrightarrow{*} C'$ if there is a sequence of configurations $C = C_0, C_1, \ldots, C_k = C'$, such that $C_i \rightarrow C_{i+1}$ for all i, $0 \leq i < k$, in which case we say that C' is *reachable* from C.

The *transition graph* $G(\mathcal{A}, \mathcal{P})$ of protocol $\mathcal{A}$ running in population $\mathcal{P}$ is a directed graph whose nodes are all possible population configurations and whose edges are all possible transitions on those nodes. A strongly connected component of a directed graph is *final* iff no edge leads from a node in the component to a node outside. A configuration is *final* iff it belongs to a final strongly connected component of the transition graph.

An *execution* is a finite or infinite sequence of population configurations $C_0, C_1, C_2, \ldots$ such that for each i, $C_i \rightarrow C_{i+1}$. An infinite execution is *fair* if for every possible transition $C \rightarrow C'$, if C occurs infinitely often in the execution, then C' occurs infinitely often.[2] A *computation* is an infinite fair execution.

Lemma 1 *Let* $\Xi = C_0, C_1, C_2, \ldots$ *be a computation of population protocol* $\mathcal{A}$ *running in population* $\mathcal{P}$. *Let* $\mathcal{F}$ *be the set of configurations that occur infinitely often in* Ξ, *and let* $G_{\mathcal{F}}$ *be the subgraph of* $G(\mathcal{A}, \mathcal{P})$ *induced by* $\mathcal{F}$. $G_{\mathcal{F}}$ *is a final strongly connected component of* $G(\mathcal{A}, \mathcal{P})$, *and every element of* F *is final.*

Proof Every $C' \in \mathcal{F}$ is reachable from every $C \in \mathcal{F}$ via a subsequence of transitions in Ξ, so $G_{\mathcal{F}}$ is strongly connected. Suppose $C \rightarrow C'$ and $C \in \mathcal{F}$. By fairness, C' occurs infinitely often in Ξ, so $C' \in \mathcal{F}$. Hence, $G_{\mathcal{F}}$ is final, so every element of $\mathcal{F}$ is also final.   □

3.2 Input-output behavior of population protocols

As with nondeterministic Turing machines, we define notions of input and output, and we define what it means for a population protocol to compute a particular output given a particular input. The input to a population protocol is a mapping that associates an input value with each agent. The output of a population protocol is a mapping that associates an output value with each agent.

Unlike Turing machines, population protocols do not halt, so there is no obvious fixed time at which to view the output of the population. Rather, we say that the computation converges if it reaches a point after which no agent can subsequently change its output value, no matter how the computation proceeds. Convergence is a global property of the population configuration, so individual agents in general do not know when convergence has been reached. However, with suitable stochastic assumptions on the rate at which interactions occur, it is possible to bound the expected number of interactions until the output stabilizes. We explore this approach in Sect. 6.

Formally, an *input assignment* is a function $x : A \rightarrow X$, where A is the set of agents in the population. We let $\mathcal{X} =$

---

[1] Note that we distinguish typographically between a protocol $\mathcal{A}$ and an agent set A. We hope that context will help the reader distinguish between these two similar notations.

[2] Note that this definition is not equivalent to the intuitive notion of fairness, given in Sect. 1, that every permitted encounter between agents takes place infinitely often. Our formal definition only requires that certain configurations appear in a fair execution; it does not specify which encounters give rise to them. On the other hand, it is also not sufficient that every permitted encounter take place infinitely often. We require that infinitely many encounters result in specific configurations C'.

X^A denote the set of all input assignments. The input assignment determines the initial configuration of the protocol. Namely, if $x \in \mathcal{X}$, then the protocol begins in configuration C_x, where $C_x(w) = I(x(w))$ for all agents w.

An *output assignment* is a function $y : A \rightarrow Y$. We let $\mathcal{Y} = Y^A$ denote the set of all output assignments. Each configuration C determines an output assignment y_C, where $y_C(w) = O(C(w))$ for all agents w.

A configuration C is said to be *output-stable* if $y_{C'} = y_C$ for all C' reachable from C. Note that we do not require that $C' = C$, only that the output be the same. An infinite computation *converges* if it contains an output-stable configuration C, in which case we say that it *converges (or stabilizes) to output* $y = y_C$. It is immediate that an infinite computation converges to at most one output, which we call the output of the computation when it exists, and we say that the output is undefined otherwise. Because of the nondeterminism inherent in the choice of encounters, the same initial configuration may lead to different computations that stabilize to different outputs or do not stabilize at all. We say a protocol $\mathcal{A}$ is *always-convergent* if every computation on every input x converges. In this paper, we are only interested in always-convergent protocols.

An always-convergent population protocol $\mathcal{A}$ running in a population $\mathcal{P}$ *stably computes an input-output relation* $R_\mathcal{A}$ as follows. For each $x \in \mathcal{X}$ and $y \in \mathcal{Y}$, $R_\mathcal{A}(x, y)$ holds iff there is a computation of $\mathcal{A}$ beginning in configuration C_x that stabilizes to output y. In the special case that $R_\mathcal{A}$ is single-valued,[3] we write $F_\mathcal{A}(x) = y$ for $R_\mathcal{A}(x, y)$ and say that $\mathcal{A}$ *stably computes the function* $F_\mathcal{A}$.

Example (continued) Continuing our count-to-five illustration, assume that the agents are $u_1, \ldots, u_6$ and the interaction graph is complete. Let the input assignment x be described by the vector

$$(0, 1, 0, 1, 1, 1),$$

assigning input symbols to the agents $u_1, \ldots, u_6$ in that order. The corresponding input configuration is

$$I(x) = (q_0, q_1, q_0, q_1, q_1, q_1),$$

which leads to the following possible computation:

$$(q_0, \underline{q_1}, q_0, \underline{q_1}, q_1, q_1) \xrightarrow{(2,4)} (q_0, \underline{q_2}, q_0, \underline{q_0}, q_1, q_1)$$

$$(q_0, q_2, q_0, q_0, \underline{q_1}, \underline{q_1}) \xrightarrow{(6,5)} (q_0, q_2, q_0, q_0, \underline{q_0}, \underline{q_2})$$

$$(q_0, \underline{q_2}, q_0, q_0, q_0, \underline{q_2}) \xrightarrow{(2,6)} (q_0, \underline{q_4}, q_0, q_0, q_0, \underline{q_0})$$

$$(q_0, \underline{q_4}, \underline{q_0}, q_0, q_0, q_0) \xrightarrow{(3,2)} (q_0, \underline{q_0}, \underline{q_4}, q_0, q_0, q_0).$$

The configurations reachable from the last one above are those with five agents assigned q_0 and one agent assigned q_4, and the outputs of all of them are equal to

$$(0, 0, 0, 0, 0, 0).$$

Therefore, $R((0,1,0,1,1,1), (0,0,0,0,0,0))$ holds, where R is the input-output relation computed by this protocol. In fact, R is single-valued, so we can write

$$F(0, 1, 0, 1, 1, 1) = (0, 0, 0, 0, 0, 0).$$

This protocol illustrates the fact that convergence requires only that the outputs, and not necessarily the configurations, eventually stop changing. We could have solved this particular problem with a protocol in which the configurations themselves also stopped changing, but we do not know how to do that in general.

3.3 Families of populations

In Sect. 3.2, we defined what it means for a population protocol $\mathcal{A}$ running in a fixed population $\mathcal{P}$ to stably compute an input-output relation. It is natural to extend these definitions to families of populations $\{\mathcal{P}_n\}_{n \in \mathbb{N}}$, where $\mathcal{P}_n$ is a population over agent set A_n. Write $\mathcal{X}_n$ and $\mathcal{Y}_n$ for the corresponding input and output assignments on A_n. Then population protocol $\mathcal{A}$ can be regarded as stably computing a family of input-output relations $\{R_\mathcal{A}^n\}_{n \in \mathbb{N}}$. Equivalently, letting $\mathcal{X} = \cup_n \mathcal{X}_n$ and $\mathcal{Y} = \cup_n \mathcal{Y}_n$, $\mathcal{A}$ can be said to stably compute the relation $R_\mathcal{A} = \cup_n R_\mathcal{A}^n \subseteq \mathcal{X} \times \mathcal{Y}$. In the special case that $R_\mathcal{A}$ is single-valued, we write as before $F_\mathcal{A}(x) = y$ for $R_\mathcal{A}(x, y)$ and say that $\mathcal{A}$ stably computes the function $F_\mathcal{A} : \mathcal{X} \rightarrow \mathcal{Y}$.

We now define a family of populations $\{\mathcal{P}_n\}_{n \in \mathbb{N}}$ of particular interest. Let $\mathcal{P}_n$ be the population of size n consisting of the complete interaction graph on the specific agent set $A_n = \{1, \ldots, n\}$. We call A_n the *standard agent set* and $\mathcal{P}_n$ the *standard population* of size n. Because population protocols depend only on the states of agents, not on their names, there is no loss of generality in assuming a fixed agent set.

3.4 Computation on other domains

As defined in Sects. 3.2 and 3.3, a population protocol $\mathcal{A}$ computes a relation $R_\mathcal{A}$ on $\mathcal{X} \times \mathcal{Y}$. We call $\mathcal{X}$ the *natural input domain* and $\mathcal{Y}$ the *natural output domain* for $\mathcal{A}$.

In order to use population protocols to compute on other domains, we need suitable input and output encoding conventions. An *input encoding convention* for domain D_I is a function $E_I : \mathcal{X} \rightarrow D_I$, and an *output encoding convention* for D_O is a function $E_O : \mathcal{Y} \rightarrow D_O$. If $E_I(x) = u$ (respectively $E_O(y) = v$), we say that *x represents u* (respectively *y represents v*). In this terminology, we can define the *natural input and output encoding conventions* to be simply the identity functions on $\mathcal{X}$ and $\mathcal{Y}$, respectively.

E_I and E_O are not required to be either one-to-one or onto. Thus, a given element of D_I (respectively D_O) might have zero, one, or more than one representation in $\mathcal{X}$ (respectively $\mathcal{Y}$). We naturally associate with $R_\mathcal{A}$ the *represented input-output relation* $S_\mathcal{A} \subseteq D_I \times D_O$, where $S_\mathcal{A}(u, v)$ holds iff there exist $x \in \mathcal{X}$ and $y \in \mathcal{Y}$ such that $E_I(x) = u$, $E_O(y) = v$, and $R_\mathcal{A}(x, y)$ holds. We say that $R_\mathcal{A}$ (under the encoding conventions E_I and E_O) is

---

[3] A relation R is single-valued if $\forall x \forall y \forall z (R(x, y) \land R(x, z) \Rightarrow y = z)$.

representative independent iff for all $x_1, x_2 \in \mathcal{X}$ such that $E_I(x_1) = E_I(x_2)$,

$$\{E_O(y) \mid R(x_1, y)\} = \{E_O(y) \mid R(x_2, y)\}.$$

Thus, if $R_\mathcal{A}$ is representative independent and $S_\mathcal{A}(u, v)$ holds, then for all x representing u, there exists y representing v such that $R_\mathcal{A}(x, y)$ holds. We say that $\mathcal{A}$ *stably computes* $S_\mathcal{A}$ if $\mathcal{A}$ is always-convergent and representative independent. In the special case that $S_\mathcal{A}$ is single-valued, we say that $\mathcal{A}$ *stably computes a partial function* $G_\mathcal{A} : D_I \to D_O$.

In words, if $\mathcal{A}$ stably computes $S_\mathcal{A}$, then $S_\mathcal{A}(u, v)$ holds iff for every representation of u, there exists a computation of $\mathcal{A}$ starting from that representation that stabilizes to an output representing v. Moreover, since every computation of $\mathcal{A}$ stabilizes, if $\mathcal{A}$ starts with a representation of some $u \in D_I$, the computation stabilizes to an output that represents some $v \in D_O$. When $S_\mathcal{A}$ is single-valued, the computation stabilizes to an output that represents $G_\mathcal{A}(u)$.

Domain $\mathbb{Z}^k$ Integer input and output values are represented diffusely across the population rather than being stored locally by individual agents. We describe two natural encoding conventions for vectors of integers.

The *symbol-count input convention* assumes an arbitrary input alphabet $X = \{\sigma_1, \ldots, \sigma_k\}$ and $D_I = \mathbb{N}^k$. The k-tuple represented by an assignment $x \in \mathcal{X}$ is $E_I(x) = (n_1, \ldots, n_k)$, where n_i is the number of agents to which x assigns σ_i. Note that the k-tuple $(n_1, \ldots, n_k)$ is only representable in a population of size $n = \sum_i n_i$.

Similarly, the *symbol-count output convention* assumes an arbitrary output alphabet $Y = \{\tau_1, \ldots, \tau_\ell\}$ and $D_O = \mathbb{N}^\ell$ and defines $E_O(y) = (m_1, \ldots, m_\ell)$, where m_i is the number of agents whose current output is τ_i.

The *integer-based input convention* can represent $O(1)$ integers with absolute values bounded by $O(n)$ in a population of size n and can represent $O(1)$ integers of any size in the family of standard populations. It assumes $X \subseteq \mathbb{Z}^k$ and $D_I = \mathbb{Z}^k$ for some k. Thus, input $x \in \mathcal{X}$ assigns a k-tuple of integers $x(w)$ to each agent w. The k-tuple represented by x is $E_I(x) = \sum_{w \in A} x(w)$, the sum across the population of all assigned input tuples.

Note that if X contains the zero vector $0 = (0, 0, \ldots, 0)$ and each of the unit vectors $e_i = (0, \ldots, 1, \ldots, 0)$, where e_i is 0 in all coordinates except for i and 1 at i, then all tuples in $\mathbb{N}^k$ for which the sum of the elements is bounded by n can be represented in a population of size n. If, in addition, X contains $-e_i$ for each i, then all tuples in $\mathbb{Z}^k$ for which the sum of the absolute values of the elements is bounded by n can be so represented.

Similarly, the *integer-based output convention* assumes $Y \subseteq \mathbb{Z}^\ell = D_O$ and defines $E_O(y) = \sum_{w \in A} y(w)$ for output $y \in \mathcal{Y}$.

Example of an integer function We describe a population protocol $\mathcal{A}$ to compute the function $f(m) = \lfloor m/3 \rfloor$, the integer quotient of m and 3. We take $X = Y = \{0, 1\}$. An input assignment x represents $m = E_I(x)$, the number of agents assigned 1, and similarly for output assignments. Given the

standard population $\mathcal{P}_n$, all values of $m \le n$ can be represented, so the partial integer function $G^n_\mathcal{A}(m)$ computed by $\mathcal{A}$ running in $\mathcal{P}_n$ is $f(m)$ restricted to $m \le n$. From this, it easily follows that $\mathcal{A}$ computes f over the family of standard populations.

The states in Q are ordered pairs (i, j) of integers such that $0 \le i \le 2$ and $0 \le j \le 1$. Let C be a configuration. We interpret C as a pair of integers (r, q), where r is the sum over all agents of the first coordinate of the state, and q is the sum of the second coordinate.

The input map I maps 1 to the state $(1, 0)$ and 0 to the state $(0, 0)$. The output map O maps state (i, j) to j. The transition function is defined as follows: $\delta((1, 0), (1, 0)) = ((2, 0), (0, 0))$, and if $i + k \ge 3$ then $\delta((i, 0), (k, 0)) = ((i + k - 3, 0), (0, 1))$. All other transitions are defined to leave the pair of states unchanged.

By induction, one can show that if C is any reachable configuration and (r, q) is the integer pair represented by C, then $m = r + 3q$. Initially, $r = m$ and $q = 0$. Transitions of the first type can accumulate two 1's to a 2 but do not change either r or q. Transitions of the second type reduce r by 3 and increase q by 1, leaving the quantity $r + 3q$ invariant. Eventually, no more transitions of either type will be possible. At this time, $r \le 2$, in which case $q = \lfloor m/3 \rfloor$, as desired. We note that if the output map were changed to the identity (and the output alphabet Y changed accordingly), this protocol would compute the ordered pair $(m \bmod 3, \lfloor m/3 \rfloor)$.

Domain Σ^* Strings inputs are represented diffusely across the population, with the ith input symbol being assigned to the ith agent. We assume an ordered agent set $A = \{a_1, \ldots, a_n\}$ and an arbitrary input alphabet $X = \{\sigma_1, \ldots, \sigma_k\}$. The *string input convention* defines $D_I = X^*$ and $E_I(x) = x(a_1) \cdot \ldots \cdot x(a_n)$, where $x \in \mathcal{X}$.

Predicates A predicate can be regarded as a function whose output is a truth value. The *all-agents predicate output convention* assumes $Y = \{0, 1\}$ and requires *every* agent to agree on the output. Formally, let $\mathbf{0}(w) = 0$ and $\mathbf{1}(w) = 1$ be constant output assignments in $\mathcal{Y}$, and let $D_O = \{\text{false}, \text{true}, \bot\}$. We define

$$E_O(y) = \begin{cases} \text{false} & \text{if } y = \mathbf{0} \\ \text{true} & \text{if } y = \mathbf{1} \\ \bot & \text{otherwise.} \end{cases}$$

Thus, every output assignment in which the agents do not agree represents $\bot$.

Let E_I be an input encoding convention over D_I, and let E_O be the all-agents predicate output convention. We say that protocol $\mathcal{A}$ *stably computes a predicate on* D_I if $\mathcal{A}$ stably computes a total function $G_\mathcal{A} : D_I \to D_O$ and $G_\mathcal{A}(u) \ne \bot$ for any $u \in D_I$. Thus, every computation of $\mathcal{A}$ converges to an output in which all agents have the same output value 0 or 1.

Example The formal count-to-five protocol described above stably computes the predicate of $x \in \mathcal{X}$ that is true iff x assigns 1 to at least 5 different agents.

3.5 Symmetry in standard populations

All agents in standard population $\mathcal{P}_n$ are identical, so it makes no difference to which agent each input symbol is assigned. Under the all-agents predicate output convention, it also makes no difference which agent produces which output symbol since all agents are required to produce the same output.

Formally, a predicate F on $\mathcal{X}$ is *invariant under agent renaming* if $F(x) = F(x \circ \pi)$ for every permutation π on A_n.

Theorem 1 *Every predicate on $\mathcal{X}$ that is stably computable by a population protocol running in standard population $\mathcal{P}_n$ is invariant under agent renaming.*

Proof Suppose population protocol $\mathcal{A}$ running in $\mathcal{P}_n$ computes predicate $G_{\mathcal{A}}$. Let π be a permutation on A_n and $R_{\mathcal{A}}(x, y)$ the input-output relation stably computed by $\mathcal{A}$. Then it is easily shown that $R_{\mathcal{A}}(x \circ \pi, y \circ \pi)$. Since $G_{\mathcal{A}}$ is a predicate under the predicate output convention, the output assignment y is a constant function, so $y \circ \pi = y$. It follows that $y \circ \pi$ and y encode the same output, so $G_{\mathcal{A}}(x \circ \pi) = G_{\mathcal{A}}(x)$ as desired.   □

Language acceptance Let χ_L be the characteristic function of L, that is, $\chi_L(\sigma) = \mathsf{true}$ iff $\sigma \in L$. We say that $\mathcal{A}$ *accepts* L iff $\mathcal{A}$ stably computes χ_L under the string input convention.

We say a language is *symmetric* if it is closed under permuting the letters in a word. The following is immediate from Theorem 1:

Corollary 1 *Let $L \subseteq \Sigma^*$ be a language accepted by a population protocol over the family of standard populations. Then L is symmetric.*

All that matters for acceptance of symmetric languages is the number of occurrences of each input symbol. Let $X = \{\sigma_1, \ldots, \sigma_k\}$ and $\sigma \in X^*$. The *Parikh map* Ψ takes σ to the the vector $(n_1, \ldots, n_k)$, where n_i is the number of times σ_i occurs in σ [21].

Lemma 2 *Let L be a symmetric language over alphabet Σ of size k. Then L is accepted by population protocol $\mathcal{A}$ iff $\Psi(L)$ is stably computed by $\mathcal{A}$ under the symbol-count input convention.*

Proof Immediate from the fact that $E_I(x) = \Psi(\sigma_1, \ldots, \sigma_n)$, where $x \in \mathcal{X}$, E_I is the symbol-count input convention, and x represents $\sigma_1, \ldots, \sigma_n$ under the string input convention. □

In light of Corollary 1 and Lemma 2, we will often identify a language L with the predicate $\Psi(L)$ when talking about population protocols over the family of standard populations and talk loosely about L being accepted under the symbol-count input convention.

3.6 Other predicate output conventions

One might ask whether the class of stably computable predicates on $\mathcal{X}$ changes if we adopt a weaker output convention.

For example, suppose we take $Y = \{0, 1\}$ as in the all-agents predicate output convention, but we change the output encoding function to

$$E_O(y) = \begin{cases} \mathsf{false} & \text{if } y = \mathbf{0} \\ \mathsf{true} & \text{otherwise.} \end{cases}$$

Call this the *zero/non-zero predicate output convention*.

Theorem 2 *Let ψ be predicate on $\mathcal{X}$ and $\mathcal{P}$ a population of size n over the complete interaction graph. There exists a protocol $\mathcal{A}$ that stably computes ψ according to the all-agents predicate output convention iff there exists a protocol $\mathcal{B}$ that stably computes ψ according to the zero/non-zero predicate output convention.*

Proof The forward direction is immediate since the all-agents predicate output convention is more restrictive than the zero/non-zero predicate output convention.

For the converse, assume $\mathcal{B}$ stably computes ψ according to the zero/non-zero predicate output convention. We construct a protocol $\mathcal{A}$ that stably computes ψ according to the all-agents predicate output convention.

Intuitively, we want $\mathcal{A}$ to simulate $\mathcal{B}$ step by step. When $\mathcal{B}$ stabilizes, all agents in $\mathcal{A}$ should eventually choose output 0 if all agents in $\mathcal{B}$ have chosen 0; otherwise, all agents in $\mathcal{A}$ should eventually choose 1. The problem with this approach is that there is no way for the agents of $\mathcal{A}$ to know when $\mathcal{B}$ has stabilized. Hence, we need a subprotocol that runs in parallel with the simulation of $\mathcal{B}$ to monitor $\mathcal{B}$'s outputs and distribute the correct output bit to the agents of $\mathcal{A}$.

The states of $\mathcal{A}$ are triples $\langle \ell, b, q \rangle$, where q is a state of $\mathcal{B}$ and ℓ and b are single bits called "leader" and "output", respectively. We call any agent with $\ell = 1$ a *leader*. Initially $\ell = 1$, $b = 0$, and q is the agent's initial state in protocol $\mathcal{B}$. The output function maps $\langle \ell, b, q \rangle$ to b.

When two agents interact, they update their state fields according to protocol $\mathcal{B}$. The leader fields interact according to the usual leader-election protocol, namely, when two leaders encounter each other, one remains a leader and the other sets its leader bit to 0. Otherwise, the leader bits do not change, with one exception: When a non-leader whose current output in protocol $\mathcal{B}$ is 1 encounters a leader whose current output in protocol $\mathcal{B}$ is 0, the two agents swap leader bits. Finally, the output bit b of a leader always follows its current output in protocol $\mathcal{B}$, that is, at the end of every encounter, the leader updates b accordingly. A non-leader sets its output bit to the output bit of the last leader that it encountered.

This works because eventually $\mathcal{B}$ stabilizes to an output assignment y and there is only a single leader. If one or more agents stabilize to output 1 in $\mathcal{B}$, then leadership transfers to one of those agents and does not change subsequently. If all agents stabilize to output 0 in $\mathcal{B}$, then leadership also does not change subsequently. The leader's output value is 1 or 0 depending on whether the output of $\mathcal{B}$ is greater than 0 or equal to 0. After the leadership and the leader's output value have stabilized, then every other agent assumes the correct

output value upon its next encounter with the leader and does not change it thereafter.   □

Similar leader-based techniques can be used to show that other natural predicate output conventions are also equivalent to the all-agents convention, e.g., representing false by the integer 0 and true by the integer 1 (i.e., one agent has output 1 and the others have output 0).

4 Computing predicates by population protocols

In this section, we explore the predicates that are stably computable by population protocols running in standard populations using the predicate output convention. We consider predicates with both the natural input convention and also the integer input convention. We show that families of predicates related to the well-studied family of Presburger-definable predicates over the integers [22] are all stably computable by population protocols. It is an open problem whether population protocols can compute more. We conclude this section with a theorem that shows our results are not sensitive to reasonable changes in the conventions used for representing the output of predicates.

4.1 Boolean closure of population predicates

We begin by showing that the family of population-computable predicates is closed under the Boolean operations.

Lemma 3 *Let X be an input set, and let E_I be an input encoding convention over domain D_I. Let F and G be predicates over D_I that are stably computable by population protocols over X. Let ξ be any 2-place Boolean function. Then the predicate $\xi(F, G)$ is stably computable by a population protocol with input set X.*

Proof Let $\mathcal{A}$ stably compute F and $\mathcal{B}$ stably compute G. We assume that $\mathcal{A}$ and $\mathcal{B}$ have the same input set X. We construct a population protocol $\mathcal{C}$, also with input set X, to stably compute $\xi(F, G)$.

$\mathcal{C}$ is the parallel composition of $\mathcal{A}$ and $\mathcal{B}$, together with a suitably chosen output function. Let $Q_\mathcal{A}$ and $Q_\mathcal{B}$ be the states of $\mathcal{A}$ and $\mathcal{B}$, respectively. Let $\mathcal{C}$ have states $Q_\mathcal{C} = Q_\mathcal{A} \times Q_\mathcal{B}$. The input function $I_\mathcal{C}$ maps $s \in X$ to state $(I_\mathcal{A}(s), I_\mathcal{B}(s))$. The transition function $\delta_\mathcal{C}$ is defined by

$$\delta_\mathcal{C}((p_1, p_2), (q_1, q_2)) = ((p_1', p_2'), (q_1', q_2'))$$

where

$$\delta_\mathcal{A}(p_1, q_1) = (p_1', q_1') \quad \text{and} \quad \delta_\mathcal{B}(p_2, q_2) = (p_2', q_2').$$

The output function applies ξ to the outputs of the two component protocols. That is,

$$O_\mathcal{C}((q_1, q_2)) = \xi(O_\mathcal{A}(q_1), O_\mathcal{B}(q_2)).$$

We must show that $\mathcal{C}$ stably computes $\xi(F, G)$. Every fair execution of $\mathcal{C}$ projects onto a fair execution of $\mathcal{A}$ (respectively $\mathcal{B}$) by erasing the second (respectively first) component of each state pair. Since every fair execution of $\mathcal{A}$ and $\mathcal{B}$ converges, then also every fair execution of $\mathcal{C}$ converges.

Now, suppose a fair execution of $\mathcal{C}$ stabilizes to output assignment $y_\mathcal{C}$. Let $y_\mathcal{A}$ and $y_\mathcal{B}$ be the stable outputs of the corresponding embedded computations of $\mathcal{A}$ and $\mathcal{B}$, respectively. Since $\mathcal{A}$ and $\mathcal{B}$ both compute predicates according to the predicate output convention, then all agents agree on the output in each embedded computation, and $y_\mathcal{A}$ and $y_\mathcal{B}$ each represent a truth value $b_\mathcal{A}$ and $b_\mathcal{B}$, respectively. By the definition of $O_\mathcal{C}$, it follows that $y_\mathcal{C}$ represents the truth value $\xi(b_\mathcal{A}, b_\mathcal{B})$. Since $\mathcal{A}$ stably computes F and $\mathcal{B}$ stably computes G, it follows that $\mathcal{C}$ stably computes $\xi(F, G)$, as desired.   □

Corollary 2 *Any Boolean formula over stably computable predicates with a common input set X is stably computable.*

Proof Immediate by repeated application of Lemma 3.   □

4.2 Presburger definable predicates

Presburger arithmetic [9, 10, 15, 22] is the first-order theory of the integers under addition and less than. It is a rich but decidable theory, enabling one to define predicates such as parity and majority. In this section, we review the properties of Presburger arithmetic and the closely-related semilinear sets. In the next section, we show that every predicate definable in Presburger arithmetic is stably computable by population protocols.

The usual definition of Presburger arithmetic considers a first-order logical language with one function symbol "+", constants "0" and "1", predicate symbols "=" and "<", the usual logical operators of "∧", "∨", and "¬", variables $x_1, x_2, \ldots$, and quantifiers "∀" and "∃". Formulas are interpreted with quantifiers ranging over the integers. "+" is usual integer addition. "0" and "1" have their usual meanings as integers. "=" and "<" are interpreted as the integer relations of equality and less than.

A formula $\phi(x_1, \ldots, x_k)$ with free variables $x_1, \ldots, x_k$ *defines* a predicate $F_\phi : \mathbb{Z}^k \to \{0, 1\}$ as follows: For integers $u_1, \ldots, u_k$, $F_\phi(u_1, \ldots, u_k) = 1$ if $\phi(x_1, \ldots, x_k)$ evaluates to true when $x_1, \ldots, x_k$ are interpreted as $u_1, \ldots, u_k$, respectively, and $F_\phi(u_1, \ldots, u_k) = 0$ otherwise.

The predicates definable in Presburger arithmetic are closely related to the semilinear sets. A set $L \subseteq \mathbb{N}^k$ is *linear* if there are vectors $v_0, v_1, \ldots, v_m \in \mathbb{N}^k$ such that

$$L = \{v_0 + \kappa_1 v_1 + \cdots + \kappa_m v_m \mid \kappa_1, \ldots, \kappa_m \in \mathbb{N}\}.$$

A set is *semilinear* if it is the finite union of linear sets.

Theorem 3 (Ginsburg and Spanier) *A subset of $\mathbb{N}^k$ is semilinear iff it is definable in Presburger arithmetic.*

Proof This was proved originally by Ginsburg and Spanier [10]. Kracht gives a more recent simplified proof [15].   □

Although Presburger arithmetic seems to talk only about addition, the use of quantifiers allows some predicates involving multiplication and division to be defined. Let m be a constant, and let $\equiv_m$ be the 2-place predicate such that $x \equiv_m y$ holds iff $x \equiv y \pmod{m}$. This can be defined by a formula $\xi_m(x, y)$ as follows. For any variable or constant q, let $\underline{m}q$ be the expression that adds together m copies of q, i.e., $\underline{m}q = \underbrace{q + q + \cdots + q}_{m \text{ times}}$. Then

$$\xi_m(x, y) \overset{\text{df}}{=} \exists z \exists q((x + z = y) \wedge \underline{m}q = z).$$

$\xi_m(x, y)$ is satisfied only when $z = y - x$ and $q = z/m$. Such integers q and z exist exactly when $x \equiv y \pmod{m}$, as desired.

An *extension* of an interpreted first-order theory results from augmenting the theory with new predicates and new symbols to denote them. An extension that does not change the class of definable predicates is called *conservative*. Let *extended Presburger arithmetic* result from augmenting Presburger arithmetic with relation symbols $\equiv_m$ denoting equivalence modulo m, for $m \geq 2$.

Lemma 4 *Extended Presburger arithmetic is a conservative extension of Presburger arithmetic.*

Proof Immediate from the fact that ξ_m defines $\equiv_m$. □

Our definition of ξ_m makes essential use of quantifiers. Rather surprisingly, once we augment Presburger arithmetic with $\equiv_m$, quantifiers are no longer needed.

Theorem 4 (Presburger) *Every definable predicate of Presburger arithmetic can be defined in the extended language by a quantifier-free formula.*

Proof Presburger, in his original 1929 paper [22], shows the decidability of closed formulas of Presburger arithmetic without the "$<$" operator. His proof method is to transform any closed formula[4] into an easily-decided normal form in which the only quantifiers appear in subformulas expressing $\equiv_m$. While he does not explicitly consider either our extended language or predicates definable by open formulas, his methods would seem to easily extend to our case.

It is unclear where our form of Theorem 4 first appears, although it is well known in the folklore. This result was mentioned in Ginsburg and Spanier [10] and probably elsewhere. Kracht presents a proof [15] that he attributes to Monk [19]. □

Example We now return to the question raised at the end of Sect. 1 of whether at least 5% of the birds in the flock have elevated temperatures. Using the symbol-count input convention, the sensors in the flock encode a pair (x_0, x_1), where x_0 is the number of birds with normal temperatures and x_1 is the number of birds with elevated temperatures. The question we wish to answer is whether $x_1 \geq 0.05(x_0 + x_1)$. This is easily seen to be equivalent to the predicate $20x_1 \geq x_0 + x_1$. It will follow from Theorem 5 that this predicate is stably computable.

---

[4] A closed formula is one with no free variables.

4.3 Computing Presburger predicates by population protocols

In this section, we show that every Presburger-definable predicate is stably computable by a population protocol using the integer input encoding convention. We first show that all Presburger definable predicates under the symbol-count input convention are stably computable. We then use this result to show the computability of all Presburger definable predicates under the integer input convention.

Lemma 5 *Let $X = \{\sigma_1, \ldots, \sigma_k\}$ be an arbitrary input alphabet. Let a_i, c, and m be integer constants with $m \geq 2$. Then the following predicates on non-negative integers $x_1, \ldots, x_k$ are stably computable in the family of standard populations under the symbol-count input convention:*

1. $\sum_i a_i x_i < c$.
2. $\sum_i a_i x_i \equiv c \pmod{m}$.

Proof We define population protocols for computing the two predicates as follows. Let $s = \max(|c| + 1, m, \max_i |a_i|)$, where m is taken to be 0 for the threshold protocol. In both protocols, the state space Q is the set $\{0, 1\} \times \{0, 1\} \times \{u \in \mathbb{Z} : -s \leq u \leq s\}$, and the input function I maps $\sigma_i \in X$ to $(1, 0, a_i)$. The first bit in each state is a "leader bit" that is used to elect a unique leader who collects the value of the linear combination. The second bit is an output bit that records for each agent the output value computed by the last leader it encountered. The third field is a "count" field used to accumulate the linear combination of the x_i on the left-hand side. The output map O simply maps $(\cdot, b, \cdot)$ to b.

We now give the transition rules for the two protocols and prove their correctness. We start with the threshold protocol, as the analysis is more involved; we then argue the correctness of the remainder protocol by analogy with the argument of the threshold protocol.

For any integers u, u' with $-s \leq u, u' \leq s$, define

$$q(u, u') = \max(-s, \min(s, u + u'))$$

and

$$r(u, u') = u + u' - q(u, u').$$

Observe that both $q(u, u')$ and $r(u, u')$ lie in the range $[-s, \ldots, s]$ and that $q(u, u') + r(u, u') = u + u'$. Let $b(u, u')$ be 1 if $q(u, u') < c$ and 0 otherwise.

The transition rule is given by the formula

$$(\ell, \cdot, u), (\ell', \cdot, u') \mapsto (1, b(u, u'), q(u, u')), (0, b(u, u'), r(u, u'))$$

if at least one of ℓ or ℓ' is 1. If both ℓ and ℓ' are zero, the encounter has no effect.

Informally, the initiator becomes a leader if either agent was a leader before the transition; the transition assigns as much of the sum of u and u' to the initiator as possible, with the remained assigned to the responder. The output bits of both agents are set to 1 if and only if the part of the sum

assigned to the initiator is less than c. We now show that all output values converge to the truth value ($\sum_i a_i x_i < c$) by proving a sequence of claims about any fair execution.

The protocol converges to a single leader Define $\Lambda(C)$ to be the set of agents whose leader bit equals 1 in configuration C. Then $|\Lambda(C_0)| = n$. Any encounter between two leaders reduces $|\Lambda(C)|$ by one, and no encounter increases $|\Lambda(C)|$. By the fairness condition, if there are two leaders, they eventually meet. It follows that after finitely many steps $|\Lambda(C)| = 1$.

The single leader's value converges to $\max(-s, \min(s, \sum_i a_i x_i))$ For each agent j let $u_j(C)$ be the value of its count field in configuration C. From the definition of the input mapping I, we have $\sum_j u_j(C_0) = \sum_i a_i x_i$, where C_0 is the initial configuration. Because the transition rule preserves the sum of the count fields of the two participating agents, $\sum_j u_j(C)$ continues to equal $\sum_i a_i x_i$ throughout the computation.

For a given configuration C, define $\Lambda(C)$ as above to be the set of agents that are leaders, and define $p(C) = \sum_{j \notin \Lambda(C)} |u_j(C)|$. For notational simplicity, and when the meaning is clear, we will write p instead of $p(C)$ and u_i instead of $u_i(C)$. Call a configuration C *stable* if there is a unique leader ℓ and one of the following conditions holds:

1. $p = 0$.
2. $u_\ell = s$, and $u_j \geq 0$ for all $j \neq \ell$.
3. $u_\ell = -s$, and $u_j \leq 0$ for all $j \neq \ell$.

By checking the three cases, it is not hard to see that in a stable configuration, $u_\ell = \max(-s, \min(s, \sum_i a_i x_i))$.

We will now show that the protocol converges to a stable configuration by showing that from any configuration with a unique leader that is not stable, there is a transition that reduces p, and no transition increases p. We let ℓ continue to be the identity of the leader.

Suppose $u_\ell = s$, and there is some $j \neq \ell$ for which $u_j < 0$; then an encounter between ℓ and j sets the count field of the initiator (which becomes the leader) to $s + u_j$ and sets the count field of the responder to 0. This reduces p by $-u_j > 0$. If, on the other hand $u_\ell = -s$ and there is some $j \neq \ell$ for which $u_j > 0$, then an encounter between ℓ and j again sets the count field of the responder to 0, reducing p. If $-s < u_\ell < s$ and there is some $j \neq \ell$ with $u_j \neq 0$, then in an encounter between ℓ and j either (a) $u_j > 0$, the initiator's count becomes $\min(u_\ell + u_j, s) = u_\ell + \min(u_j, s - u_\ell)$, and p drops by $\min(u_j, s - u_\ell) > 0$; or (b) in the symmetric case $u_j < 0$, p drops by $\min(-u_j, s + u_j) > 0$. So in any configuration with a single leader that is not stable, there exists a transition that reduces p; by fairness, a transition that reduces p eventually occurs.

It remains to show that other transitions will not increase p. The remaining possible transitions are (a) those between two non-leaders, which are no-ops and thus do not affect p; (b) those that involve a leader ℓ with $u_\ell = s$ and an agent j with $u_j \geq 0$, which do not change p because in such cases

the initiator becomes a leader with count $q(s, u_j) = s$ and the responder receives $r(s, u_j) = u_j$; and (c) those that involve a leader ℓ with $u_\ell = -s$ and an agent j with $u_j \leq 0$, which are symmetric to the previous case. These last two cases also demonstrate that once a stable configuration with a unique leader ℓ with $|u_\ell| = s$ is reached, the value held by the leader does not change. For a stable configuration with $|u_\ell| < s$, the fact that $p = 0$ implies that the leader never encounters a nonzero count in another agent, so again the leader's value never changes.

Since p is non-negative, bounded, never rises, and eventually falls in any non-stable configuration with a unique leader, it follows that the protocol eventually converges to a stable configuration once a unique leader exists.

Convergence of the output fields to the correct value. In a stable configuration, if $\sum_i a_i x_i < c$, then the leader's count u_ℓ is either $\sum_i a_i x_i$ or $-s < c$. In either case $b(u_\ell + u_j)$ gives the correct output, and any encounter between a leader and another agent sets the output fields of both agents to 1. No other transition sets the output field of any agent to 0, and by fairness the leader eventually encounters all other agents; it follows that after some finite interval, all agents output 1. Alternatively, if $\sum_i a_i x_i \geq c$, then the leader's count u_ℓ is either $\sum_i a_i x_i$ or s; in either case encounters between the leader and another agent sets both agents' outputs to 0, and again all agents eventually converge, this time to 0. This completes the the proof of correctness for the threshold protocol.

We now turn to the remainder protocol. Here the transition rule is given by the formula

$$(\ell, \cdot, u), (\ell', \cdot, u') \mapsto (1, b, (u + u') \bmod m), (0, b, 0),$$

if at least one of ℓ or ℓ' is 1, where b is 1 if $u + u' \equiv c$ (mod m) and 0 otherwise. If both ℓ and ℓ' are zero, the encounter has no effect.

Repeating the argument for the threshold algorithm, we immediately see that the protocol eventually converges to a single leader. Inspection of the transition rule reveals that $(\sum_j u_j(C)) \bmod m$ is invariant throughout the protocol, and that any non-leader has count 0. It follows that when a single leader exists, its count field is exactly $(\sum_j u_j(C_0)) \bmod m = (\sum_i a_i x_i) \bmod m$. Further encounters between the single remaining leader and other agents eventually set all output fields to $\sum_i a_i x_i \equiv c$ (mod m), as claimed. $\square$

Theorem 5 *Any Presburger-definable predicate on non-negative integers is stably computable in the family of standard populations under the symbol-count input convention.*

Proof Given a Presburger formula Φ, apply Theorem 4 to convert it to a quantifier-free formula Φ' over the extended language described in Sect. 4.2. This formula Φ' will be a Boolean formula over predicates that can be written in one of the following three forms:

$$\sum a_i x_i + c_1 < \sum b_i x_i + c_2 \tag{1}$$

$$\sum a_i x_i + c_1 = \sum b_i x_i + c_2 \tag{2}$$

$$\sum a_i x_i + c_1 \equiv_m \sum b_i x_i + c_2 \qquad (3)$$

If we can show that each such predicate is stably computable, then Φ' is stably computable by Corollary 2.

By rearranging terms, predicates of the form (1) involving inequalities can be rewritten as

$$\sum d_i x_i < c,$$

where each $d_i = a_i - b_i$ and $c = c_2 - c_1$; such predicates can be stably computed by the first case of Lemma 5.

Predicates of the form (2) involving equality can be replaced by the AND of a pair of predicates:

$$\sum a_i x_i + c_1 < \sum b_i x_i + c_2 + 1$$
$$\sum a_i x_i + c_1 > \sum b_i x_i + c_2 - 1$$

These two predicates can then be stably computed by the first case of Lemma 5 and their AND can be stably computed by Lemma 3.

Predicates of the form (3) can be rewritten as

$$\sum d_i x_i \equiv_m c,$$

where c and the d_i are defined as in the first case; such predicates can be stably computed by the second case of Lemma 5. □

Theorem 5 places strong restrictions on the input, and it would appear that it would only permit computing Presburger-definable predicates on non-negative values that sum to less than n. However, it is possible to extend the result of Theorem 5 to the integer-based input convention by building a translator for the integer-based input convention into the Presburger formula itself. The result is:

Corollary 3 *Any Presburger-definable predicate on $\mathbb{Z}^k$ is stably computable in the standard population $\mathcal{P}_n$ with the integer-based input convention.*

Proof Let $\Phi(y_1, \ldots, y_k)$ be a Presburger-definable predicate on $\mathbb{Z}^k$. We will convert Φ to a new Presburger-definable predicate over free variables $x_{\vec{v}}$, where each variable $x_{\vec{v}}$ counts the occurrence of specific tokens representing each k-vector $\vec{v} = \langle v_1, v_2, \ldots, v_k \rangle$ in X.

Recall that in the integer-based input convention, each y_i is the sum over all agents of the i-th vector coordinate. Define

$$\Phi' = \exists y_1, \ldots, y_k : \Phi(y_1, \ldots, y_k) \wedge \bigwedge_{i=1}^{k} \left(y_i = \sum_{\vec{v} \in X} v_i x_{\vec{v}} \right).$$

Observe that the values v_i in each sum are constants, so that Φ' is a formula in Presburger arithmetic, which is stably computable on the standard population by Theorem 5. Observe further that Φ' is true if and only if Φ is satisfied by a set of values $y_1, \ldots, y_k$ that are equal to the integer values given by the integer-based input convention. It follows that Φ is stably computable. □

Example Consider the Presburger predicate

$$\Phi(y_1, y_2) \stackrel{\mathrm{df}}{=} (y_1 - 2y_2 \equiv 0 \pmod 3).$$

Let

$$X = \{(0, 0), (1, 0), (-1, 0), (0, 1), (0, -1)\}$$

be an input alphabet. The related predicate

$$\Phi' \stackrel{\mathrm{df}}{=} \exists y_1, y_2 \big(y_1 - 2y_2 \equiv 0 \pmod 3$$
$$\wedge y_1 = x_{(1,0)} - x_{(-1,0)}$$
$$\wedge y_2 = x_{(0,1)} - x_{(0,-1)} \big)$$

has five free variables $x_{(u,v)}$, one for each $(u, v) \in X$. Let E_I^{int} be the integer input convention and E_I^{SC} be the symbol-count input convention on the same set X. It is easily verified that

$$\Phi'\big(E_I^{\mathrm{SC}}(x)\big) = \Phi\big(E_I^{\mathrm{int}}(x)\big)$$

for every $x \in \mathcal{X}$.

Corollary 4 *A symmetric language $L \subseteq X^*$ is accepted by a population protocol if its image under the Parikh map is a semilinear set.*

Proof Let $L \subseteq X^*$ be a symmetric language whose image under the Parikh map Ψ is a semilinear set. From Theorem 3, $\Psi(L)$ is definable in Presburger arithmetic. From Theorem 5, there is a protocol $\mathcal{A}$ to stably compute $\Psi(L)$ under the symbol-count input convention. From Lemma 2, $\mathcal{A}$ accepts L. □

4.4 Predicates not stably computable

Theorem 5 gives a partial characterization of the stably computable predicates in the population model with all pairs enabled. We do not know if this characterization is complete. However, we can obtain an upper bound on the set of predicates stably computable in this model by showing that it is contained in the complexity class *NL*.

Because stably computable predicates in this model are symmetric, it is sufficient to represent a population configuration by the multiset of states assigned to the agents. Since there are $|Q|$ possible states and the population consists of n agents, each population configuration can thus be represented by $|Q|$ counters of $\lceil \log n \rceil$ bits each. A population protocol step can be simulated by drawing two elements from the multiset, applying the transition function and returning the resulting two elements to the multiset.

Suppose there is a population protocol $\mathcal{A}$ that stably computes a predicate F in the family of standard populations. Define L_F to be the set of strings x such that $F(x) = 1$, where we interpret a string x of length n as an element of X^n. We describe a nondeterministic Turing machine to

accept L_F in space $O(\log n)$. To accept input x, the Turing machine must verify two conditions: that there is a configuration C reachable from $I(x)$ in which all states have output 1, and there is no configuration C' reachable from C in which some state has output 0. The first condition is verified by guessing and checking a polynomial-length sequence of multiset representations of population configurations reaching such a C. The second condition is the complement of a similar reachability condition. It is in nondeterministic $O(\log n)$ space because this class is closed under complement [13]. It follows that:

Theorem 6 *All predicates stably computable in the model with all pairs enabled are in the class NL.*

It is an open problem to characterize exactly the power of this model of stable computation. Concretely, we conjecture that predicates such as "x is a power of 2" and "$z = x \times y$" are not stably computable by population protocols. Our intuition is that the model lacks the ability to sequence or iterate computations, and we suspect that a pumping lemma of some form exists for the model.

5 Computation with restricted interactions

Some interaction graphs may permit very powerful computations by population protocols; for example, a population whose interaction graph is a directed line can easily simulate a linear-space Turing machine. In this section, we prove that the complete interaction graph we have been assuming up until now is in a sense the *weakest* structure for stably computing predicates, in that any predicate that is stably computable in a complete interaction graph can also be computed in any weakly-connected interaction graph.

Theorem 7 *For any population protocol $\mathcal{A}$, there exists a population protocol $\mathcal{A}'$ such that for every n, if $\mathcal{A}$ stably computes predicate ψ on the standard population $\mathcal{P}_n$, and if $\mathcal{P}'$ is any population with agents $1,2,\ldots,n$ and a weakly-connected interaction graph, then $\mathcal{A}'$ stably computes ψ on $\mathcal{P}'$.*

We present the proof in the following sections. First, we construct the simulator $\mathcal{A}'$. Next, we relate the reachable configurations in $\mathcal{A}$ to the reachable configurations in $\mathcal{A}'$. We then conclude that $\mathcal{A}'$ correctly computes ψ.

Construction of $\mathcal{A}'$ First assume without loss of generality that n is at least 4; we will need this assumption to avoid getting our agents tangled. The case where $n < 4$ can be handled by a parallel simulation that collects up to three input values together, computes the resulting output by table lookup, and overrides the output of the main simulation if it (stably) computes that n is indeed less than 4.

The computation of $\mathcal{A}$ is simulated using one agent in $\mathcal{P}'$ to hold the state of each agent in $\mathcal{P}_n$. Simulated agents migrate from agent to agent in $\mathcal{P}'$; this allows any two simulated agents to interact infinitely often. The key idea is to

have any interaction in $\mathcal{A}'$ choose nondeterministically between swapping the states of the two interacting agents or simulating an interaction in $\mathcal{A}$; most of the details of the simulation involve implementing this nondeterministic choice with deterministic transitions. To do so, the state space in $\mathcal{A}'$ is augmented to add two "batons", S (for the initiator) and R (for responder) which move somewhat independently of the simulated agents. The presence or not of the two batons is used to control what effect an interaction has: an interaction that involves no batons swaps the states; an interaction that involves one baton moves the baton; and an interaction that involves both batons simulates a transition in $\mathcal{A}$.

Formally, let $\mathcal{A}$ have input alphabet X, output alphabet Y, state space Q, input function I, output function O, and transition function δ. Define $Q' = Q \times \{D, S, R, -\}$ where D is a default initial state of the baton field, S marks the initiator baton, R marks the responder baton, and $-$ marks a "blank" or absent baton. To avoid a profusion of parentheses we will write ordered pairs in Q' using simple concatenation, e.g., qD for (q, D). The transition function δ' is shown in Fig. 1. Finally, define $I'(X) = I(X)D$ and $O'(qB) = O(q)$. Let $\mathcal{A}'$ be the population protocol $(X, Y, Q', I', O', \delta')$.

Group (a) transitions consume all initial D batons, producing at least one S and at least one R baton; group (b) eventually reduces the set of non-blank batons to exactly one S and one R. The remaining groups implement (c) baton movement, (d) state swapping, and (e) $\mathcal{A}$-transitions. Note that group (e) transitions also swap batons; this is done to allow S and R batons to pass each other in narrow graphs, which may be necessary to bring duplicates together in the initial stage.

Note that the group of an $\mathcal{A}'$-transition can be uniquely identified by looking at the changes to the baton fields. If the number of D batons decreases, it is group (a). If the number of S or R batons decreases, it is group (b). If exactly one R or S moves from one agent to another, it is group (c). If the

Group (a):	(xD, yD)	$\mapsto$	(xS, yR)
	$(xD, y*)$	$\mapsto$	$(x-, y*)$
	$(x*, yD)$	$\mapsto$	$(x*, y-)$
Group (b):	(xS, yS)	$\mapsto$	$(xS, y-)$
	(xR, yR)	$\mapsto$	$(xR, y-)$
Group (c):	$(xS, y-)$	$\leftrightarrow$	$(x-, yS)$
	$(xR, y-)$	$\leftrightarrow$	$(x-, yR)$
Group (d):	$(x-, y-)$	$\leftrightarrow$	$(y-, x-)$
Group (e):	(xS, yR)	$\mapsto$	$(x'R, y'S)$
	(yR, xS)	$\mapsto$	$(y'S, x'R)$

Key: x and y range over all states in Q.
      $*$ represents any non-D baton.
      $(x', y') = \delta(x, y)$.

Fig. 1 Transition function δ' for simulator in proof of Theorem 7

batons don't change, it is group (d). If an S and R switch places, it is group (e).

We now make precise the sense in which $\mathcal{A}'$ "simulates" $\mathcal{A}$. A simulated $\mathcal{A}$-configuration C is obtained by ignoring both the batons and agent order in an $\mathcal{A}'$-configuration C'. Let $\rho(C')$ be the configuration C obtained from C' by erasing the second component of each agent's state in C', that is, for all $a \in A$, if $C'(a) = pB$, then $C(a) = p$. Let π be a permutation of agents A. For any $\mathcal{A}$ or $\mathcal{A}'$-configuration C_1, let $\pi(C_1) = C_2$, where $C_2(\pi(a)) = C_1(a)$. Say C is a Q-restriction of C' if there is a permutation π of the agents A such that $\pi(\rho(C')) = C$; in other words, the Q components of the C' states equal the C states modulo reordering the population members.

Call two configurations C and $\bar{C}$ of $\mathcal{A}$ equivalent, written $C \equiv \bar{C}$, if $\bar{C} = \pi(C)$ for some permutation π of the agents. For convenience, we extend the definition of equivalence to the union of $\mathcal{A}$- and $\mathcal{A}'$-configurations. If C is an $\mathcal{A}$-configuration and C' is an $\mathcal{A}'$-configuration, then let $C \equiv C'$ if C is a Q-restriction of C', and close $\equiv$ under reflexivity, symmetry, and transitivity. For $\mathcal{A}'$-configurations C_1' and C_2', it then follows that $C_1' \equiv C_2'$ iff $\rho(C_1) \equiv \rho(C_2)$.

Call an $\mathcal{A}'$-configuration *clean* if it has exactly one S and one R baton and no D batons.

Lemma 6 *Let C' be any configuration of $\mathcal{A}'$ reachable from an initial configuration C_0'. Then $C' \overset{*}{\to} D'$ for some clean configuration D'.*

Proof Either $C' = C_0$ or C' contains at least one S baton and at least one R baton. This is because the only transitions that can be applied to C_0 change two D batons into an S and an R, respectively, and no subsequent transition can remove the last S or the last R.

Starting from C', apply group (a) transitions to C' to remove all D batons. If there are two or more S batons, apply group (c) and group (e) transitions to bring them to adjacent nodes, and apply a group (b) transition to eliminate one. Repeat until only one S baton remains. In a similar way, repeatedly eliminate R batons until only one remains. Let D' be the resulting configuration. D' contains exactly one S and one R baton and no D batons, as desired. □

Lemma 7 *Let C' be final in $\mathcal{A}'$. Then C' is clean.*

Proof By Lemma 6, there is a clean configuration D' reachable from C'. Since C' is final, then so is D', and C' is reachable from D'. No $\mathcal{A}'$-transition takes a clean configuration to an unclean one; hence, C' is also clean. □

Lemma 8 *Let $C \equiv C'$, where C is reachable in $\mathcal{A}$ and C' is reachable in $\mathcal{A}'$. Suppose $C' \overset{*}{\to} D'$ in $\mathcal{A}'$. Then $C \overset{*}{\to} D$ and $D \equiv D'$ for some $\mathcal{A}$-configuration D.*

Proof Proof is by induction on the length k of the execution $C' \overset{*}{\to} D'$.

Base case: If $k = 0$, it suffices to take $D = C$.

Inductive case: Suppose the lemma holds for $k - 1$. Let $C' = C_0' \to C_1' \to \cdots \to C_k' = D'$. By the induction hypothesis, there exists C_{k-1} such that $C \overset{*}{\to} C_{k-1}$ and $C_{k-1} \equiv C_{k-1}'$. If $C_{k-1}' \to C_k'$ is a transition in groups (a)–(d), then $C_{k-1}' \equiv C_k'$, so we choose $C_k = C_{k-1}$. If it belongs to group (e), then $\rho(C_{k-1}') \to \rho(C_k')$ is an $\mathcal{A}$-transition by construction. Let π be the agent permutation such that $C_{k-1} = \pi(\rho(C_{k-1}'))$. Define $C_k = \pi(\rho(C_k'))$. It is easily seen that $C \overset{*}{\to} C_{k-1} \to C_k$ and $C_k \equiv C_k'$. Hence, the lemma holds for k by choosing $D = C_k$

By induction, the claim holds for all k. □

Lemma 9 *Let C' be a reachable clean configuration of $\mathcal{A}'$. Let C be a reachable configuration of $\mathcal{A}$ such that $C \equiv C'$. Suppose $C \to D$ is a possible $\mathcal{A}$-transition. Then $C' \overset{*}{\to} D'$ and $D \equiv D'$ for some $\mathcal{A}'$-configuration D'.*

Proof Suppose $C \overset{(u,v)}{\to} D$ via encounter $e = (u, v)$. Suppose $C(u) = p$, $C(v) = q$, and $(p', q') = \delta(p, q)$. We proceed to construct D'.

Begin by fixing a spanning tree in the interaction graph of $\mathcal{P}'$. We restrict attention to encounters described by edges in the spanning tree. States p and q are the state components of two distinct nodes in C'. Similarly, batons S and R lie in two distinct nodes. We describe a sequence of transitions whose effect will be to move state p and baton R along spanning tree edges to some node u', and to similarly move state q and baton S to some node v', where u' and v' are the endpoints of some edge.

Using a sequence of group (c) transitions, move the S and R batons to distinct leaves of the spanning tree. Let u' be the leaf now containing baton S, and let v' be some adjacent node. Thus, (u', v') or (v', u') (or both) is an edge; choose one and call it e. Using a sequence of group (d) transitions, move state p to node u' and move state q to node v'. Using a sequence of group (c) transitions, move baton R to node v'. Finally, apply a group (e) transition to e to obtain D'.

We have thus constructed a sequence of configurations $C' = C_0' \equiv C_1' \equiv \cdots \equiv C_{k-1}' \overset{e}{\to} C_k' = D'$. It is easily seen that C is a Q-restriction of C_{k-1}' via some permutation π that maps u' to u and v' to v. Since D' is identical to C_{k-1}' except for the states of u' and v', and the simulated state components of u' and v' have been replaced by p' and q', respectively, it follows that D is a Q-restriction of D'. □

Lemma 10 *Let $C \equiv C'$, where C and C' are reachable configurations of $\mathcal{A}$ and $\mathcal{A}'$, respectively, and C' is final in $\mathcal{A}'$. Then C is final in $\mathcal{A}$.*

Proof Let $G(\mathcal{A}', \mathcal{P}')$ be the transition graph of $\mathcal{A}'$ and $\mathcal{P}'$, and let $\mathcal{S}'$ be the final strongly connected component of $G(\mathcal{A}', \mathcal{P}')$ that contains C'. Let $\mathcal{S}$ be the set of all reachable $\mathcal{A}$-configurations D such that $D \equiv D'$ for some $D' \in \mathcal{S}'$. Hence, $C \in \mathcal{S}$. We now show that $\mathcal{S}$ is a union of final strongly connected components of $G(\mathcal{A}, \mathcal{P}_n)$.

It suffices to show that if $C_1 \in \mathcal{S}$ and $C_1 \overset{*}{\to} C_2$, then $C_2 \in \mathcal{S}$ and $C_2 \overset{*}{\to} C_1$. By definition of $\mathcal{S}$, there exists $C_1' \in \mathcal{S}'$ such that $C_1' \equiv C_1$. By Lemma 7, since C_1' is final, then

C_1' is clean. By repeated application of Lemma 9, there exists $C_2' \equiv C_2$ such that $C_1' \xrightarrow{*} C_2'$. Since $\mathcal{S}'$ is final, then $C_2' \in \mathcal{S}'$ and $C_2' \xrightarrow{*} C_1'$. By Lemma 8, $C_2 \xrightarrow{*} \bar{C}_1$ and $\bar{C}_1 \equiv C_1'$ for some $\mathcal{A}$-configuration $\bar{C}_1$. If $\bar{C}_1 = C_1$, we are done. If not, we have established that $C_1 \equiv C_1' \equiv \bar{C}_1$ and $C_1 \xrightarrow{*} \bar{C}_1$. Hence $\bar{C}_1 = \pi(C_1)$ for some agent permutation π, so $C_1 \xrightarrow{*} \pi(C_1)$. From this, it follows that

$$\pi^k(C_1) \xrightarrow{*} \pi^k(\pi(C_1)),$$

where π^k is the kth iterate of π, that is, $\pi^k(C_1) = \pi(\pi(\ldots\pi(C_1)\ldots))$ k times. Hence,

$$C_1 \xrightarrow{*} \pi(C_1) \xrightarrow{*} \pi(\pi(C_1)) \xrightarrow{*} \cdots \xrightarrow{*} \pi^k(C_1),$$

For some k_0, π^{k_0} is the identity function, so in particular,

$$\bar{C}_1 = \pi(C_1) \xrightarrow{*} \pi^{k_0}(C_1) = C_1.$$

Hence, $C_2 \xrightarrow{*} C_1$, as desired. □

We now complete the proof of Theorem 7.

Proof We must show that every computation of $\mathcal{A}'$ on input x stabilizes to $\psi(x)$. Let Ξ' be a computation of $\mathcal{A}'$ on input x. Let C' occur infinitely often in Ξ'. By Lemma 1, C' is final. By Lemma 8, $C' \equiv C$ for some reachable configuration C of $\mathcal{A}$. By Lemma 10, C is final in $\mathcal{A}$. Let $y = y_C$ be the output determined by C. Since $\mathcal{A}$ computes a predicate, then y is the constant assignment **0** or **1**, and y is correct for ψ. The output determined by C' is some permutation of y, but since y is the constant function, all permutations of y are identical. Hence, the output determined by C' is y, which is correct.

We conclude that $\mathcal{A}'$ stably computes ψ. □

6 Computation with randomized interactions: Conjugating automata

"Stability" is probably not a strong enough guarantee for most practical situations, but it is the best we can offer given only the fairness condition. To make stronger guarantees, we must put some additional constraints on the interactions between members of the population.

Let us add a probabilistic assumption on how the next pair to interact is chosen. Many assumptions would be reasonable to study. We consider one of the simplest: the ordered pair to interact is chosen at random, independently and uniformly from all ordered pairs corresponding to edges in the interaction graph. When the interaction graph is complete, this is the model of *conjugating automata*, inspired by models introduced by Diamadi and Fischer to study the acquisition and propagation of knowledge about trustworthiness in populations of interacting agents [5].

Random pairing is sufficient to guarantee fairness with probability 1, so any protocol that stably computes a predicate g in a fair model computes g with probability 1 on every input in the corresponding random-pairing model, assuming both run on the same population.

However, probabilities also allow us to consider problems where we only compute the correct answer with high probability, or to describe the expected number of interactions until a protocol converges. Given a function f mapping $\mathcal{X}$ to $\mathcal{Y}$, a population protocol $\mathcal{A}$, and an input x, we define the probability that $\mathcal{A}$ computes f on input x to be the probability of all computations beginning with $I(x)$ that stabilize with output $f(x)$.

For example, for the (mod m) protocol, we can compute both the expected number of interactions in a computation until there is just one leader and the expected number of further interactions until every member of the population has interacted with the unique leader.

The time (meaning the number of interactions) to get a single leader is equal to the sum of the times until two leaders meet with $n, n-1, \ldots$ leaders; this is

$$\sum_{i=2}^{n} \frac{\binom{n}{2}}{\binom{i}{2}} = (n-1)^2.$$

Once there is a unique leader, it must participate in $\Theta(n \log n)$ interactions on average before it encounters every other member of the population (immediate application of the Coupon Collector Problem). But since the leader participates in only $(n-1)/\binom{n}{2} = 2/n$ of the interactions, this translates into a total of $\Theta(n^2 \log n)$ interactions in the full population.

Summing these two bounds, the expected total number of interactions until the output is correct is $\Theta(n^2 \log n)$. In general, we are interested in protocols that accomplish their tasks in an expected number of interactions polynomial in n, the population size.[5]

Generalizing this argument to the constructions of Lemma 5, Theorem 5, and Corollary 3, we obtain the following:

Theorem 8 *Let ψ be a Presburger definable predicate. Then there is a conjugating automaton (randomized population protocol) that computes ψ with probability 1, where the population converges to the correct answer in expected total number of interactions $O(k_\psi n^2 \log n)$, where k_ψ is a constant depending on ψ.*

Proof Observe that the construction used in Theorem 5 involves (a) electing a unique leader, followed by (b) computing in parallel zero or more base predicates of the form $\sum a_i x_i < c$ or $\sum a_i x_i \equiv c \pmod{m}$; and (c) combining the results of these base computations according to the formula and distributing the results to all agents.

We have already observed that step (a) takes $O(n^2)$ time. We will now show that step (b) takes $O(n^2 \log n)$ time. We have already shown that computing a single sum (mod m) takes $O(n^2 \log n)$ time, as the leader just needs to encounter each other agent once.

---

[5] Note that such protocols do not terminate with a final answer; they remain capable of resuming indefinitely.

For the threshold predicate, the situation is slightly more complicated; it is possible that some encounters between the leader and another agent will not make progress, because the leader is already "maxed out" and cannot collect any values from the other agent. Define n_- as the number of agents carrying negative values and n_+ as the number of agents carrying positive values. Then in any configuration with a unique leader,

1. If the leader's count is non-positive and the other agent's count is positive, then n_+ drops by one.
2. If the leader's count is non-negative and the other agent's count is negative, then n_- drops by one.

Now consider the length of the interval starting from some configuration until either n_- or n_+ drops. If the leader's count is positive, then n_- drops after an expected $O(n^2/n_-)$ interactions. If the leader's count is negative, then n_+ drops after an expected $O(n^2/n_+)$ interactions. In either case, there is at most one interval in which the leader's count has the appropriate sign for each distinct value of n_- or n_+, and its expected length is at most $O(n^2/n_-)$ or $O(n^2/n_+)$, depending again on the sign of the leader's count. Summing all such intervals for both n_- and n_+ gives a total expected time bounded by

$$\sum_{n_-=1}^{n-1} O(n^2/n_-) + \sum_{n_+=1}^{n-1} O(n^2/n_+)$$

$$= O(n^2 H_n) = O(n^2 \log n).$$

This establishes that a single instance of the threshold predicate can also be computed in $O(n^2 \log n)$ time.

To show that all the base predicates running in parallel take $O(n^2 \log n)$ time, let $T_i, i = 1, \ldots, k$ be the time for the i-th such predicate, where k is the (finite) number of such predicates. Then $\mathrm{E}[\max_i T_i] \leq \mathrm{E}[\sum_i T_i] = O(kn^2 \log n) = O(n^2 \log n)$.

Finally, step (c) requires that the leader encounter every other agent at least once, which we have already shown takes $O(n^2 \log n)$ time. Thus the total time for the Theorem 5 construction is $O(n^2 \log n)$. That the same asymptotic expected time bound applies to Corollary 3 follows from the fact that the proof of the corollary just constructs a new constant-size Presburger formula and applies Theorem 5 to it.   □

6.1 The benefits of a leader

Given a leader agent, it is possible to simulate a counter machine with a finite-state controller (whose state is stored in the leader) and increment, decrement, and zero-test operations, where the zero-test operation succeeds with high probability (Theorem 9). Using an initial leader election protocol and a standard reduction from Turing machines to counter machines due to Minsky [18], we can show that a conjugating automata can thus simulate logspace Turing machines on inputs given in unary (Theorem 10).

Simulating counters If we are allowed to designate a leader in the input configuration, that is, one agent that starts in a distinguished state, then the leader can organize the rest of the population to simulate a counter machine with $O(1)$ counters of capacity $O(n)$, with high probability. We assume throughout this section that the interaction graph is complete.

We use the representation described in Sect. 3.4 for integers in arithmetic computations. For a simulation of k counters in which counter i can take on a maximum value of $c_i n$, each state is mapped to a k-tuple of nonnegative integers in $[0, \ldots, c_1] \times \cdots \times [0, \ldots, c_k]$. The sum of component i over the population gives the current contents of counter i. We assume that the inputs to the counter machine are supplied in designated counters and the leader simulates the finite-state control of the counter machine.

To decrement counter i, the leader waits to encounter an agent with component i of its state greater than zero, and decrements it. Incrementing counter i is similar; component i must be less than its maximum value c_i. These operations will happen with probability 1, assuming that they are possible. However, testing counter i for zero is different; the leader must attempt to decide whether there are any agents with component i greater than zero. We give a method that is correct with high probability. It is the ability to make (possibly incorrect) decisions that enables effective sequencing and iteration of computations in this model.

The leader initially labels one other agent (the timer) with a special mark. The leader waits for one of two events: (1) an interaction with an agent with a nonzero component i, or (2) k consecutive interactions with the timer. If an event of type (1) occurs first, then the simulated counter is certainly not zero. Event (2) has low probability, so if it occurs first, the probability is high that the leader has encountered every other agent in the meantime, so the leader may conclude (with a small probability of error) that the value of simulated counter i is zero. The parameter k controls the probability of error, at the expense of increasing the expected number of interactions.

The probability that the leader prematurely concludes that there are no tokens of a particular type depends on the number of such tokens. We can model this game as an urn process, where at each step (corresponding to some interaction between the leader and one of the $n-1$ other tokens), a token is drawn from the urn, examined, and replaced. If the token is one of m counter tokens, the leader *wins*: it correctly determines that there is at least one counter token in the urn. If the token is an unmarked token or a timer token, the leader replaces it and continues to draw. The leader *loses* if it draws k timer tokens in a row without drawing any other token.

For simplicity, we write $N = n - 1$ for the size of the urn in this process. We also assume that the timer token is distinct from all the counter tokens, although later we will allow the agent carrying the timer token to also carry part of the counter value. If the timer token is also a counter token, then the probability of seeing the timer token before a counter token drops to zero, and the expected number of

steps until the first counter token is drawn when there are m counter tokens is exactly N/m.

However, in the case where the timer token and counter tokens are distinct, we have:

Lemma 11 *With an urn containing N tokens, of which m are counter tokens and 1 a timer token:*

1. *The probability of drawing the timer token k times in a row before drawing a counter token is exactly*

$$\frac{N-1}{mN^k + (N-1-m)} \le \frac{1}{mN^{k-1}}.$$

2. *Conditioned on not drawing the timer token k times in a row before drawing a counter token, and provided $m > 0$, the expected number of draws up to and including the first draw of a counter token is less than or equal to N/m.*
3. *When $m = 0$, the expected number of draws until the timer token is drawn k times in a row is $O(N^k)$.*

Proof We consider first the probability of losing, i.e., the probability that we draw k timer tokens in a row before drawing a counter token. At the start of the process, there is a probability of N^{-k} that we draw the timer token on every one of the first k draws. Call this event L. If L does not occur, then we draw the timer token between 0 and $k-1$ times, followed by some non-timer token x. Since all non-timer tokens are equally likely to be x, the probability that x is a counter token conditioned on L *not* occurring is $\frac{m}{N-1}$; in this case the process ends. If x is not a counter token, then the process starts over from the beginning.

Letting p be the probability of losing, we have

$$p = \Pr[L] + (1 - \Pr[L])\left(1 - \frac{m}{N-1}\right)p.$$

Solving this equation for p gives

$$p = \frac{\Pr[L]}{1 - (1 - \Pr[L])\left(1 - \frac{m}{N-1}\right)}$$

$$= \frac{N^{-k}}{1 - (1 - N^{-k})\left(\frac{N-1-m}{N-1}\right)}$$

$$= \frac{N^{-k}(N-1)}{(N-1) - (1 - N^{-k})(N-1-m)}$$

$$= \frac{N^{-k}(N-1)}{(N-1) - (N-1-m) + N^{-k}(N-1-m)}$$

$$= \frac{N^{-k}(N-1)}{m + N^{-k}(N-1-m)}$$

$$= \frac{N-1}{mN^k + (N-1-m)}.$$

For the upper bound, observe that

$$\frac{N-1}{mN^k + (N-1-m)} \le \frac{N}{mN^k} = \frac{1}{mN^{k-1}}.$$

For the second part, consider again the initial state. From this state we first draw the timer token zero or more times, followed by a non-timer token. The expected number of such draws until we get the first non-timer (without any conditioning) is $\frac{N}{N-1}$, and conditioning on not drawing the timer k times in a row can only reduce this value. Having drawn a non-timer, the probability that it is a counter token is again $\frac{m}{N-1}$; if it is not, we start over from the beginning.

Letting T be the expected number of draws, we have:

$$T \le \left(\frac{N}{N-1}\right) + \left(\frac{N-1-m}{N-1}\right)T.$$

Solving for T gives

$$T \le \frac{N/(N-1)}{1 - (N-1-m)/(N-1)}$$

$$= \frac{N}{(N-1) - (N-1-m)} = \frac{N}{m}.$$

For the third part, we again consider sampling from the urn without stopping, and start with 0 or more timer token draws, followed by a non-timer token draw. Each such phase includes an expected $\frac{N}{N-1}$ draws, and has probability N^{-k} of including k timer tokens. Stopping after k timer tokens can only reduce the time, so we have

$$T \le \left(\frac{N}{N-1}\right) + (1 - N^{-k})T,$$

from which $T \le N^k (\frac{N}{N-1}) = O(N^k)$. $\quad\square$

We now use Lemma 11 to bound the time and error of performing a *zero test* operation in a population protocol, where a unique leader wishes to determine if there are no nonzero counter tokens in the rest of the population. As in the urn process, the leader gives up if it sees the timer token (held by one of the other agents) in k consecutive interactions, without first seeing a nonzero counter value.

We again assume that the timer token sits on an agent with a zero counter value, and that there are m agents with nonzero counter values. To translate the time bounds of Lemma 11 into expected steps of the population process, we must not only substitute $n-1$ for N, but must also take into account the fact that when testing for zero, only a fraction $2/n$ of all interactions involve the leader. This gives an expected number of population protocol steps per draw of $\Theta(n)$, so that the time bounds for a zero test become $O(n^2/m)$ when $m > 0$ and $O(n^{k+1})$ when $m = 0$. We summarize these bounds as:

Theorem 9 *Given a standard population with n agents, of which one is a leader agent, one carries a timer token, and m carry counter tokens, and a zero test operation that waits for either (a) an encounter between the leader and a counter token, or (b) k encounters between the leader and timer tokens with no intervening encounter between the leader and any other token:*

1. *The probability that the zero test incorrectly reports zero when $m > 0$ is zero if the timer token is on the same agent as a counter token and $\Theta(n^{-k}/m)$ otherwise.*
2. *Conditioned on a correct outcome, the expected time to complete a zero test is $O(n^2/m)$ when $m > 0$ and $O(n^{k+1})$ when $m = 0$.*

How to elect a leader If we do not have a unique leader in the input configuration, it is possible to establish one using the ideas of the leader bit, as in the proof of Lemma 5, and the timer mark, as in the counter simulation above.

At the global start signal, every agent receives its input symbol (which it remembers for the duration of the computation), sets its leader bit equal to 1, and clears its timer mark (indicating that it is not a timer). Any agent whose leader bit equals 1 begins an initialization phase: it marks the first non-timer agent that it encounters as a timer and attempts to initialize every other agent. It uses the event of encountering a timer k times in a row to determine the end of the initialization phase.

Of course, at first every agent is attempting to run the initialization phase, so there will be general chaos. Whenever two agents with leader bit equal to 1 encounter each other, one (the loser) sets its leader bit to 0, and the other (the winner) keeps its leader bit 1. If the loser has already marked a timer, the winner waits until it encounters a timer and turns it back into a non-timer before proceeding. The winner then restarts the initialization phase (not creating another timer if it has already released one). When initialized, agents with leader bit equal to 0 revert to a state representing only their input and their leader bit, but they retain their timer status.

If an agent with leader bit equal to 1 completes the initialization phase, it begins the computation (e.g., simulating a counter machine, as in the preceding section). If during the computation it encounters another agent with leader bit equal to 1, the two proceed as indicated above, one setting its leader bit to 0, and the other restarting the initialization phase, with appropriate housekeeping to ensure retrieval of the extra timer, if any.

After a period of unrest lasting an expected $\Theta(n^2)$ interactions, there will be just one agent with leader bit equal to 1. After the interaction eliminating the last rival, this lucky winner will succeed in initializing all other agents with high probability (because there is only one timer in the population) and proceed with the computation as the unique leader. If and when the counter machine halts, the unique leader can propagate that fact (along with the output, if a function of one output is being computed) to all the other agents. If there have been no errors during the (final) simulation, the output of every configuration in the rest of the computation is correct.

Simulating a Turing machine We have just shown how to carry out zero tests and to elect a leader with high probability. We now show how to simulate a logspace Turing machine with high probability, using a standard reduction due to Minsky [18] from Turing machines to counter machines.

The central idea of Minsky's construction is to represent a Turing machine tape as two stacks, and then represent each stack as a counter value using a Gödel-numbering scheme where the sequence of symbols $x_0, x_1, \ldots, x_m$ is stored as

$$\sum_{i=0}^{m} x_i b^i,$$

where each symbol is assigned a positive numerical value and b is a constant base that exceeds the value of all the symbols. Pushing a new symbol x corresponds to setting $c \leftarrow cb + x$; a pop operation consists of setting $c \leftarrow \lfloor c/b \rfloor$ and returning the remainder. The product and quotient operations can each be implemented using a second counter that accumulates the new value while the first counter is decremented to zero; the remainder is accumulated in the finite-state controller (or in our simulation, the leader agent) during the quotient operation. A total of three counters—one for each side of the tape plus an extra accumulator—are used for the simulation.

We represent these counters using the integer-based input convention. Each agent other than the leader and the timer stores a vector of values in the range $0, \ldots, M$ for some M; the value of counter i is the sum of the i-th positions in these vectors, and may be as large as $(n-2)M$. A counter is zero if and only if every agent holds a zero share of the counter.

To multiply the value of counter i by b, storing the result in counter j (which is assumed to start at zero), the leader executes the following simple loop:

1. Test counter i for zero; if zero, exit the loop.
2. Decrement counter i.
3. Increment counter j b times.
4. Repeat from step 1.

The first step uses the zero-test protocol with waiting parameter k. When counter i has a nonzero value ℓ, the number of interactions to complete the zero test is $\Theta(n^2/m)$ and the probability of error is $O(n^{-k}/m)$, where $m \geq \lceil \ell/M \rceil$ is the number of agents with nonzero shares in the counter (Theorem 9). The second step can be combined with the zero test, since the first encounter between the leader and an agent with non-zero counter value i can also decrement the counter. The third step requires waiting for b encounters between the leader and agents with counter shares less than M; assuming there is always at least one such agent, this requires an expected $O(bn^2)$ interactions. Note that the second step does not add any probability of error: the timer token is not used to bound the time for this step, as the leader is guaranteed to eventually encounter a counter agent that is not full.

The last zero test has $\ell = 0$, and takes $O(n^{k+1})$ interactions, again by Theorem 9.

For an initial counter value bounded by nM, the total probability of error is

$$O\left(\sum_{\ell=1}^{nM} \frac{n^{-k}}{\lceil \ell/M \rceil}\right) = O\left(M^2 n^{-k} \sum_{h=1}^{n} \frac{1}{h}\right) = O(n^{-k} \log n),$$

and the total time is

$$O\left(\sum_{\ell=1}^{nM}\left(\frac{n^2}{\lceil \ell/M \rceil}\right) + bn^2\right) + O(n^{k+1})$$

$$= O(n^2 \log n + n^{k+1}).$$

For a push operation, the additional $O(xn^2)$ expected interactions needed to add in x is dominated by the time for the product even when k is small.

For the quotient operation, the analysis is essentially the same, the only difference being that counter j is only incremented once for every b passes through the loop instead of b times per pass. So again the probability of error for a single quotient operation is $O(n^{-k} \log n)$ and the expected number of interactions $O(n^2 \log n + n^{k+1})$.

Finally, the same bounds apply for the same reasons to the initialization step where the unique surviving leader initializes all the other agents; again, we are simply waiting for the leader to encounter all the non-timer agents before seeing the timer k times in a row.

We now have all the pieces we need to show the simulation result.

Theorem 10 *Let $f(x)$ be a function in logspace, where the input x is represented in unary. Let $T(n) = O(n^d)$, where d is an integer, be the worst-case running time of some logspace Turing machine that computes f. Then for any fixed integer $c > 0$, there is a conjugating automaton that, when run in the standard population with n members, computes $f(x)$ for any $x \le n$ with probability of error $O(n^{-c} \log n)$ in expected time $O(n^{d+2} \log n + n^{2d+c+1})$.*

Proof Let $k = c + d$, where k is the waiting parameter of the zero test operation.

Simulating one step of the Turing machine involves $O(1)$ product and quotient operations, each of which contributes $O(n^{-k} \log n)$ to the error probability. The total probability of error is then

$$T(n)O(n^{-k} \log n) = O(n^d n^{-(c+d)} \log n) = O(n^{-c} \log n).$$

The expected running time for the simulation, including the initial leader election phase, is

$$O(n^2) + T(n)O(n^2 \log n + n^{k+1})$$

$$= O(n^{d+2} \log n + n^{2d+c+1}). \qquad \square$$

6.2 Simulating conjugating automata

In this section, we show that either deterministic polynomial time or randomized logarithmic space (with exponential time) is sufficient to recognize predicates computable with probability at least $1/2 + \epsilon$ by conjugating automata.

Suppose that a conjugating automaton $\mathcal{A}$ computes a predicate F with probability at least $1/2 + \epsilon$. Then F can be computed by a polynomial-time Turing machine. As before,

we assume that a string x of symbols from X represents an input assignment x to $\mathcal{A}$, so that n represents both the input length and the population size.

On input x, a polynomial-time Turing machine can construct the matrix representing the Markov chain whose states are the multiset representations of the population configurations reachable from $I(x)$, since there are at most $n^{|Q|}$ of them. Solving for the stationary distribution of the states, the Turing machine can determine a set of configurations of probability greater than $1/2$. that all have the same output (which must be correct, as an incorrect output can only appear with probability less than $1/2 - \epsilon$). The Turing machine then writes this common output to its output tape and halts.

Under the same conditions, F can be computed by a randomized Turing machine with probability $1/2 + \epsilon'$ using space $O(\log n)$. A randomized Turing machine simulates the automaton by using a finite number of $O(\log n)$-bit counters to keep track of the number of members of the population in each state. Using coin flips, it simulates drawing a random pair of population members and updating the counters according to the transition function of $\mathcal{A}$. By running the simulation for long enough, the randomized Turing machine can be almost certain of being in a terminal strongly connected component of the states of the Markov chain, at which point the Turing machine halts and writes the output of the current configuration on its output tape.

How long is this? The number of distinct simulated configurations is less than $(n+1)^{|Q|}$, so the diameter of the state space of the Markov chain is less than $d = (n+1)^{|Q|}$. Given any state that is not in a terminal component, there is some path of length at most d that leads to a state that is. It follows that in each interval of d simulated transitions, there is a probability of at least $(n(n-1))^{-d} > n^{-2d}$ of reaching a terminal component. So the probability of *not* reaching a terminal component after Kd simulated transitions is less than

$$(1 - n^{-2d})^K \le \exp(K/n^{2d}).$$

It follows that we can achieve any constant probability $1 - \delta$ of convergence after

$$O(dn^{2d}) = O\big((n+1)^{|Q|}n^{2(n+1)^{|Q|}}\big)$$

$$= O\big(2^{(|Q|+2(n+1)^{|Q|})\lg n}\big) = O\big(2^{n^{2|Q|}}\big)$$

simulated transitions.

To wait this long, the randomized Turing machine allocates a counter of $c\lceil \log n \rceil$ bits and flips a coin before each simulated interaction, adding 1 to the counter on heads, and clearing the counter on tails. The simulation is stopped when the counter overflows, that is, when there have been at least n^c consecutive heads. The probability that this event occurs starting at any particular time is 2^{-n^c}; it follows that during the first t trials the expected number of times that it occurs (and thus the probability that it occurs at least once) is at most $t2^{-n^c}$. Thus we expect to finish in time t with probability $o(1)$ provided $t = o(2^{n^c})$. Setting $t = 2^{n^{2|Q|}}$ and $c = 3|Q|$ thus gives an $o(1)$ probability of failing to

converge before the simulation stops. It follows that the randomized logspace simulation produces a correct answer with probability at least $1/2 + \epsilon - \delta - o(1) = 1/2 + \epsilon'$ for sufficiently large n.

We have just shown:

Theorem 11 *The set of predicates accepted by a conjugating automaton with probability $1/2 + \epsilon$ is contained in P. Further, they can be computed by a randomized logspace machine in exponential time.*

7 Other related work

In a Petri net, a finite collection of tokens may occupy one of a finite set of places, and transition rules specify how the tokens may move from place to place.[6] Viewing the states of a population protocol as places and the population members as tokens, our models can also be interpreted as particular kinds of Petri nets. Randomized Petri nets were introduced by Volzer [23] using a transition rule that does not depend on the number of tokens in each input place, in contrast to conjugating automata where the probability of an interaction between a particular state pair increases with the number of agents possessing those two states.

The Chemical Abstract Machine of Berry and Boudol [3] is an abstract machine designed to model a situation in which components move about a system and communicate when they come into contact, based on a metaphor of molecules in a solution governed by reaction rules. A concept of enforced locality using membranes to confine subsolutions allows the machines to implement classical process calculi or concurrent generalizations of the lambda calculus.

Ibarra, Dang, and Egecioglu [12] consider a related model of catalytic P systems. They show that purely catalytic systems with one catalyst define precisely the semilinear sets, and also explore other models equivalent in power to vector addition systems. The relationships between these models and ours is an intriguing topic.

Brand and Zafiropulo [4] define a model of communicating processes consisting of a collection of finite state machines that can communicate via pre-defined FIFO message queues. They focus on general properties of protocols defined in the model, such as the possibility of deadlock or loss of synchronization.

Milner's bigraphical reactive systems [17] address the issues of modeling locality and connectivity of agents by two distinct graph structures. In this work the primary focus is upon the expressiveness of the models, whereas we consider issues of computational power and resource usage.

8 Discussion and open problems

In addition to the open problem of characterizing the power of stable computation, many other intriguing questions and

directions are suggested by this work. One direction we have explored [2] is to define a novel storage device, the *urn*, which contains a multiset of tokens from a finite alphabet. It functions as auxiliary storage for a finite control with input and output tapes, analogous to the pushdown or work tape of traditional models. Access to the tokens in the urn is by uniform random sampling, making it similar to the model of conjugating automata.

We have primarily considered the case of a complete interaction graph, which we have shown in Theorem 7 provides the least computational power of all weakly-connected interaction graphs in the stable computation model. The question of characterizing the power of stable computations on particular restricted interaction graphs remains open. We can also consider the interaction graph itself as part of the input and ask what interesting properties of its underlying graph can be stably computed by a population protocol. This problem may have applications in analyzing the structure of deployed sensor networks. Some initial work in this direction has been carried out in [1].

An interesting restriction of our model is to consider only *one-way communication* between the two agents in an interaction, that is, the transition function δ can be restricted to change only the state of the responder in the interaction, keeping the state of the initiator the same. Although there are still protocols to decide whether the number of 1's in the input is at least k, this condition appears to restrict the class of stably computable predicates severely.

The models in this paper assume a "snapshot" of the inputs is taken when the global start signal is received. A model accommodating streaming inputs, as is typically assumed in sensor networks, would be very interesting.

We have assumed uniform sampling of pairs to interact, but for some applications it may make sense to consider other sampling rules. One idea is weighted sampling, in which population members are sampled according to their weights, possibly depending on their current states. We conjecture that with reasonable restrictions on the weights, weighted sampling yields the same power as uniform sampling. Other sampling rules might be based on more accurate models of patterns of interaction in populations of interest.

The interaction rules we consider are deterministic and specify pairwise interactions. What happens if the rules are nondeterministic, or specify interactions of larger groups, or allow the interaction to increase or decrease the population?

Our bound on the number of interactions in Theorem 8 applies only to stable computations of Presburger-definable predicates. The bounds on the simulation results in the Turing-machine simulation in Theorem 10 are higher, but still polynomial (for polynomial error bounds). It is not clear whether there are *any* useful computations of a conjugating automaton that require more than polynomial time; just as logspace machines do not have enough states to exploit superpolynomial time bounds, it may be that the lack of structure in a conjugating automaton's memory means that increasing its time bound adds no actual power.

---

[6] See [6, 7] for surveys of Petri nets.

Furthermore, we give bounds on the expected total number of interactions, but other resource measures may be more appropriate in some applications. For many applications, interactions happen in parallel, so that the total number of interactions may not be well correlated with wall-clock time; defining a useful notion of time is a challenge. Alternatively, if we consider only the number of interactions in which at least one state changes (which might be correlated with the energy required by the computation), then the bounds can be finite even in the stable computation model, and the expected bounds can be smaller in the conjugating automata model.

Finally, we have not addressed the issue of fault tolerance, which is of course of immense practical importance in real sensor networks. In one sense, our underlying model should be very robust in the face of faults since we are making such weak assumptions about when interactions occur. If an agent dies, say from an exhausted battery, the interactions between the remaining agents are unaffected. Of course, many of the algorithms we describe here would not survive the failure of a single agent, especially those based on leader election. It is a challenging open problem to design fault-tolerant algorithms for some of the problems addressed here, or show that fault-tolerant solutions do not exist.

References

1. Angluin, D., Aspnes, J., Chan, M., Fischer, M.J., Jiang, H., Peralta, R.: Stably computable properties of network graphs. In: Viktor K. Prasanna, Sitharama Iyengar, Paul Spirakis and Matt Welsh (eds.), Distributed Computing in Sensor Systems: First IEEE International Conference (2005). Lecture Notes in Computer Science **3560**, 63–74 (June/July, 2005) Proceedings Marina del Rey, CA, USA
2. Angluin, D., Aspnes, J., Diamadi, Z., Fischer, M.J., Peralta, R.: Urn automata. Tech. Rep. YALEU/DCS/TR–1280, Yale University Department of Computer Science (2003)
3. Berry, G., Boudol, G.: The chemical abstract machine. Theor. Comp. Sci. **96**, 217–248 (1992)
4. Brand, D., Zafiropulo, P.: On communicating finite-state machines. J. ACM **30**(2), 323–342 (1983)
5. Diamadi, Z., Fischer, M.J.: A simple game for the study of trust in distributed systems. Wuhan Univ. J. Natur. Sci. **6**(1–2), 72–82 (2001). Also appears as Yale Technical Report TR–1207, January 2001, available at URL ftp://ftp.cs.yale.edu/pub/TR/tr1207.ps
6. Esparza, J.: Decidability and complexity of Petri net problems-an introduction. In: Rozenberg, G., Reisig, W., (eds.) Lectures on Petri Nets I: Basic models, pp. 374–428. Springer Verlag (1998). Published as LNCS 1491
7. Esparza, J., Nielsen, M.: Decibility issues for Petri nets—a survey. J. Inform. Process. Cybern. **30**(3), 143–160 (1994)
8. Fang, Q., Zhao, F., Guibas, L.: Lightweight sensing and communication protocols for target enumeration and aggregation. In: Proceedings of the 4th ACM International Symposium on Mobile ad hoc Networking & Computing, pp. 165–176. ACM Press (2003)
9. Fischer, M.J., Rabin, M.O.: Super-exponential complexity of Presburger arithmetic. In: Complexity of Computation, SIAM-AMS Proceedings, vol. VII, pp. 27–41. American Mathematical Society (1974)
10. Ginsburg, S., Spanier, E.H.: Semigroups, Presburger formulas, and languages. Pac. J. Math. **16**, 285–296 (1966)
11. Grossglauser, M., Tse, D.N.C.: Mobility increases the capacity of ad hoc wireless networks. IEEE/ACM Transac. Networking **10**(4), 477–486 (2002)
12. Ibarra, O.H., Dang, Z., Egecioglu, O.: Catalytic p systems, semilinear sets, and vector addition systems. Theor. Comput. Sci. **312**(2–3), 379–399 (2004)
13. Immerman, N.: Nondeterministic space is closed under complementation. SIAM J. Comput. **17**(5), 935–938 (1988)
14. Intanagonwiwat, C., Govindan, R., Estrin, D.: Directed diffusion: a scalable and robust communication paradigm for sensor networks. In: Proceedings of the 6th Annual International Conference on Mobile computing and networking, pp. 56–67. ACM Press (2000)
15. Kracht, M.: The Mathematics of Language, Studies in Generative Grammar, vol. 63. Mouton de Gruyter (2003). ISBN 3-11-017620-3
16. Madden, S.R., Franklin, M.J., Hellerstein, J.M., Hong, W.: TAG: A Tiny AGgregation service for ad-hoc sensor networks (December, 2002). In OSDI 2002: Fifth Symposium on Operating Systems Design and Implementation
17. Milner, R.: Bigraphical reactive systems: basic theory. Tech. rep., University of Cambridge (2001). UCAM-CL-TR-523
18. Minsky, M.L.: Computation: Finite and Infinite Machines. Prentice-Hall Series in Automatic Computation. Prentice-Hall, Inc., Englewood Cliffs, N.J. (1967)
19. Monk, J.D.: Mathematical Logic. Springer, Berlin, Heidelberg (1976)
20. von Neumann, J.: Theory and organization of complicated automata. In: A.W. Burks (ed.) Theory of Self-Reproducing Automata [by] John von Neumann, pp. 29–87 (Part One). University of Illinois Press, Urbana (1949). Based on transcripts of lectures delivered at the University of Illinois, in December 1949. Edited for publication by A.W. Burks
21. Parikh, R.J.: On context-free languages. J. ACM **13**(4), 570–581 (1966). DOI http://doi.acm.org/10.1145/321356.321364
22. Presburger, M.: Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. In: Comptes-Rendus du I Congrès de Mathématiciens des Pays Slaves, pp. 92–101. Warszawa (1929)
23. Volzer, H.: Randomized non-sequential processes. In: Proceedings of CONCUR 2001-Concurrency Theory, pp. 184–201 (2001)
24. Zhao, F., Liu, J., Liu, J., Guibas, L., Reich, J.: Collaborative signal and information processing: An information directed approach. Proc. IEEE **91**(8), 1199–1209 (2003)