Leaderless Population Protocols Decide Double-exponential Thresholds

Anonymous author(s)

04.04.2022

Population protocols are a model of distributed computation in which finite-state agents interact randomly in pairs. A protocol decides for any initial configuration whether it satisfies a fixed property, specified as a predicate on the set of configurations. The state complexity of a predicate is the smallest number of states of any population protocol deciding that predicate. For threshold predicates $\varphi(x) \Leftrightarrow x \geq k$, with k constant, it is known to be in $\Omega(\log\log k) \cap \mathcal{O}(\log k)$. We close this remaining gap by showing that it is $\Theta(\log\log k)$, i.e. we construct protocols with $\mathcal{O}(n)$ states deciding $x \geq k$ with $k \geq 2^{2^n}$. This matches the known bound if the model is extended with leaders. Moreover, our construction is the first that is not 1-aware, making it robust against noisy initialisation.

1. Introduction

Population protocols are a distributed model of computation where a large number of indistinguishable finite-state agents interact randomly in pairs. The goal of the computation is to decide whether an initial configuration satisfies a given property. The model was introduced in 2004 by Angluin et al. [4, 5] to model mobile sensor networks with limited computational capabilities (see e.g. [27, 21]). It is also closely related to the model of chemical reaction networks, in which agents, representing discrete molecules, interact stochastically [17].

A protocol is a finite set of transition rules according to which agents interact, but it can be executed on an infinite family of initial configurations. Agents decide collectively whether the initial configuration fulfils some (global) property. This is done by *stable consensus*; each agent holds an opinion about the output and may freely change it, but eventually all agents agree.

An example of a property decidable by population protocols is *majority*: initially all agents are in one of two states, x and y, and they try to decide whether x has at least as many agents as y. This property may be expressed by the predicate $\varphi(x, y) \Leftrightarrow x \geq y$.

In a seminal paper, Angluin et al. [7] proved that the predicates that can be decided by population protocols correspond precisely to the properties expressible in Presburger arithmetic, the first-order theory of addition.

Time complexity. To execute a population protocol, the scheduler picks two agents uniformly at random and executes a pairwise transition on these agents. These two agents interact and may change states. The number of agents does not change during the computation, it will be denoted m throughout this paper.

The scheduler repeats this process until a stable consensus has emerged. By counting the expected number of transitions for inputs of a certain size, one can define the *time complexity* of a protocol.[1] For example, there exist protocols deciding the majority predicate from above in $\mathcal{O}(m^2)$ interactions [6, 19].

In the area of time complexity, several strong results have been obtained. A general construction is given in [5], which can decide *any* predicate expressible in Presburger arithmetic within $\mathcal{O}(m^2 \log m)$ interactions. Population protocols are often extended with a *leader* — an auxiliary agent not part of the input, which can assist the computation. It is known that this does not increase the expressive power of the model, i.e. it can still decide precisely the predicates expressible in Presburger Arithmetic. However, [6] gives a general construction that runs within $\mathcal{O}(m \operatorname{polylog} m)$ interactions. Together with the $\Omega(m^2/\operatorname{polylog} m)$ lower bound for leaderless protocols shown in [1] it follows that leaders enable protocols to be faster.

Space complexity. However, even without allowing the state space to grow, many of the above constructions need a large number of states. We estimate, for example, that the protocols of [6] needs tens of thousands of states. This is a major obstacle to implementing these protocols in chemical reactions, as every state corresponds to a chemical compound. If equipped with digital storage, the number of bits an agent needs to store its state corresponds to the logarithm of the number of states — not as prohibitive, but still something that is desirable to minimise.

We thus consider the *space-complexity*, the minimal number of states necessary for a population protocol solving a given problem. There are two major lines of inquiry into this direction:

- The first considers a family of protocols for a fixed predicate (usually the majority predicate), where each protocol is specialised for a population size m.
- The second also considers a family of protocols, but each decides a different predicate (for *all* population sizes m).

Growing state-space. In the original model (which is also the model of this paper), the set of states is fixed, and the same protocol can be used for an arbitrary number of agents. Relaxing this requirement has opened up a fruitful line of research; here, the number of states depends on m (e.g. $\mathcal{O}(\log m)$ states, or even $\mathcal{O}(\log \log m)$ states). In this model, faster protocols can be achieved [3, 25, 26].

[1] More commonly, one considers *parallel time*, the number of transitions divided by the number of agents.

It has also led to the development of space-efficient, fast protocols, which stabilise within $\mathcal{O}(m \operatorname{polylog} m)$ interactions (i.e. polylogarithmic parallel time), using a state-space that grows only slowly with the number of agents, e.g. $\mathcal{O}(\operatorname{polylog} m)$ states [1, 12, 2, 10, 9, 11, 20]. These protocols have focused on the majority predicate.

Moreover, lower bounds and results on time-space tradeoffs have been developed in this model [1, 2].

Predicate size. The above results focus on giving fast protocols for specific, fixed tasks. However, there is a second factor influencing the size of protocols: the predicate that the protocol decides. This is perhaps clearest in the case of *flock-of-bird* predicates $\varphi_k(x) \Leftrightarrow x \geq k$, for $k \in \mathbb{N}$. Intuitively, a protocol for $x \geq k$ decides whether the number of agents participating in the computation is at least k. Clearly, the number of states necessary to decide φ_k increases with k.

A simple protocol (close to the construction of [5]) for φ_k uses states $\{0, \ldots, k\}$ and transitions

$$i, j \mapsto i + j, 0 \quad \text{for } i + j < k \qquad\qquad i, j \mapsto k, k \quad \text{for } i + j \geq k$$

In the context of chemical reactions, this protocols is clearly insufficient. Each agent is a molecule, so $k \approx 10^{23}$, but one cannot synthesise 10^{23} different compounds.

Flock-of-bird predicates are a natural choice for investigating space complexity and prior research has analysed the construction of succinct protocols for these predicates. This question is also of theoretical interest, as it relates to the expressiveness of population protocols (cf the Busy Beaver function) and is linked to fundamental problems in the theory of Petri nets.

The original construction given above was improved by Blondin, Esparza and Jaax [14] to $\Theta(\log k)$ states. Perhaps surprisingly, they show that extending the model with a leader enables $\Theta(\log \log k)$ states. A constant-factor improvement for the former bound was later given in [22]. We remark that the $\Theta(\log \log k)$ result is not a construction for arbitrary flock-of-bird predicates, it only produces protocols for some infinite family of thresholds.

Very recently a number of lower bounds have been shown as well. Czerner and Esparza [18] proved that every ordinary population protocol for $x \geq k$ must use $\Omega(\log \log \log k)$ states, while a protocol with leaders needs $\Omega(\operatorname{ack}^{-1} k)$ states, where ack is an Ackermann-type function. A (yet unpublished) result by Czerner, Esparza and Leroux improves the bound in the leaderless case to $\Omega(\log \log k)$. The first elementary bound for the case with leaders was given by Leroux [23], showing that a $\Omega(\log \log k)$ bound holds here as well. At this point, the bounds are tight for the case with leaders, while an exponential gap remains for the leaderless case. These results are summarised in Table 1.

Finally, there have also been results on the space complexity of arbitrary predicates. Blondin et al. [13] give a general construction, and thus extend (in some sense) the $\Theta(\log k)$ bound to the whole of Presburger arithmetic. This result was very recently improved by Czerner et al. [19], giving a construction that is both succinct and fast, i.e. runs within $\mathcal{O}(m^2)$ interactions.

Table 1: Prior results on the state complexity of flock-of-bird predicates $\varphi(x) \Leftrightarrow x \geq k$, for $k \in \mathbb{N}$. Upper bounds need only hold for infinitely many k.

year	result	type	ordinary	with leaders
2018	Blondin, Esparza, Jaax [14]	construction	$\mathcal{O}(\log k)$	$\mathcal{O}(\log \log k)$
2021	Czerner, Esparza [18]	impossibility	$\Omega(\log \log \log k)$	$\Omega(\text{ack}^{-1} k)$
2021	Czerner, Esparza, Leroux	impossibility	$\Omega(\log \log k)$	
2022	Leroux [23]	impossibility		$\Omega(\log \log k)$
2022	this paper	construction	$\mathcal{O}(\log \log k)$	

2. Main result

We construct population protocols (without leaders) for an infinite family of flock-of-bird predicates $x \geq k$, proving an $\mathcal{O}(\log \log k)$ upper bound on their state complexity. This closes the last remaining gap.

As for previous results in this area, our result is not a construction for arbitrary thresholds k, only for an infinite family of them. Indeed, a $\mathcal{O}(\log \log k)$ bound for the former can be shown impossible by a simple counting argument [14]. Our result, therefore, is easier to formally state by fixing the number of states n and specifying the largest threshold k that can be decided by a protocol with n states.

Theorem 1. *For every $n \in \mathbb{N}$ there is a population protocol with $\mathcal{O}(n)$ states deciding the predicate $\varphi(x) \Leftrightarrow x \geq k$ for some $k \geq 2^{2^n}$.*

The result is surprising, as the prevailing opinion tended towards the existing constructions being optimal.[2] It also shows that leaders do not provide an advantage in terms of space-complexity (as opposed to time-complexity, where they are known to be faster).

Our construction also evades a conditional impossibility result by [14], which gives a $\Omega(\log k)$ lower bound for leaderless 1-aware protocols. (Essentially, protocols where some agent knows at some point that the threshold has been exceeded.) Our protocol only accepts provisionally and continues to check that no invariant has been violated, making it (to the extend of our knowledge) the first construction that is not 1-aware.

As a corollary, this makes our construction the first to be robust against certain types of noise, in particular the addition of agents. This is related to research on self-stabilising population protocols [8, 16, 15] – our protocols are not quite self-stabilising, however. We discuss this in Appendix A.

Overview. We build on the technique of Lipton [24]. Implementing this technique requires the use of recursive procedure calls; our first contribution are *population programs*,

[2] In particular, the technique to show the $\Omega(\log \log k)$ lower bound could also be used for a $\Omega(\log k)$ bound. The $\Omega(\log \log k)$ bound is due to a reliance on Rackoff's theorem, a general result for Petri nets. It was hoped that the special structure of population protocols could be exploited instead.

a model in which population protocols can be constructed by writing structured programs, in Section 4.

Implementing the technique of [24] in our model, however, is not straightforward. We are only able to convert population programs to population protocols by relaxing requirements on the input representation. In fact, in our model the initial values of *all* registers are chosen by an adversary. The construction, therefore, must be robust against undesirable inputs. Our second contribution therefore is extending the original technique by adding error-checking routines. These provide some weak invariants, and we take care to ensure that our subroutines fail gracefully under error conditions that we cannot exclude. A high level overview of both the original technique as well as our error-checking strategy is given in Section 5. We give a detailed description with proofs of correctness in Section 6.

To get population protocols, we need to convert our population programs. We split this into two parts. First, we lower population programs to an assembly-like programming language using standard techniques; we refer to these as population machines. In a second step we construct population protocols to simulate arbitrary population machines. To implement the control state, we need a unique agent that can coordinate the computation. Of course, we do not have a leader; instead, we perform a leader election. This does not work perfectly, leading to the weak guarantees provided by population programs. This conversion is described in Section 7.

To start out, Section 3 introduces the necessary mathematical notation and formally defines population protocols as well as the notion of stable computation.

3. Preliminaries

Multisets. We assume $0 \in \mathbb{N}$. For a finite set Q we write $\mathbb{N}^Q$ to denote the set of multisets containing elements in Q. For such a multiset $C \in \mathbb{N}^Q$, we use $|C| := \sum_{q \in Q} C(q)$ to denote the total number of elements in C. Given two multisets $C, C' \in \mathbb{N}^Q$ we write $C \leq C'$ if $C(q) \leq C'(q)$ for all $q \in Q$, and we write $C + C'$ and $C - C'$ for the componentwise sum and difference (the latter only if $C \geq C'$). Abusing notation slightly, we use an element $q \in Q$ to represent the multiset C containing exactly q, i.e. $C(q) = 1$ and $C(r) = 0$ for $r \neq q$.

Population protocols. A *population protocol* is a tuple $PP = (Q, \delta, I, O)$, where

- Q is a finite set of *states*,
- $\delta \subseteq Q^4$ is a set of *transitions*,
- $I \subseteq Q$ is a set of *input states*, and
- $O \subseteq Q$ is a set of *accepting states.*

We write transitions as $(q, r \mapsto q', r')$, for $q, r, q', r' \in Q$. A *configuration* of PP is a multiset $C \in \mathbb{N}^Q$ with $|C| > 0$. A configuration C is *initial* if $C(q) = 0$ for $q \notin I$ (one might also say $C \in \mathbb{N}^I$ instead). It has output `true` if $C(q) = 0$ for $q \notin O$, and output

```
procedure MAIN                procedure TEST(i)               procedure CLEAN
  OF := false                   for j = 1, ..., i do            if maybe z > 0 then
  while ¬TEST(4) do               if maybe x > 0 then             restart
    CLEAN                           x ↦ y                       swap x, y
  OF := true                      else                          while maybe y > 0 do
  while ¬TEST(7) do                 return false                  y ↦ x
    CLEAN                       return true
  OF := false
  while true do
    CLEAN
```

Figure 1: A population program for $\varphi(x) \Leftrightarrow 4 \leq x \leq 7$ using registers x, y, z.

false if $C(q) = 0$ for $q \in O$. For two configurations C, C' we write $C \to C'$ if $C = C'$ or if there is a transition $(q, r \mapsto q', r') \in \delta$ s.t. $C \geq q + r$ and $C' = C - q - r + q' + r'$.

Stable computation. We are going to give a general definition of stable computation, so that we can later reuse it for population programs and population machines. Let $\mathcal{C}$ denote a set of configurations and $\to$ a left-total binary relation on $\mathcal{C}$ (i.e. for every $C \in \mathcal{C}$ there is a $C' \in \mathcal{C}$ with $C \to C'$). Further, we assume some notion of output, i.e. some configurations have an output $b \in \{\text{true}, \text{false}\}$ (but not necessarily all).

A sequence $\tau = (C_i)_{i \in \mathbb{N}}$ with $C_i \in \mathcal{C}$ is a *run* if $C_i \to C_{i+1}$ for all $i \in \mathbb{N}$. We say that τ *stabilises to* b, for $b \in \{\text{true}, \text{false}\}$, if there is an i s.t. C_j has output b for every $j \geq i$. A run τ is *fair* if every $C \in \mathcal{C}$ where $C_i \to C$ holds for infinitely many i appears infinitely often in C_i.

Turning back to population protocols, let $\varphi : \mathbb{N}^I \to \{\text{true}, \text{false}\}$ denote a predicate. We say that *PP decides* φ, if every fair run starting at an initial configuration $C \in \mathbb{N}^I$ stabilises to $\varphi(C)$.

As is usual, we use the notion of fairness instead of defining a stochastic process, as the former is easier to reason about. Additionally, if one considers only the predicate decided by a protocol (and not, say, its time complexity) the two coincide.

4. Population Programs

We introduce the model of population programs, which allows us to specify population protocols using structured programs. An example is shown in Figure 1; it is explained at the end of the section.

Formally, we say that a *population program* is a tuple $\mathcal{P} = (Q, \text{Proc})$, where Q is a finite set of *registers* and Proc is a list of *procedures*. Each procedure has a name and consists of (possibly nested) while-loops, if-statements and instructions. These are described in detail below.

Primitives. Each register $x \in Q$ can take values in $\mathbb{N}$. However, only three operations on these registers are supported.

The move instruction $(x \mapsto y)$, for $x, y \in Q$, decreases the value of x by one, and increases the value of y by one. We also say that it moves one unit from x to y. If x is empty, i.e. its value is zero, the programs hangs and makes no further progress.

The second operation is the nondeterministic nonzero check (**maybe** $x > 0$), for $x \in Q$. Briefly, it nondeterministically returns either false or whether $x > 0$. In other words, if it does return true, it certifies that x is nonzero. If it returns false, however, no information has been gained. We consider only fair runs, so if x is indeed nonzero the check must return true eventually if called infinitely often.

Finally, we allow for swaps between registers, i.e. (**swap** x, y) for $x, y \in Q$. A swap exchanges the values of the two registers. This primitive is not necessary, but it simplifies the implementation.

Loops and branches. Both while-loops and if-statements function as one would expect. We allow for simple boolean formulae in their arguments, but limit ourselves to at most a single binary operator (with any number of negations). This ensures that a single instruction can encode only a constant amount of information.

We also use for-loops. These, however, are just a macro and expand into multiple copies of their body. For example, in the program in Figure 1 the for-loop in TEST expands into i copies of the contained if-statement.

Procedures. Our model allows for a very limited kind of recursion. Procedures have no arguments, but we may have parametrised copies of a procedure. The program in Figure 1, for example, has four procedures: MAIN, CLEAN, TEST(4), and TEST(7).

Procedure calls must be acyclic. It is thus not possible for a procedure to call itself, and the size of the call stack remains bounded. We remark that it is possible to inline every procedure call (apart from the small detail that our model has no consideration for storing intermediate boolean values). The main reason to make use of recursion at all is succinctness: if our program contains too many instructions, the resulting population protocol has too many states.

We do allow procedures to return a single boolean value, and procedure calls can be used as expressions in conditions of while- or if-statements.

Output flag. There is an output flag, which can be modified only via the instructions $OF :=$ true and $OF :=$ false. (These are special instructions; it is not possible to assign values to registers.) The output flag determines the output of the computation.

Size. The *size* of $\mathcal{P}$ is defined as $|Q| + L + S$, where L is the number of instructions and S is the *swap-size*. The latter is defined as the number of pairs $(x, y) \in Q^2$ for which it is syntactically possible for x to swap with y via any sequence of swaps. Unfortunately, without restrictions we would convert swaps to population protocols with a quadratic blow-up in states, so we introduce this technical notion to quantify the overhead.

Initialisation and restarts. The only guarantee on the initial configuration is that execution starts at MAIN. In particular, all registers may have arbitrary values.

There is one final kind of instruction: **restart**. As the name suggests, it restarts the computation. It does so by nondeterministically picking any initial configuration s.t. the sum of all registers does not change.

Configurations and Computation. A *configuration* of $\mathcal{P}$ is a tuple $D = (C, OF, \sigma)$, where $C \in \mathbb{N}^Q$ is the *register configuration*, $OF \in \{\mathsf{true}, \mathsf{false}\}$ is the value of the output flag, and σ is the stack of called procedures. Such a configuration is *initial* if $\sigma = (\textsc{Main})$ and it has output OF. For two configurations D, D' we write $D \to D'$ if D can move to D' after executing one instruction.

Using the notions of stable computation defined in Section 3, we say that $\mathcal{P}$ *decides* a predicate $\varphi(x)$, for $k \in \mathbb{N}$, if every run started at an initial configuration (C, OF, σ) stabilises to $\varphi(|C|)$.

Note that this definition limits population programs to decide only unary predicates.

Notation When analysing population programs it often suffices to consider only the register configuration. Let $C, C' \in \mathbb{N}^Q$, $b \in \{\mathsf{false}, \mathsf{true}\}$ and let $f \in \mathrm{Proc}$ denote a procedure. We consider the possible outcomes when executing f in a configuration with registers C. Note that the program is nondeterministic, so multiple outcomes are possible. If f may return b with register configuration C', we write $C, f \to C', b$. For procedures not returning a value, we use $C, f \to C'$ instead. If f may initiate a restart, we write $C, f \to \textbf{restart}$. If f may hang or not terminate, we write $C, f \to \bot$. Finally, we define $\mathrm{post}(C, f) := \{S : C, f \to S\}$.

Example. An example is provided in Figure 1. The illustrated population program decides the predicate $\varphi(x) \Leftrightarrow 4 \leq x \leq 7$. It uses multiple procedures: $\textsc{Main}$ is run initially and decides the predicate, $\textsc{Test}(i)$ tries to move i units from x to y and reports whether it succeeded, and $\textsc{Clean}$ checks whether z is empty and moves some number of units from y to x. If $\textsc{Clean}$ detects an agent in z, it restarts the computation. As every run calls $\textsc{Clean}$ infinitely often, this serves to reject initial configurations where z is nonzero; eventually the protocol will be restarted with $z = 0$.

This program is only an example, and some simplifications are possible. E.g. the instruction ($\textbf{swap}\ x, y$) in $\textsc{Clean}$ is superfluous; additionally, instead of checking $z > 0$ one could omit that register entirely.

5. High-level Overview

In this section we want to give an intuitive explanation of our result. As mentioned, we use the technique of Lipton [24] to count to 2^{2^n} using $4n$ registers. We will give a brief explanation of the original technique in Section 5.1.

A straightforward application of the above technique only works if some guarantees are provided for the initial configuration (e.g. that the $4n$ registers used are empty, while an additional register holds all input agents). Sadly, no such guarantees are given in our model. Instead, we have to deal with adversarial initialisation, i.e. the notion that registers hold arbitrary values in the initial configuration. Section 5.2 describes the problems that arise, as well as our strategies for dealing with them.

5.1. Double-exponential counting

The biggest limitation of population programs (as for population protocols in general) is their inability to detect absence of agents. This is reflected in the (**maybe** $x > 0$) primitive; it may return true and thereby certify that x is nonzero, but it may always return false, regardless of whether $x = 0$ actually holds.

However, if we have two registers $x, \overline{x}$ and ensure that the invariant $x + \overline{x} = k$ holds, for some fixed $k \in \mathbb{N}$, then $x = 0$ is equivalent to $\overline{x} \geq k$. Crucially, it is possible to certify the latter property; if we have a procedure for checking $\overline{x} \geq k$, we can run both checks ($x > 0$ and $\overline{x} \geq k$) in a loop until one of them succeeds. Therefore, we may treat x as k-bounded register with deterministic zero-checks.

This seems to present a chicken-and-egg problem: to implement this register we require a procedure for $\overline{x} \geq k$, but checking such a threshold is already the overall goal of the program. However, using this idea we can implement a bootstrapping sequence. For small k, e.g. $k = 2$, one can easily implement the required $\overline{x} \geq k$ check. We use that as subroutine for *two* k-bounded registers, x and y. Using the deterministic zero-checks, x and y can together simulate a single k^2-bounded register with deterministic zero-check; this then leads to a procedure for checking $\overline{z} \geq k^2$ (for some other register $\overline{z}$).

To summarise, we have n levels of registers, with four registers $x_i, y_i, \overline{x}_i, \overline{y}_i$ on each level $i \in \{1, ..., n\}$. For each level we have a constant $N_i \in \mathbb{N}$ and ensure that $x_i + \overline{x}_i = y_i + \overline{y}_i = N_i$ holds. These constants grow by repeated squaring, so e.g. $N_1 = 2$ and $N_{i+1} = N_i^2$. Clearly, $N_n = 2^{2^n}$. (Our actual construction uses slightly different N_i.)

We have not yet broached the topic of initialising these counter s.t. the necessary invariants hold. Under assumptions on the input configuration this can easily be done by adapting the procedure for checking $\overline{z} \geq k^2$, given two k-bounded registers x, y, to instead move k^2 units into $\overline{z}$. However, we cannot make such assumptions in our model, and must instead check whether the initial configuration is adequate. At that point, initialisation is superfluous; we might as well require the initial configuration to have the registers already initialised.

5.2. Error detection

As we cannot rely on the existence of leaders, our model provides only weak guarantees. In particular, we must deal with adversarial initialisation, meaning that the initial configuration can assign arbitrary values to any register. This is not limited to a designated set of initial registers; all registers used in the computation are affected.

Let us first discuss how the above construction behaves if its invariants are violated. As above, let $x, \overline{x}$ denote registers for which we want to keep the invariant $x + \overline{x} = k$, for some $k \in \mathbb{N}$. If instead $x + \overline{x} > k$, the zero check is still guaranteed to terminate, as either $x > 0$ or $\overline{x} \geq k$ must hold. However, it might erroneously return that $x = 0$ when it is not. The procedure we use to combine two k-bounded counter to simulate a k^2-bounded one exhibits erratic behaviour under these circumstances; when we try to use it to count to k^2 we might instead only count to some lower value $k' < k^2$, even $k' \in \mathcal{O}(k)$. This leads us to the case of $x + \overline{x} < k$; here we can never detect $x = 0$ and will instead run into an infinite loop.

The latter case is more problematic, as detecting it would require detecting absence. For the former, we can ensure that we check $x + \overline{x} \geq k + 1$ infinitely often; if $x + \overline{x} > k$, this check will eventually return true and we can initiate a restart. While we cannot *detect* the latter case, we can *exclude* it: we issue a single check $x+\overline{x} \geq k$ in the beginning. If it fails, we restart immediately.

A simplified model. Of course, the full picture is more complicated, as we have many levels of registers that rely on each other. We will now consider a simplified model to explain the main ideas.

In our simplified model there is only a single register x_i per level $i \in \{1, ..., n\}$ as well as one "level $n + 1$" register R. For $i \in \{1, ..., n\}$ we are given subroutines $\textsc{Check}(x_i \geq N_i)$ and $\textsc{Check}(x_i > N_i)$ which we use to check thresholds; however, they are only guaranteed to work if $x_1 = N_1$, $x_2 = N_2$, ..., $x_{i-1} = N_{i-1}$ hold.

Our goal is to decide the threshold predicate $m \geq \sum_i N_i$, where $m := \sum_i x_i + \mathsf{R}$ is the total number of agents. For each possible value of m we pick one initial configuration C_m and design our procedure s.t.

- every configuration different from C_m will cause a restart, and
- if started on C_m it is *possible* that we enter a state from which we cannot restart.

The structure of C_m is simple: we pick the largest i s.t. we can set $x_j := N_j$ for $j \leq i$ and put the remaining units into x_{i+1} (or R, if $i = n$). The procedure works as follows:

1. We nondeterministically guess $i \in \{1, ..., n\}$.
2. We run $\textsc{Check}(x_j \geq N_j)$ for all $j \in \{1, ..., i\}$. If one of these checks fails, we restart.
3. According to $i = n$ we set the output flag to true or false.
4. To verify that we are in C_m, we check the following infinitely often. For $j \in \{1, ..., i\}$ we run $\textsc{Check}(x_j > N_j)$ and restart if it succeeds. If $i < n$ we also restart if $\textsc{Check}(x_{i+1} \geq N_{i+1})$ or one of $x_{i+2}, ..., x_n, \mathsf{R}$ is nonempty.

Clearly, when started in C_m and i is guessed correctly, it is possible for step 2 to succeed, and it is impossible for step 4 to restart. If i is too large, step 2 cannot work, and if i is too small step 4 will detect $x_{i+1} \geq N_{i+1}$. So the procedure will restart until the right i is guessed and step 4 is reached.

Consider an initial configuration $C \neq C_m$, $|C| = m$. There are two cases: either there is a k with $C(x_k) < C_m(x_k)$, or some k has $C(x_k) > C_m(x_k)$. Pick a minimal such k.

In the former case, step 2 can only pass if $i < k$, but then one of $x_{i+2}, ..., x_n, \mathsf{R}$ is nonempty and step 4 will eventually restart.

The latter case is more problematic. Step 2 can pass regardless of i (for $i > k$ the precondition of $\textsc{Check}$ is not met). In step 4, either $i < k$ and then $x_{i+1} \geq N_{i+1}$ or one of $x_{i+2}, ..., x_n, \mathsf{R}$ is nonempty, or $i \geq k$ and one the checks $\textsc{Check}(x_j > N_j)$ will eventually restart, for $j = k$.

This would be what we are looking for, but note that we implicitly made assumptions about the behaviour of $\textsc{Check}$ when called without its precondition being met. We need two things: all calls to Check terminate and they do not change the values of any register. The second is easy to ensure, as our algorithms will later only move agents between registers of the same level. The former is more difficult; it runs into the problem discussed above, where a zero-check might not terminate if the invariant of its register is violated. In this simplified model it corresponds to the case $x_i < N_i$.

Here the crucial insight is that $\textsc{Check}(x_i \geq N_i)$ and $\textsc{Check}(x_i > N_i)$ are only called if $(x_1, ..., x_{i-1}) \geq_{\text{lex}} (N_1, ..., N_{i-1})$, where $\geq_{\text{lex}}$ denotes lexicographical ordering. So if the precondition is violated, there must be a $j < i$ with $(x_1, ..., x_{j-1}) = (N_1, ..., N_{j-1})$ and $x_j > N_j$. This can be detected within the execution of Check by calling itself recursively. Using this technique, we can implement Check in a way that avoids infinite loops as long as the weaker precondition $(x_1, ..., x_{i-1}) \geq_{\text{lex}} (N_1, ..., N_{i-1})$ holds.

Our actual construction follows the above closely; of course, instead of a single register per level we have four, making the necessary invariants more complicated. Additional issues arise when implementing Check, as registers cannot be detected erroneous while in use. Certain subroutines must hence take care to ensure termination, even when the registers they use are not working properly.

6. A Succinct Population Program

We now present the details of our construction and prove its correctness. The goal is to show the following theorem.

Theorem 2. *Let $n \in \mathbb{N}$. There exists a population program deciding $\varphi(x) \Leftrightarrow x \geq k$ with size $\mathcal{O}(n)$, for some $k \geq 2^{2^{n-1}}$.*

For the remainder of this section, we construct a population program $\mathcal{P} = (Q, \text{Proc})$. We use registers $Q := Q_1 \cup ... \cup Q_n \cup \{\mathsf{R}\}$, where $Q_i := \{x_i, y_i, \overline{x}_i, \overline{y}_i\}$ are *level* i registers and R is a *level* $n+1$ register. For convenience, we identify $\overline{\overline{x}}$ with x for any register x.

As explained in the previous section, x_i and $\overline{x}_i$ are supposed to sum to a constant N_i, which we define via $N_1 := 1$ and $N_{i+1} := (N_i + 1)^2$.

First, we introduce the necessary formal definitions to precisely state the guarantees of each procedure. Before we move onto the individual procedures we then give brief summaries of each as well as the logical structure of the correctness proof.

Definitions. Let $C \in \mathbb{N}^Q$ and $i \in \{1, ..., n\}$. We say that C is

- *i-proper*, if $C(x_i) = C(y_i) = 0$ and $C(\overline{x}_i) = C(\overline{y}_i) = N_i$ for $i \in \{1, ..., n\}$
- *weakly i-proper*, if C is $(i-1)$-proper and $C(x) + C(\overline{x}) = N_i$ for $x \in \{x_i, y_i\}$
- *i-low*, if C is $(i-1)$-proper, not i-proper, and $C(x) = 0$ and $C(\overline{x}) \leq N_i$ for all $x \in \{x_i, y_i\}$
- *i-high*, if C is $(i-1)$-proper, not i-proper, and $C(x) + C(\overline{x}) \geq N_i$ for all $x \in \{x_i, y_i\}$
- *i-empty*, if $C(x) = 0$ for all $x \in Q_i \cup ... \cup Q_n \cup \{\mathsf{R}\}$

A procedure f is *i-robust* if for all i-high C we have $C, f \not\to \bot$ and $C, f \to C', b$ (or $C, f \to C'$) implies that C' is i-high as well. Note that $C, f \to$ **restart** is allowed. Finally, f is *robust* if it is i-robust for $i \in \{1, ..., n\}$.

We set $\mathrm{ctr}_{x,y}(C) := C(x) \cdot (N_i + 1) + C(y)$ to the value of the two-digit, base $N_i + 1$ counter using x and y as digits, where $C \in \mathbb{N}^Q$, $i \in \{1, ..., n\}$ and $x \in \{x_i, \overline{x}_i\}, y \in \{y_i, \overline{y}_i\}$.

As a general remark, we sometimes use the slightly unusual notation $\{x : \alpha\}$, where α is independent of x. This denotes either $\{x\}$, if α, or $\emptyset$ otherwise.

Let us give some brief intuition. Most routines require i-proper registers to work correctly, while weakly i-proper configurations are intermediate configurations appearing while the counters are in use. Configurations deviating from the above can be either i-high or i-low (roughly). We can exclude the latter so that it is not problematic, but procedures must provide guarantees when run on i-high configurations, specified by the notion of robustness as well as procedure-specific guarantees.

Summary. We use the following procedures.

- Main. Computation starts by executing this procedure, and Main ultimately decides the predicate $\varphi(x) \Leftrightarrow x \geq 2\sum_{i=1}^{n} N_i$. Similar to the simplified model in Section 5.2, depending on the number of agents m a small number of initial configurations are considered "good" and may stabilise to the correct output (otherwise they restart), while all other configurations always cause a restart. More precisely, a configuration is good if it is i-low and $(i+1)$-empty for some i, or if it is n-proper.

- CheckEmpty. This is a simple helper procedure that checks whether a configuration is i-empty and initiates a restart otherwise.

- CheckProper. Similar to the previous subroutine, this checks whether a configuration is i-proper or i-low.

- Large. It checks whether a register $x \in Q_i$ is at least N_i. If so, it may return true (but also false), and otherwise it always returns false. As a side-effect, if it returns true it exchanges the units of $x - N_i$ and $\overline{x}$. This can be used to test for $x > N_i$ by checking whether $\overline{x}$ is empty afterwards. Of course, if the invariant $x + \overline{x} = N_i$ is met, Large cannot have an effect.

- Zero. Using Large, this procedure implements a deterministic zero-check on a register $x \in Q_i$ (as long as the invariant $x + \overline{x} = N_i$ holds).

- IncrPair. As described in Section 5.1, we use two level i registers (which are N_i bounded) to simulate a N_{i+1}-bounded register. This procedure implements the increment operation for the simulated register.

Logical structure. The above procedures (except for Main) are instantiated for each level and call each other recursively. Population programs allow only acyclic procedure calls, so the correctness proofs can proceed inductively and rely on the correctness of all called procedures. To be formally precise, we must note that the proofs of the following lemmata do not prove the associated lemma independently of the others. They only prove part of the induction step, and only if all proofs work do the statements of the lemmata follow.

6.1. CheckEmpty and CheckProper

We start with the two simplest procedures. CheckEmpty is supposed to determine whether a configuration is i-empty, which can easily be done by checking whether the relevant registers are nonempty.

Lemma 3. *Let $C \in \mathbb{N}^Q$, $i \in \{1, ..., n+1\}$. Then* $\mathrm{post}(C, \textsc{CheckEmpty}(i)) = \{C\} \cup S$, *where $S = \emptyset$ if C is i-empty and $S = \{\textbf{restart}\}$ otherwise. Moreover,* $\textsc{CheckEmpty}(i)$ *is robust.*

Proof. Clearly, CheckEmpty cannot affect any register, and restarts only if one of the registers $Q_i \cup ... \cup Q_n \cup \{\mathsf{R}\}$ is nonzero. Robustness follows immediately. ☐

The procedure CheckProper is used to ensure that the current configuration is not i-high. If it is, it may initiate a restart. We remark that calls to $\textsc{CheckProper}(0)$ have no effect and can simply be omitted.

Lemma 4. *Let $C \in \mathbb{N}^Q$, $i \in \{1, ..., n\}$. Then*

(a) $\mathrm{post}(C, \textsc{CheckProper}(i)) = \{C\}$ *if C is i-proper or i-low,*

(b) $C, \textsc{CheckProper}(i) \to \textbf{restart}$ *if C is j-high, for some $j \in \{1, ..., i\}$,*

(c) $C, \textsc{CheckProper}(i) \to \textbf{restart}$ *if C is $(i-1)$-proper and $C(x) > 0 \lor C(\overline{x}) > N_i$, for some $x \in \{x_i, y_i\}$, and*

(d) $\textsc{CheckProper}(i)$ *is robust.*

Proof. **(a)** By induction, the recursive call in line 2 must return with C. As C is weakly i-proper, line 6 has no effect (Lemma 7a) and neither line 5 nor line 8 is executed.

(b) The case $j < i$ is covered inductively, otherwise it follows directly from (c).

(c) If $C(x) > 0$, line 5 may execute a restart. If $C(\overline{x}) > N_i$, we use Lemma 7b to derive that x may be nonzero at line 7. If $x = y_i$, we must also note that Lemma 7b ensures

---

Algorithm CheckEmpty**.** Verify that registers are empty.

Parameter: $i \in \{1, ..., n+1\}$
Effect: If i-empty, do nothing, else it may restart

```
procedure CheckEmpty(i) [for i ≤ n]
    CheckEmpty(i + 1)
    for x ∈ Q_i do
        if maybe x > 0 then
            restart
procedure CheckEmpty(i) [for i = n + 1]
    if maybe R > 0 then
        restart
```

Algorithm CheckProper**.** Verify that registers are proper.

Parameter: $i \in \{1, ..., n\}$

Effect: If i-proper or i-low, do nothing, else it may restart.

procedure CheckProper(i)
  CheckProper($i-1$)
  for $x \in \{x_i, y_i\}$ **do**
    if maybe $x > 0$ **then**
      restart
    Large($\overline{x}$)
    if maybe $x > 0$ **then**
      restart

that the first iteration of the for-loop either restarts or terminates without affecting x and $\overline{x}$.

(d) Let C be a j-high configuration, for some j. If $j > i$ then we need only invoke property (a). Otherwise, we use that CheckProper and Large are robust (Lemma 7c and induction), so their execution terminates and does not affect whether the configuration is j-high. ☐

6.2. Zero

This procedure implements a deterministic zero check, as long as the register configuration is weakly i-proper. To ensure robustness, CheckProper is called within the loop.

Lemma 5. *Let $i \in \{1, ..., n\}$, $x \in \{x_i, \overline{x}_i, y_i, \overline{y}_i\}$, $C, C' \in \mathbb{N}^Q$. Then*

(a) $\mathrm{post}(C, \textsc{Zero}(x)) = \{(C, C(x) = 0)\}$ *if C is weakly i-proper,*

(b) $\mathrm{post}(C, \textsc{Zero}(x)) = \{(C, \mathsf{false}) : C(x) > 0\} \cup \{(C', \mathsf{true}) : C(\overline{x}) \geq N_i)\}$ *if C is $(i-1)$-proper and $C(x) + C(\overline{x}) \geq N_i$, where $C'(\overline{x}) = C(x) + N_i$, $C'(x) = C(\overline{x}) - N_i$ and $C'(z) = C(z)$ for $z \notin \{x, \overline{x}\}$*

(c) $C, \textsc{Zero}(x) \to C', \mathsf{false}$ *implies $C'(x) > 0$, for all C', and*

(d) $\textsc{Zero}(x)$ *is robust.*

Proof. **(a)** This follows immediately from (b): if $C(x) + C(\overline{x}) = N_i$ then $C(x) = 0$ is equivalent to $C(\overline{x}) \geq N_i$, and $C' = C$ (assuming $C(\overline{x}) \geq N_i$).

(b) As C is $(i-1)$-proper, the call to CheckProper has no effect (Lemma 4a). Further, Large has no effect as long as it returns false (Lemma 7b). Hence, for all iterations of the loop, registers start in C. Line 5, therefore, may execute iff $C(x) > 0$. Again due to Lemma 7b, line 7 can execute iff $C(\overline{x}) \geq N_i$, and if so, registers are according to C'. Finally, either $C(x) > 0$ or $C(\overline{x}) \geq N_i$ holds, so eventually line 5 or line 7 will return the correct result due to fairness and the procedure terminates.

(c) This follows from the observation that false can only be returned in line 5.

(d) Let C be j-high. If $j > i$ we can invoke property (a). For $j = i$ we use (b), noting that C' is still i-high. Otherwise, we use that CheckProper and Large are robust

Algorithm Zero. Check whether a register is 0.

Parameter: $x \in \{x_i, \overline{x}_i, y_i, \overline{y}_i\}$
Output: whether $x = 0$

1: **procedure** Zero(x)
2:     **while** true **do**
3:         CheckProper($i-1$)
4:         **if maybe** $x > 0$ **then**
5:             **return** false
6:         **if** Large($\overline{x}$) **then**
7:             **return** true

Algorithm IncrPair. Decrement a two-digit, base $\beta := N_i + 1$ register

Parameter: $x \in \{x_i, \overline{x}_i\}$, $y \in \{y_i, \overline{y}_i\}$
Effect: $\beta x + y \pmod{\beta^2}$ decreases by one

1: **procedure** IncrPair(x, y)
2:     **if** Zero($\overline{y}$) **then**
3:         **swap** $y, \overline{y}$
4:         **if** Zero($\overline{x}$) **then**
5:             **swap** $x, \overline{x}$
6:         **else**
7:             $\overline{x} \mapsto x$
8:     **else**
9:         $\overline{y} \mapsto y$

and do not affect whether the register configuration is j-high. Finally, we know that line 3 is eventually going to restart (Lemma 4b and fairness), so the loop cannot repeat infinitely often. □

6.3. IncrPair

This is a helper procedure to increment the "virtual", N_{i+1}-bounded counter simulated by x and y. It works by first incrementing the second digit, i.e. y. If an overflow occurs, x is incremented as well. It is also be used to decrement the counter, by running it on $\overline{x}$ and $\overline{y}$.

Property (b) of the following lemma is interesting; it states that IncrPair is "reversible" in some sense, under only the weak assumption that the configuration is i-high (or i-proper). We need this property later to show that Large is robust.

Regarding property (c) we remark that, contrary to the other procedures, IncrPair is not j-robust for all j, but only $j \leq i$. This is simply due to the fact that it is designed to change the value of level i registers; if executed on an i-proper configuration it results only in a weakly i-proper configuration.

Lemma 6. *Let* $i \in \{1, ..., n\}$, $x \in \{x_i, \overline{x}_i\}, y \in \{y_i, \overline{y}_i\}$, $C, C' \in \mathbb{N}^Q$. *Then*

(a) $\mathrm{post}(C, \textsc{IncrPair}(x, y)) = \{C'\}$ *if* C *is weakly* i*-proper, where* C' *is the unique weakly* i*-proper multiset with* $\mathrm{ctr}_{x,y}(C') = \mathrm{ctr}_{x,y}(C) + 1 \pmod{N_{i+1}}$ *and* $C'(w) = C(w)$ *for* $w \notin \{x_i, \overline{x}_i, y_i, \overline{y}_i\}$*,*

(b) $C, \textsc{IncrPair}(x, y) \to C'$ *implies both* $C', \textsc{IncrPair}(\overline{x}, \overline{y}) \to C$ *and* $C'(z) = C(z)$ *for* $z \notin Q_i$*, if* C *is* $(i-1)$*-proper and* $C(w) + C(\overline{w}) \geq N_i$ *for* $w \in \{x_i, y_i\}$*, and*

(c) $\textsc{IncrPair}(x, y)$ *is* j*-robust, for* $j \leq i$*.*

Proof. **(a)** If C is weakly i-proper, the calls to $\textsc{Zero}$ work deterministically and the registers x and y are adjusted according to the specification: line 2 checks whether y (the least significant digit) is N_i. If not, it is incremented. Otherwise, it overflows; y is set to 0 and x is incremented, checking whether it overflows as well. Finally, note $N_{i+1} = (N_i + 1)^2$.

(b) The property $C'(z) = C(z)$ for $z \notin Q_i$ follows immediate from Lemma 5b. In particular, lines 4-7 only affect the values of x and $\overline{x}$, while lines 2,3 and 9 only affect y and $\overline{y}$. We now consider executing $\textsc{IncrPair}$ twice, first with arguments x, y, then with $\overline{x}, \overline{y}$. We start with registers C, and argue that it is possible for the second execution to take the same branches (in lines 2 and 4) as the first. Afterwards we derive that the registers again have values C.

Consider line 2. If the branch is not taken, $\textsc{Zero}$ had no effect. After $\overline{y} \mapsto y$ in line 9, clearly $C'(y) > 0$. In the second execution, line 2 runs $\textsc{Zero}(y)$ (recall that the second execution has different arguments). This may now return false and the same branch is taken.

If the branch in line 2 is taken, after line 3 registers $y, \overline{y}$ have been changed. More precisely, N_i units have been moved from y to $\overline{y}$. Lines 4-7 do not affect $y, \overline{y}$, so $C'(\overline{y}) \geq N_i$. In the second execution, the call $\textsc{Zero}(y)$ may then return true.

The argument for the branch in line 4 is analogous. Finally, we argue that, if the same branches are taken, the second execution undoes the changes of the first. Briefly, if the branch in line 2 is not taken, only line 9 changes any registers. Clearly, executing $\overline{y} \mapsto y$ and then $y \mapsto \overline{y}$ has no effect. If it is taken, the combined effect of lines 2 and 3 is moving N_i units from y to $\overline{y}$, which are then moved back in the second execution. Again the situation for lines 4-7 is analogous.

(c) Let C be j-high, for $j \leq i$. As $\textsc{Zero}$ is robust, it does not affect whether the register configuration is j-high and either terminates or restarts. Lines 3,5,7 and 9, if executed, also do not affect j-highness. Finally, there is no loop and Lemma 5c implies that lines 7 and 9 cannot hang, so $\textsc{IncrPair}$ either terminates or restarts. □

6.4. Large

This is the last of the subroutines, and the most involved one. The goal is to determine whether $x \geq N_i$, by using the registers of level $i - 1$ to simulate a "virtual" N_i-bounded register. To ensure that the procedure is robust, we have opted for a "random" walk, which nondeterministically moves either up or down. More concretely, at each step either x is found nonempty, one unit is moved to $\overline{x}$ and the virtual register is incremented, or

Algorithm LARGE. Nondeterministically check whether a register is maximal.

Parameter: $x \in \{x_i, \overline{x}_i, y_i, \overline{y}_i\}, x \neq y$
Output: if $x \geq N_i$ return true and swap units of $x - N_i$ and $\overline{x}$; or return false

1: **procedure** LARGE(x) [for $i = 1$]
2:   **if maybe** $x > 0$ **then**
3:     $x \mapsto \overline{x}$
4:     **swap** $x, \overline{x}$
5:     **return** true
6:   **else**
7:     **return** false

8: **procedure** LARGE(x) [for $i > 1$]
9:   **if** $\neg$ZERO(x_{i-1}) $\vee$ $\neg$ZERO(y_{i-1}) **then**
10:     **restart**
11:   **while** true **do**
12:     CHECKPROPER($i-2$)
13:     **if maybe** $x > 0$ **then**
14:       $x \mapsto \overline{x}$
15:       INCRPAIR(x_{i-1}, y_{i-1})
16:       **if** ZERO(x_{i-1}) $\wedge$ ZERO(y_{i-1}) **then**
17:         **swap** $x, \overline{x}$
18:         **return** true
19:     **else**
20:       **if** ZERO(x_{i-1}) $\wedge$ ZERO(y_{i-1}) **then**
21:         **return** false
22:       **if maybe** $\overline{x} > 0$ **then**
23:         $\overline{x} \mapsto x$
24:         INCRPAIR($\overline{x}_{i-1}, \overline{y}_{i-1}$)

conversely $\overline{x}$ is nonempty, one unit moved to x, and the virtual register decremented. If the virtual register reaches 0 from above, LARGE had no effect and returns false. Once the virtual register overflows, a total of N_i units have been moved. These are put back into x by swapping x and $\overline{x}$ and true is returned.

In the previous section we have shown that INCRPAIR is in some sense reversible even under weak assumptions. This property will ensure that the random walk terminates in that case, as it can always retrace its prior steps to go back to its starting point.

Lemma 7. *Let $i \in \{1, ..., n\}$, $x \in \{x_i, \overline{x}_i, y_i, \overline{y}_i\}$, and $C \in \mathbb{N}^Q$. Then*

(a) post(C, LARGE(x)) $= \{(C, \text{false}), (C, C(x) \geq N_i)\}$ *if C is weakly i-proper,*

(b) post(C, LARGE(x)) $= \{(C, \text{false})\} \cup \{(C', \text{true}) : C(x) \geq N_i\}$ *if C is $(i-1)$-proper, with $C'(x) = C(\overline{x}) + N_i$, $C'(\overline{x}) = C(x) - N_i$ and $C'(z) = C(z)$ for $z \notin \{x, \overline{x}\}$, and*

(c) LARGE(x) *is robust.*

Proof. **(a)** Follows directly from (b); if $C(x) \geq N_i$ and C is weakly i-proper, then $C(x) = N_i$ and $C(\overline{x}) = 0$, which implies $C' = C$.

(b) The case $i = 1$ is trivial. Assume $i > 1$. The registers will remain in a weakly $(i-1)$-proper configuration; lines 14, 17 and 23 do not affect this, and neither do the calls to INCRPAIR (Lemma 6a), to CHECKPROPER (Lemma 4a), nor to ZERO (Lemma 5a). As the registers are weakly $(i-1)$-proper, the calls to ZERO work as intended and deterministically check whether the register is zero (again, Lemma 5a). In particular, using $C(x_{i-1}) = C(y_{i-1}) = 0$ we find that line 10 cannot execute. Additionally, since the registers remain weakly $(i-1)$-proper and thus $(i-2)$-proper, line 12 has no effect (Lemma 4a).

We consider the register simulated by IncrPair; for convenience we introduce the shorthand $\mathrm{ctr} := \mathrm{ctr}_{x_{i-1},y_{i-1}}$. As C was $(i-1)$-proper, $\mathrm{ctr}(C) = 0$. This counter is only modified by the calls to IncrPair, as specified by Lemma 6a. Line 15 increments the counter, and line 24 decrements it. Line 15 may overflow the counter, but then the branch in line 16 will immediately be taken. Line 24 can only execute of the check in line 20 fails, so it cannot underflow the counter.

As the counter neither over- nor underflows, for any register configuration C^* the procedure reaches at the beginning of the loop in line 12, $\mathrm{ctr}(C^*)$ correspond to units moved from x to $\overline{x}$ via lines 14 and 23.

We now show $C, \textsc{Large}(x) \to C, \mathsf{false}$ and, if $C(x) \geq N_i$, $C, \textsc{Large}(x) \to C', \mathsf{true}$. For the former, we even show the stronger property that C, false can be returned from any iteration of the loop. Let C^* denote some configuration reached at line 12. From now on, we never take the branch in line 13. If $\mathrm{ctr}(C^*) = 0$, then we claim $C^*(z) = C(z)$ for all z. If z has level at most $i-2$ this follows from C^* being weakly $(i-1)$-proper. If z has level $i-1$, we use $\mathrm{ctr}(C^*) = 0 = \mathrm{ctr}(C)$. For z at level i or above, note that only registers x and $\overline{x}$ can be modified by the procedure, but $\mathrm{ctr}(C^*) = 0$ ensures that no units have moved between them. Using $C^* = C$ we now see that the branch in line 20 can be taken and we return false with registers C.

If $\mathrm{ctr}(C^*) > 0$, then the branch in line 20 cannot be taken. Using $C^*(\overline{x}) \geq \mathrm{ctr}(C^*) > 0$, we can take the branch in line 22. In the next iteration of the loop we have decreased ctr by one; the property then follows from induction. We remark that this also shows that the procedure always terminates.

We now prove $C, \textsc{Large}(x) \to C', \mathsf{true}$, assuming $C(x) \geq N_i$. Here, it is possible to take the branch in line 13 N_i times and we do. Afterwards, the counter overflows and line 18 returns true. As before, the only registers that may have changed relative to C are x and $\overline{x}$. We moved N_i units from x to $\overline{x}$, swapping them then results in C'.

Finally, we need to show that the above two cases cover all possibilities. We already argued that the procedure always terminates and no restart can occur. If we return in line 18, the counter was overflowed and N_i units have been moved, resulting in C'. If we return in line 21, changes to x and $\overline{x}$ have cancelled out, and we are in C.

(c) Let C be a j-high configuration, for some j. If $j \geq i$ we need only refer to (b), noting that C' is still j-high. For $j < i$ we can rely on CheckProper, Zero and IncrPair being j-robust (lemmata 4d, 5d and 6c). In particular, they do not affect whether the register configuration is j-high. Neither do lines 14, 17 or 23, so the registers stay j-high. Additionally, this yields that the calls to these procedures terminate or restart.

It remains to argue that the loop terminates. If $j \leq i-2$ this is ensured by CheckProper (Lemma 4b), so we are left with $j = i-1$. In this case the call to CheckProper in line 12 has no effect and we shall ignore it. Further, note that the calls to Zero and IncrPair can only change a register z if z or $\overline{z}$ is one of their arguments (lemmata 5b and 6b).

Let $\mathcal{C}$ denote the set of j-high configurations. For $D, D' \in \mathcal{C}$ we write $D \sim D'$ if one iteration of the loop (i.e. executing lines 12-24 in sequence), starting with registers according to D, may end with registers in D' (without returning). We now claim that $\sim$ is symmetric. To see that this claim suffices, let C^* denote the register configuration after line 9. Using Lemma 5b, $C^*(\overline{x}_{i-1}), C^*(\overline{y}_{i-1}) \geq N_i$ hold. Our claim then implies that the loop can go back to C^* after any number of iterations. Eventually, it will do so due to fairness. Then, it may take the else branch in line 19. Using Lemma 5b again, line 21 may execute and the procedure returns.

We now show the claim. Fix D, D' with $D \sim D'$. There are now two cases: either D' results from D by executing lines 14-16, or lines 20-24. We now need to argue that D may result if the loop starts with D'. Consider the first case. The else branch in line 19 may always be taken, so it suffices that lines 20-24 may undo the effects of lines 14-16 from earlier. Due to line 14, $D'(\overline{x}) > 0$, and the branch in line 22 may be taken. Using Lemma 5b, lines 16 and 20 may cancel out, Lemma 6b implies that lines 15 and 24 may cancel, and lines 14 and 23 undo each other as well.

The argument for the second case is analogous. There, line 23 ensures that we can subsequently take the branch in line 13, and the lines cancel in the same manner. $\square$

6.5. Main

Finally, we put things together to arrive at the complete program. The implementation is very close to the steps described in Section 5.2 in the simplified model, but instead of guessing an i we iterate through the possibilities.

As mentioned before, Main considers a small set of initial configurations "good" and may stabilise. All other configurations lead to a restart. The following lemma formalises this property.

Lemma 8. *Let $C \in \mathbb{N}^Q$. Then* Main*, started with register configuration C, can only restart or stabilise. Additionally,*

(a) it may stabilise to false *if C is j-low and $(j+1)$-empty, for some $j \in \{1, ..., n\}$,*

Algorithm Main**.** Decide whether there are at least $2\sum_i N_i$ agents.

```
procedure Main
    OF := false
    for i = 1, ..., n do
        while ¬Large(x̄_i) ∨ ¬Large(ȳ_i) do
            CheckProper(i)
            CheckEmpty(i + 1)
    OF := true
    while true do
        CheckProper(n)
```

(b) *it may stabilise to* true *if* C *is* n*-proper, and*

(c) *it always restarts otherwise.*

Proof. The output register OF is only changed by lines 2 and 7. (This can easily be checked syntactically; no called procedure uses OF.) So either the execution restarts; or one of the two loops in lines 4 and 8 does not terminate and the computation stabilises.

Before moving to claims (a-c), we argue that, if C is i-proper, the i-th iteration of the for-loop in line 3 may terminates without effect, otherwise it restarts. Here, we use Lemma 7a to derive that line 4 has no effect and that the loop condition may be false; due to fairness the loop terminates eventually. Line 5 has no effect as well (Lemma 4a), and line 6 either restarts or does nothing (Lemma 3).

(a) C is $(j-1)$-proper, so, as argued above, iterations $i \in \{1, ..., j-1\}$ of the for-loop may terminate without changing a register, and they restart otherwise. In iteration $i = j$ the while-loop in line 4 cannot terminate, and lines 5-6 have no effect and cannot initiate a restart, so the computation stabilises to false.

(b) Now all n iterations of the for-loop in line 3 may terminate without effect (or restart, otherwise). If they do, we enter the second while-loop, in line 8, and stabilise to true.

(c) Let $j \in \{1, ..., n\}$ be maximal s.t. C is $j-1$-proper. (Such a j always exists.) As argued before, the first $j-1$ iterations of the for-loop cannot change any registers. There are the following cases.

Case 1, $C(\overline{x}) < N_i$ for some $x \in \{x_i, y_i\}$. Iteration $i = j$ cannot terminate, as $\textsc{Large}(\overline{x})$ will always return false. As C is not i-low, we have $C(y) > 0$ or $C(\overline{y}) > N_i$ for some $y \in \{x_i, y_i\}$. Using Lemma 4c, we then find that line 5 may restart. Due to fairness, this happens eventually.

Case 2, C is j-high. As $\textsc{Large}$, $\textsc{CheckProper}$ and $\textsc{CheckEmpty}$ are robust (lemmata 7c, 4d and 3), the register configuration will remain j-high. Assuming that no restart occurs we know that the subsequent computation would execute $\textsc{CheckProper}(k)$ infinitely often, for some $k \geq j$. (This occurs either in line 5, or line 9.) However, Lemma 4b guarantees that these calls may restart, so a restart will happen eventually due to fairness.

Note that the above case distinction is exhaustive, as C cannot be j-proper (either j would not be maximal, or C would be n-proper). ☐

We conclude this section by arguing that the overall population program is correct and fulfils the stated bounds.

Theorem 2. *Let* $n \in \mathbb{N}$*. There exists a population program deciding* $\varphi(x) \Leftrightarrow x \geq k$ *with size* $\mathcal{O}(n)$*, for some* $k \geq 2^{2^{n-1}}$*.*

Proof. We define $k := 2\sum_{i=1}^{n} N_i$. (Recall that $N_{i+1} = (N_i + 1)^2$ and $N_1 = 1$, implying $k \geq 2^{2^n}$.)

Let $m \in \mathbb{N}$ and let $\mathcal{C} := \{C \in \mathbb{N}^Q : |C| = m\}$ denote the configurations where registers sum to i. It suffices to show that $\mathcal{C}$ contains a "good" configuration; i.e. an n-proper configuration iff $m \geq k$, or a j-low and $(j+1)$-empty configuration for some $j \in \{1, .., n\}$ iff $m < k$. If these hold, Lemma 8 guarantees that every run starting with another kind of configuration eventually restarts. By fairness, at some point the computation restarts with a good configuration and stabilises to the correct output.

It remains to argue that the above claim holds. If $m \geq k$, we note that superfluous units can be left in register R, keeping the configuration n-proper; conversely, any n-proper configuration C clearly has $|C| \geq k$. Otherwise, a good configuration can have at most $k-1$ agents. To construct such a configuration, let j be maximal s.t. $2\sum_{i=1}^{j-1} N_i \leq m$. (We remark that $j \in \{1, ..., n\}$, due to $m < k$.) We now start with a $(j-1)$-proper and j-empty configuration C, and distribute the remaining $m - |C| \leq 2N_j$ units evenly across $\overline{x}_j$ and $\overline{y}_j$. The resulting configuration is j-low and $(j+1)$-empty.

Regarding the size bound, note that we have $4n + 1$ registers. We also have $\mathcal{O}(n)$ instructions: Main has $\mathcal{O}(n)$ instructions and exists only once, while every other procedure has constant length and is instantiated $\mathcal{O}(n)$ times. The swap-size is $\mathcal{O}(n)$ as well, as only registers x and $\overline{x}$ are swapped, for $x \in \bigcup_i Q_i$. $\square$

7. Converting Population Programs into Protocols

In the previous section we have shown that succinct population programs for the flock-of-birds predicate exist. We must now justify our model and prove that we can convert population programs into population protocols, keeping the number of states low. We do this in two steps; first we introduce population machines, which are a low-level representation of population programs, then we convert these into population protocols.

Population machines are introduced in Section 7.1, they serve to provide a simplified model. Converting population programs into machines is straightforward and uses standard techniques, similar to how one would convert a structured program to use only goto-statements. We will describe this in Section 7.2. The conversion to population protocols is then described in Section 7.3, resulting in the following statement:

Theorem 9. *If a population program deciding φ with size n exists, then there is a population protocol deciding $\varphi'(x) \Leftrightarrow \varphi(x-i) \wedge x \geq i$ with $\mathcal{O}(n)$ states, for an $i \in \mathcal{O}(n)$.*

Proof. This will follow from propositions 10 and 11, which are proved in the following two sections. $\square$

We remark that φ' is still a flock-of-bird predicate if φ is, and the constant has even increased.

7.1. Formal Model

Definition 1. *A* population machine *is a tuple $\mathcal{A} = (Q, F, \mathcal{F}, \mathcal{I})$, where Q is a finite set of* registers*, F a finite set of* pointers*, $\mathcal{F} = (\mathcal{F}_i)_{i \in F}$ a list of* pointer domains*, each of which is a finite set, and $\mathcal{I} = (\mathcal{I}_1, ..., \mathcal{I}_L)$ is a sequence of* instructions*, with $L \in \mathbb{N}$. Additionally, $OF, CF, IP \in F$, $\mathcal{F}_{OF} = \mathcal{F}_{CF} = \{\mathsf{false}, \mathsf{true}\}$ and $\mathcal{F}_{IP} = \{1, .., L\}$. For $x \in Q \cup \{\square\}$ we also require $V_x \in F$, and $x \in \mathcal{F}_{V_x} \subseteq Q$*

Let $x, y \in Q$, $x \neq y$, $X, Y \in F$, $l \in \{1, ..., L\}$ and $f : \mathcal{F}_Y \to \mathcal{F}_X$. There are three types of instructions: $\mathcal{I}_i = (x \mapsto y)$, $\mathcal{I}_i = (\textbf{maybe } x > 0)$, or $\mathcal{I}_i = (X := f(Y))$.

The size *of* $\mathcal{A}$ *is* $|Q| + |F| + \sum_{X \in F} |\mathcal{F}_X| + |\mathcal{I}|$.

A population machine has a number of registers, as usual, and a number of pointers. While each register can take any value in $\mathbb{N}$, a pointer is associated with a finite set of values it may assume. There are three special pointers: the output flag OF, which we have already seen in population programs and is used to indicate the result of the computation, the condition flag CF used to implement branches, and the instruction pointer IP, storing the index of the next instruction to execute. To implement swap instructions we use a register map; the pointer V_x, for a register $x \in Q$, stores the register x is actually referring to. ($V_\square$ is a temporary register for swapping.) The model allows for arbitrary additional pointers, we will use a one per procedure to store the return address.

There are only three kinds of instructions: $(x \mapsto y)$ and $(\mathbf{maybe}\ x > 0)$ are present in population programs as well and have the same meaning here. (With the slight caveat that x and y are first transformed according to the register map.) The third, $(X := f(Y))$ is a general-purpose instruction for pointers. It can change IP and will be used to implement control flow constructs. We now define the semantics.

Definition 2. *A* configuration *is a map C with $C(x) \in \mathbb{N}$ for $x \in Q$ and $C(X) \in \mathcal{F}_X$ for $X \in F$. The* output *of C is $C(OF)$. A configuration C is* initial *if $C(IP) = 1$ and $C(V_x) = x$ for $x \in Q$. For two configurations C, C' we write $C \to C'$ if*

- $\mathcal{I}_{C(IP)} = (x \mapsto y)$, $C'(IP) = C(IP) + 1$, $C'(C(V_x)) = C(C(V_x)) - 1$, $C'(C(V_y)) = C(C(V_y)) - 1$ *and* $C'(z) = C(z)$ *for* $z \notin \{IP, C(V_x), C(V_y)\}$,
- $\mathcal{I}_{C(IP)} = (\mathbf{maybe}\ x > 0)$, $C'(IP) = C(IP) + 1$, $C'(CF) \in \{\mathsf{false}, C(C(V_x)) > 0\}$ *and* $C'(z) = C(z)$ *for* $z \notin \{IP, CF\}$,
- $\mathcal{I}_{C(IP)} = (X := f(Y))$, $X \neq IP$, $C'(IP) = C(IP) + 1$, $C'(X) = f(C(Y))$ *and* $C'(z) = C(z)$ *for* $z \notin \{IP, X\}$, *or*
- $\mathcal{I}_{C(IP)} = (IP := f(Y))$, $C'(IP) = f(C(Y))$ *and* $C'(z) = C(z)$ *for* $z \notin \{IP\}$.

To make the $\to$ relation left-total, we also define $C \to C$ if there is no $C' \neq C$ with $C \to C'$.

The above definition allows for the computation to "hang" in certain situations, e.g. when executing $x \mapsto y$ while x is 0. If this happens, the computation enters an infinite loop and makes no progress.

We use the general definitions of stable computation from Section 3. We say that $\mathcal{A}$ *decides* a predicate $\varphi(x)$ if every fair run starting at an initial configuration C stabilises to $\varphi(\sum_{q \in Q} C(q))$.

7.2. From Population Programs to Machines

Population machines do not have high-level constructs such as loops or procedures, but these can be implemented as macros using standard techniques. Let $\mathcal{P} = (Q, \mathrm{Proc})$ denote a population program, we convert it to a population machine $\mathcal{A} = (Q, F, \mathcal{F}, \mathcal{I})$.

If and while. Our model allows for direct manipulation of the instruction pointer. We use this to implement both conditional and unconditional jumps. To evaluate branches, we use the CF pointer to store the intermediate boolean results. An example is given in Figure 2. For more complicated boolean formulae one needs multiple jumps, but since we limited population programs to binary expressions (with optional negations) we only need a constant number of instructions for each branch.

Recall also that for-loops are only a macro in population programs, so we do not have to implement them here.

while $\neg$(**maybe** $x > 0$) **do** $x \mapsto y$ ...	$\rightsquigarrow$	1: **maybe** $x > 0$ 2: $IP := \begin{cases} 5 & \text{if } CF \\ 1 & \text{else} \end{cases}$ 3: $x \mapsto y$ 4: $IP := 1$ 5: ...

Figure 2: Implementation of a while-loop.

Procedure calls. In a population program, procedures only have bounded recursion. More precisely, the directed graph of calls is acyclic. Recall also that procedures do not take arguments, instead the parameters specify a family of procedures. To take an example from Section 6, CheckProper is not a procedure, but CheckProper(1), ..., CheckProper(n) are. Hence our implementation only needs to deal with returning from a procedure, jumping to the correct instruction and propagating the return value.

For the former, we use a pointer P for each procedure $P \in \mathrm{Proc}$. This pointer has domain $\mathcal{F}_P \subseteq \{1, ..., L\}$. Calling a procedure involves setting this pointer to the address the procedure should return to, before jumping to the first instruction of the procedure. To propagate return values, we store them in CF. A simple example is shown in Figure 3. While $\mathcal{F}_P := \{1, ..., L\}$ would work, we limit $\mathcal{F}_P$ to contain only the necessary elements (i.e. one per call of P) to reduce the size of the resulting machine.

The population program is specified to start by executing Main, so we insert a call to it as the first instruction followed by an infinite loop in case Main returns.

AddTwo
...
procedure AddTwo
    $x \mapsto y$
    $x \mapsto y$
    return true
	$\rightsquigarrow$	1: AddTwo $:= 3$
		2: $IP := 4$
		3: ...
		4: $x \mapsto y$
		5: $x \mapsto y$
		6: $CF :=$ true
		7: $IP :=$ AddTwo

Figure 3: Implementation of a procedure.

Swaps. Most of the heavy lifting is in the definition of the machine model (and the later conversion to population protocols). To implement (**swap** x, y) we replace it by the instructions ($V_\square := V_x; V_x := V_y; V_y := V_\square$), which adjust the register map. Similar to procedure calls, we prune $\mathcal{F}_{V_x}$ to contain only necessary elements to reduce size; the sum $\sum_{x \in Q} |\mathcal{F}_{V_x}|$ then matches to the swap-size introduced in Section 4.

Restarts. A restart changes registers arbitrarily and then continues execution at the beginning. We first transform the population program so that it does the first part by itself, as sketched in Figure 4. Afterwards, a restart is simply $IP := 1$. (One could reset the register map by executing $V_x := x$ for $x \in Q$, but this is not necessary as it is always a permutation.)

restart
...

$\rightsquigarrow$

RESTART
...
procedure RESTART
    for $(y, z) \in Q \times \{x\} \cup \{x\} \times Q$ **do**
        while maybe $y > 0$ **do**
            $y \mapsto z$
    restart

Figure 4: Implementing restarts. As an intermediate step, restarts are replaced by a helper procedure that moves to a new configuration before restarting. Here $x \in Q$ is arbitrary.

To summarise the above, we end up with the following proposition.

Proposition 10. *Let $k \in \mathbb{N}$. If a population program deciding φ with size λ exists, then there is a population machine deciding φ with size $\mathcal{O}(\lambda)$.*

Proof. Recall that the size of a population program is $\lambda = n + L + S$, where n is the number of registers, L the number of instructions, and S the swap-size.

Our conversion has exactly n registers. We create a pointer for each register and each procedure, so the number of pointers is $\mathcal{O}(n + L)$. As the pointer domains of the procedure pointers correspond to the call-sites of the respective procedures, the total size of these domains is $\mathcal{O}(L)$. The total size of the domains of the register pointers corresponds to the swap-size, so it is $\mathcal{O}(S)$. The domains of the three special pointers OF, CF and IP have size $\mathcal{O}(L)$.

To estimate the number of instructions note that all primitives, except for **restart**, expand to a constant number if instructions. For **restart** we need to introduce the helper procedure of length $\Theta(n)$, but this overhead is only incurred once. So in total we end up with $\mathcal{O}(n + L)$ instructions. $\square$

7.3. Conversion to Population Protocols

Let $\mathcal{A} = (Q, F, \mathcal{F}, \mathcal{I})$ denote a population machine. Our goal is to convert $\mathcal{A}$ to a population protocol $PP = (Q^*, \delta, I, O)$.

We will use two types of agents: *register agents* to store the values of the registers, and *pointer agents* to store the pointers. For a register we have many identical agents, and the value of the register corresponds to the total number of those agents. However, for each pointer we use a unique agent, storing the value of the pointer in its state.

The population protocol has no leaders, so we implement a leader election for each pointer. If two agents store the value of a pointer, they eventually meet and one of them is moved to another state. When this happens, the computation is restarted — but note that the values of the registers are not reset. As population machines have no restrictions on the initial register configuration, this poses no issues.

For the population to come to a consensus, we use a standard output broadcast. In a first step, we define a population protocol that merely simulates the machine, without coming to a consensus. Afterwards, we add a single bit to all states. In this bit an agent stores its current opinion. When any agent meets the pointer agent of the output flag OF, the former will assume the opinion of the latter. Eventually, the value of the output flag has stabilised and will propagate throughout the entire population, at which point a consensus has formed.

We now define our conversion formally.

States. The register agents use states Q, while the pointer agent for pointer $X \in F \setminus \{IP\}$ uses states of the form $Q_X := \{X_s^v : v \in \mathcal{F}_X, s \in S_X\}$. Here, $v \in \mathcal{F}_X$ stores the current value of the pointer, while $s \in S_X$ indicates intermediate stages during the execution of an instruction. The possible values of s depend on the type of pointer:

$$\begin{aligned} S_{IP} &:= \{\mathsf{none}, \mathsf{wait}, \mathsf{half}\} && \\ S_X &:= \{\mathsf{none}, \mathsf{done}, \mathsf{emit}, \mathsf{take}, \mathsf{test}, \mathsf{true}, \mathsf{false}\} && \text{if } X = V_x \\ S_X &:= \{\mathsf{none}, \mathsf{done}\} && \text{if } X \neq V_x,\ X \neq IP \end{aligned}$$

Finally, to perform the mappings necessary for instruction of the form $(X := f(Y))$, we add states $Q_{\text{map}} := \{X^i_{\mathsf{map}} : \mathcal{I}_i = (X := f(Y))\}$.

In total, we have states $Q^* := Q \cup \bigcup_{X \in F} Q_X \cup Q_{\text{map}}$.

Initial states and leader election. Let $X_1, ..., X_{|F|}$ denote some enumeration of F with $X_{|F|} = IP$. We set $I := \{X_1\}$, i.e. we use X_1 as unique initial state.

For each pointer X_i, fix an initial value $v_i \in \mathcal{F}_{X_i}$. These initial values must fulfil the requirements of initial configurations set forth in Definition 2, i.e. $v_{|F|} := 1$ (recall $X_{|F|} = IP$), and $v_i := x$ if $X_i = V_x$ for $x \in Q$. To define the transitions, we also fix some arbitrary register $x \in Q$. For convenience, we use $*$ as a wildcard.

$$\begin{aligned} (X_i)^*_*, (X_i)^*_* &\mapsto (X_i)^{v_i}_{\mathsf{none}}, (X_{i+1})^{v_{i+1}}_{\mathsf{none}} \qquad \text{for } i = 1, ..., |F| - 1 \\ IP^*_*, IP^*_* &\mapsto (X_1)^{v_1}_{\mathsf{none}}, x \end{aligned} \qquad \langle\mathsf{elect}\rangle$$

Intuitively, whenever two agents in X_i meet, one of them moves to X_{i+1}, initialising it in the process. The pointer IP is handled slightly differently: here one of the agents moves to x and thus becomes a register agent, while the other moves to X_1. This will then re-initialise $X_1, ..., X_{|F|}$.

Instructions. The transitions for executing an instruction I_i, $i \in \{1, ..., L\}$, depend on the type of instruction. The first case is $I_i = (x \mapsto y)$. This is somewhat involved as we need to first translate x and y using the register map. First (the agent responsible for) IP instructs V_x to move one agent from the register currently assigned to x to some fixed register z. (Note that z is independent of the instruction.) After that is completed, V_y moves the agent from z to its target. Note that $i = L$ means that the machine hangs.

$$\begin{array}{llll}
IP^i_{\mathsf{none}},\ (V_x)^v_* & \mapsto IP^i_{\mathsf{wait}},\ (V_x)^v_{\mathsf{emit}} & \text{for } v \in \mathcal{F}_{V_x} & \\
(V_x)^v_{\mathsf{emit}},\ v & \mapsto (V_x)^v_{\mathsf{done}},\ z & \text{for } v \in \mathcal{F}_{V_x} & \\
IP^i_{\mathsf{wait}},\ (V_x)^v_{\mathsf{done}} & \mapsto IP^i_{\mathsf{half}},\ (V_x)^v_{\mathsf{none}} & \text{for } v \in \mathcal{F}_{V_x} & \\
IP^i_{\mathsf{half}},\ (V_y)^v_* & \mapsto IP^i_{\mathsf{wait}},\ (V_y)^v_{\mathsf{take}} & \text{for } v \in \mathcal{F}_{V_y} & \langle\mathsf{move}\rangle \\
(V_y)^v_{\mathsf{take}},\ z & \mapsto (V_y)^v_{\mathsf{done}},\ v & \text{for } v \in \mathcal{F}_{V_y} & \\
IP^i_{\mathsf{wait}},\ (V_y)^v_{\mathsf{done}} & \mapsto IP^{i+1}_{\mathsf{none}},\ (V_y)^v_{\mathsf{none}} & \text{if } i < L, \text{ for } v \in \mathcal{F}_{V_x} &
\end{array}$$

For $I_i = (\textbf{maybe } x > 0)$ the IP agent again recruits the V_x agent to do the actual operation. The latter either detects x or it does not, and then stores the result in CF.

$$\begin{array}{llll}
IP^i_{\mathsf{none}},\ (V_x)^v_* & \mapsto IP^i_{\mathsf{wait}},\ (V_x)^v_{\mathsf{test}} & \text{for } v \in \mathcal{F}_{V_x} & \\
(V_x)^v_{\mathsf{test}},\ v & \mapsto (V_x)^v_{\mathsf{true}},\ v & \text{for } v \in \mathcal{F}_{V_x} & \\
(V_x)^v_{\mathsf{test}},\ q & \mapsto (V_x)^v_{\mathsf{false}},\ q & \text{for } v \in \mathcal{F}_{V_x},\ q \in Q^* \setminus \{v\} & \langle\mathsf{test}\rangle \\
(V_x)^v_b,\ \ CF^*_* & \mapsto (V_x)^v_{\mathsf{done}},\ CF^b_{\mathsf{none}} & \text{for } v \in \mathcal{F}_{V_x},\ b \in \{\mathsf{true}, \mathsf{false}\} & \\
IP^i_{\mathsf{wait}},\ (V_x)^v_{\mathsf{done}} & \mapsto IP^{i+1}_{\mathsf{none}},\ (V_x)^v_{\mathsf{none}} & \text{if } i < L, \text{ for } v \in \mathcal{F}_{V_x} &
\end{array}$$

The third type, $I_i = (X := f(Y))$, has some special cases. We first assume $Y \neq IP$ wlog, as the value of IP is simply i and $f(Y)$ could be replaced by a constant expression. Both $X = Y$ and $X = IP$ have to be handled separately. The general procedure then is that (the agent responsible for) IP moves X into an intermediate state in Q_{map} and waits. Then, X meets Y, updates its value, and finally signals IP to continue to computation.

We start with the ordinary case $X \notin \{Y, IP\}$.

$$\begin{array}{rlll}
IP^i_{\mathsf{none}}, X^*_* & \mapsto IP^i_{\mathsf{wait}}, X^i_{\mathsf{map}} & \text{if } i < L & \\
X^i_{\mathsf{map}}, Y^v_* & \mapsto X^{f(v)}_{\mathsf{done}}, Y^v_{\mathsf{none}} & \text{for } v \in \mathcal{F}_Y & \langle\mathsf{pointer}\rangle \\
IP^i_{\mathsf{wait}}, X^v_{\mathsf{done}} & \mapsto IP^{i+1}_{\mathsf{none}}, X^v_{\mathsf{none}} & \text{for } v \in \mathcal{F}_X &
\end{array}$$

Now we handle the special cases. These are easier, as only two agents are involved.

$$\begin{array}{llll}
IP^i_{\mathsf{none}}, Y^v_* & \mapsto IP^{f(i)}_{\mathsf{none}}, Y^v_{\mathsf{none}} & \text{if } X = IP, \text{ for } v \in \mathcal{F}_Y & \langle\mathsf{pointer}\rangle \\
IP^i_{\mathsf{none}}, Y^v_* & \mapsto IP^{i+1}_{\mathsf{none}}, Y^{f(v)}_{\mathsf{none}} & \text{if } X = Y,\ i < L, \text{ for } v \in \mathcal{F}_Y &
\end{array}$$

Output broadcast. As mentioned above, we need to ensure that the agents come to a consensus. So we convert PP again, to the final population protocol $PP' = (Q', \delta', I', O')$. This uses the standard broadcast construction, so $Q' := Q^* \times \{\mathsf{true}, \mathsf{false}\}$, $I' := I \times \{\mathsf{false}\}$, $O' := Q' \times \{\mathsf{true}\}$ and for all $q_1, q_2, q_1', q_2' \in Q^*$ with $(q_1, q_2 \mapsto q_1', q_2') \in \delta$ or $(q_1, q_2) = (q_1', q_2')$ we have transitions

$$\begin{array}{rll} (q_1, *), (q_2, *) \mapsto (q_1', b), (q_2', b) & & \text{if } OF_*^b \in \{q_1', q_2'\}, \text{ for a } b \in \{\mathsf{true}, \mathsf{false}\} \\ (q_1, b_1), (q_2, b_2) \mapsto (q_1', b_1), (q_2', b_2) & & \text{otherwise} \end{array}$$

We now summarise the above conversion.

Proposition 11. *If a population machine deciding φ with size n exists, then there is a population protocol deciding $\varphi'(x) \Leftrightarrow \varphi(x - i) \wedge x \geq i$ with $\mathcal{O}(n)$ states, for some $i \leq n$.*

Proof. Let π denote a mapping between configurations of the population machine $\mathcal{A}$ and the population protocol PP resulting from our conversion. A configuration C of $\mathcal{A}$ is mapped to a configuration $\pi(C)$ of PP as follows.

$$\begin{array}{lll} \pi(C)(x) & := C(x) & \text{for } x \in Q \\ \pi(C)(X^v_{\mathsf{none}}) & := 1 & \text{if } C(X) = v, \text{ for } X \in F, v \in \mathcal{F}_X \\ \pi(C)(X^*_*) & := 0 & \text{otherwise} \end{array}$$

If PP is run on a configuration with fewer than $|F|$ agents, no agent can reach a state IP^*_* via $\langle \mathsf{elect} \rangle$, and not other transition is enabled. In particular, it is not possible for any agent to enter OF^{true}_*.

We now assume that at least $|F|$ agents are present, and argue that every fair run of PP reaches a configuration $\pi(C)$. As long as there is an $X \in F$ with at least two agents in states X^*_*, $\langle \mathsf{elect} \rangle$ is enabled. So eventually we enter a configuration where every X^*_* has exactly one agent. At the moment this happens, these agents are in X^v_{none}, where v is the initial state of the pointer. Therefore we have reached a configuration $\pi(C)$; moreover, C must be an initial configuration of $\mathcal{A}$ with $|C| = |\pi(C)| - |F|$ agents.

To see that a run of PP corresponds to one of $\mathcal{A}$, we need only convince ourselves that $\langle \mathsf{move} \rangle$, $\langle \mathsf{test} \rangle$ and $\langle \mathsf{pointer} \rangle$ correctly implement the semantics of Definition 2 and move to a configuration $\pi(C')$, where $C \to C'$.

Every fair run of $\mathcal{A}$ stabilises a $b \in \{\mathsf{true}, \mathsf{false}\}$, according to φ. So eventually there will be a unique agent in OF^b_*, and it will remain in one of these states.

It remains to argue that runs of PP' correspond to runs of PP (and thus to runs of $\mathcal{A}$), and that they stabilise to the correct output. The former is easy to see, as the output broadcast construction simply uses the first component to execute PP (and this is not affected by the second). Once a unique agent remains in OF^b_* in PP, the corresponding run in PP' will have an agent in (OF^b_*, b). Eventually, this agent will convince all other agents that the output is b, and the computation stabilises to b.

As PP (and PP') use $|F|$ agents to store the value of each pointer, the corresponding configurations of $\mathcal{A}$ are smaller, and PP' decides $\varphi'(x) \Leftrightarrow x \geq |F| \wedge \varphi(x - |F|)$.

Finally, we need to count the states of PP'. We have $|Q'| = 2 \cdot |Q^*|$ and

$$|Q^*| = |Q| + \sum_{X \in F} |Q_X| + |Q_{\text{map}}| \leq |Q| + 7 \sum_{X \in F} |\mathcal{F}_X| + L \in \mathcal{O}(n)$$

□

8. Conclusions

We have shown an $\mathcal{O}(\log \log n)$ upper bound on the state-complexity of flock-of-bird predicates for leaderless population protocols, closing the last remaining gap. Our result is based on a new model, population programs, which enable the specification of leaderless population protocols using structured programs.

Flock-of-bird predicates can be considered the most important family for the study of space complexity, as they are (perhaps) the simplest way of encoding a number into the protocol. The precise space complexity of other classes of predicates, however, is still mostly open. The existing results generalise somewhat; the construction presented in this paper, for example, can also be used to decide $\varphi(x) \Leftrightarrow x = k$ for $k \geq 2^{2^n}$ with $\mathcal{O}(n)$ states. As mentioned, there also exist succinct constructions for arbitrary (decidable) predicates, but — to the extent of our knowledge — it is still open whether, for example, $\varphi(x) \Leftrightarrow x = 0 \pmod{k}$ can be decided for $k \geq 2^{2^n}$, both with and without leaders.

(Such remainder predicates have special significance for population protocols, as any predicate decided by a population protocol can be written as a boolean combination of threshold and modulo predicates.)

As defined, our model of population programs can only decide unary predicates and it seems impossible to decide even quite simple remainder predicates (e.g. "is the total number of agents even"). Is this a fundamental limitation, or simply a shortcoming of our specific choices? We tend towards the latter, and hope that other very succinct constructions for leaderless population protocols can make use of a similar approach.

References

[1] Dan Alistarh, James Aspnes, David Eisenstat, Rati Gelashvili, and Ronald L. Rivest. Time-space trade-offs in population protocols. In Philip N. Klein, editor, *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 2560–2579. SIAM, 2017. doi:10.1137/1.9781611974782.169.

[2] Dan Alistarh, James Aspnes, and Rati Gelashvili. Space-optimal majority in population protocols. In Artur Czumaj, editor, *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*, pages 2221–2239. SIAM, 2018. doi:10.1137/1.9781611975031.144.

[3] Dan Alistarh, Rati Gelashvili, and Milan Vojnovic. Fast and exact majority in population protocols. In Chryssis Georgiou and Paul G. Spirakis, editors, *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, PODC 2015, Donostia-San Sebastián, Spain, July 21 - 23, 2015*, pages 47–56. ACM, 2015. doi:10.1145/2767386.2767429.

[4] Dana Angluin, James Aspnes, Zoë Diamadi, Michael J. Fischer, and René Peralta. Computation in networks of passively mobile finite-state sensors. In *PODC*, pages 290–299. ACM, 2004.

[5] Dana Angluin, James Aspnes, Zoë Diamadi, Michael J. Fischer, and René Peralta. Computation in networks of passively mobile finite-state sensors. *Distributed Comput.*, 18(4):235–253, 2006.

[6] Dana Angluin, James Aspnes, and David Eisenstat. Fast computation by population protocols with a leader. *Distributed Comput.*, 21(3):183–199, 2008.

[7] Dana Angluin, James Aspnes, David Eisenstat, and Eric Ruppert. The computational power of population protocols. *Distributed Comput.*, 20(4):279–304, 2007.

[8] Dana Angluin, James Aspnes, Michael J. Fischer, and Hong Jiang. Self-stabilizing population protocols. In James H. Anderson, Giuseppe Prencipe, and Roger Wattenhofer, editors, *Principles of Distributed Systems, 9th International Conference, OPODIS 2005, Pisa, Italy, December 12-14, 2005, Revised Selected Papers*, volume 3974 of *Lecture Notes in Computer Science*, pages 103–117. Springer, 2005. doi:10.1007/11795490_10.

[9] Stav Ben-Nun, Tsvi Kopelowitz, Matan Kraus, and Ely Porat. An $O(\log^{3/2} n)$ parallel time population protocol for majority with $O(\log n)$ states. In Yuval Emek and Christian Cachin, editors, *PODC '20: ACM Symposium on Principles of Distributed Computing, Virtual Event, Italy, August 3-7, 2020*, pages 191–199. ACM, 2020. doi:10.1145/3382734.3405747.

[10] Petra Berenbrink, Robert Elsässer, Tom Friedetzky, Dominik Kaaser, Peter Kling, and Tomasz Radzik. A population protocol for exact majority with o(log5/3 n) stabilization time and theta(log n) states. In Ulrich Schmid and Josef Widder, editors, *32nd International Symposium on Distributed Computing, DISC 2018, New Orleans, LA, USA, October 15-19, 2018*, volume 121 of *LIPIcs*, pages 10:1–10:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPIcs.DISC.2018.10.

[11] Petra Berenbrink, Robert Elsässer, Tom Friedetzky, Dominik Kaaser, Peter Kling, and Tomasz Radzik. Time-space trade-offs in population protocols for the majority problem. *Distributed Comput.*, 34(2):91–111, 2021. doi:10.1007/s00446-020-00385-0.

[12] Andreas Bilke, Colin Cooper, Robert Elsässer, and Tomasz Radzik. Population protocols for leader election and exact majority with o(log^2 n) states and o(log^2 n) convergence time. *CoRR*, abs/1705.01146, 2017. URL: http://arxiv.org/abs/1705.01146, arXiv:1705.01146.

[13] Michael Blondin, Javier Esparza, Blaise Genest, Martin Helfrich, and Stefan Jaax. Succinct population protocols for Presburger arithmetic. In *STACS*, volume 154 of *LIPIcs*, pages 40:1–40:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[14] Michael Blondin, Javier Esparza, and Stefan Jaax. Large flocks of small birds: On the minimal size of population protocols. In *STACS*, volume 96 of *LIPIcs*, pages 16:1–16:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

[15] Janna Burman, Ho-Lin Chen, Hsueh-Ping Chen, David Doty, Thomas Nowak, Eric E. Severson, and Chuan Xu. Time-optimal self-stabilizing leader election in population protocols. In Avery Miller, Keren Censor-Hillel, and Janne H. Korhonen, editors, *PODC '21: ACM Symposium on Principles of Distributed Computing, Virtual Event, Italy, July 26-30, 2021*, pages 33–44. ACM, 2021. doi:10.1145/3465084.3467898.

[16] Shukai Cai, Taisuke Izumi, and Koichi Wada. How to prove impossibility under global fairness: On space complexity of self-stabilizing leader election on a population protocol model. *Theory Comput. Syst.*, 50(3):433–445, 2012. doi:10.1007/s00224-011-9313-z.

[17] Ho-Lin Chen, Rachel Cummings, David Doty, and David Soloveichik. Speed faults in computation by chemical reaction networks. *Distributed Comput.*, 30(5):373–390, 2017. doi:10.1007/s00446-015-0255-6.

[18] Philipp Czerner and Javier Esparza. Lower bounds on the state complexity of population protocols. In Avery Miller, Keren Censor-Hillel, and Janne H. Korhonen, editors, *PODC '21: ACM Symposium on Principles of Distributed Computing, Virtual Event, Italy, July 26-30, 2021*, pages 45–54. ACM, 2021. doi:10.1145/3465084.3467912.

[19] Philipp Czerner, Roland Guttenberg, Martin Helfrich, and Javier Esparza. Fast and succinct population protocols for presburger arithmetic. *CoRR*, abs/2202.11601, 2022. arXiv:2202.11601.

[20] David Doty, Mahsa Eftekhari, Leszek Gasieniec, Eric E. Severson, Przemyslaw Uznanski, and Grzegorz Stachowiak. A time and space optimal stable population protocol solving exact majority. In *62nd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2021, Denver, CO, USA, February 7-10, 2022*, pages 1044–1055. IEEE, 2021. doi:10.1109/FOCS52979.2021.00104.

[21] Moez Draief and Milan Vojnovic. Convergence speed of binary interval consensus. *SIAM J. Control. Optim.*, 50(3):1087–1109, 2012. doi:10.1137/110823018.

[22] Alexander Kozachinskiy. New bounds for the flock-of-birds problem. *CoRR*, abs/2110.12452, 2021. arXiv:2110.12452.

[23] Jérôme Leroux. State complexity of counting population protocols with leaders. *CoRR*, abs/2109.15171, 2021. arXiv:2109.15171.

[24] Richard J. Lipton. The reachability problem requires exponential space. Technical report, Yale University, Dept. of CS, 1976. URL: http://www.cs.yale.edu/publications/techreports/tr63.pdf.

[25] Yves Mocquard, Emmanuelle Anceaume, James Aspnes, Yann Busnel, and Bruno Sericola. Counting with population protocols. In D. R. Avresky and Yann Busnel, editors, *14th IEEE International Symposium on Network Computing and Applications, NCA 2015, Cambridge, MA, USA, September 28-30, 2015*, pages 35–42. IEEE Computer Society, 2015. doi:10.1109/NCA.2015.35.

[26] Yves Mocquard, Emmanuelle Anceaume, and Bruno Sericola. Optimal proportion computation with population protocols. In Alessandro Pellegrini, Aris Gkoulalas-Divanis, Pierangelo di Sanzo, and Dimiter R. Avresky, editors, *15th IEEE International Symposium on Network Computing and Applications, NCA 2016, Cambridge, Boston, MA, USA, October 31 - November 2, 2016*, pages 216–223. IEEE Computer Society, 2016. doi:10.1109/NCA.2016.7778621.

[27] Etienne Perron, Dinkar Vasudevan, and Milan Vojnovic. Using three states for binary consensus on complete graphs. In *INFOCOM 2009. 28th IEEE International Conference on Computer Communications, Joint Conference of the IEEE Computer and Communications Societies, 19-25 April 2009, Rio de Janeiro, Brazil*, pages 2527–2535. IEEE, 2009. doi:10.1109/INFCOM.2009.5062181.

A. Robustness of Flock-of-bird Protocols

A major motivation behind the construction of succinct protocols for flock-of-bird predicates is the application to chemical reactions. In this, as in other environments, computations must be able to deal with errors. While not the focus of this paper, we want to briefly discuss how our construction, by virtue of dealing with adversarial initialisation, is robust against some (but not all) sources of noise.

Related research has considered *self-stabilising* population protocols [8, 16, 15]. Such a protocol must converge to a desired output regardless of the input configuration. This matches exactly the guarantees provided by our population programs; however, in the process of converting to population protocols the property is lost. Indeed, it is impossible for a self-stabilising population protocol (with a constant number of states) to decide $\varphi(x) \Leftrightarrow x \geq k$ for any $k > 1$, as any stably accepting configuration must still be stably accepting if one agent is removed.

However, we can consider a weaker notion of noise, where an adversary may add or remove agents during the computation. To the extend of our knowledge, our protocol is the first to be robust against addition of agents. In particular, all prior known constructions are 1-aware, and thus can be made to accept by adding a single agent to the 1-aware state.

Our construction is not robust against removal of agents, even if the removal does not cause the number of agents to drop below the threshold. Intuitively, it is possible to remove the agent responsible for storing the instruction pointer (see Section 7.3) and the computation stops before the output is determined.

This limitation is shared by all prior constructions except for one (discussed below). Intuitively, the constructions of [4, 13, 19] collect the value of multiple agents into a single one. Eliminating the latter may then change the output.

There is a known construction that can handle removal of agents. It uses states $\{0, ..., k\}$ and transitions

$$i, i \mapsto i+1, i \quad \text{for } i < k \qquad\qquad k, i \mapsto k, k \quad \text{for } i < k$$

However, this protocol cannot handle addition of agents (a single state-k will cause it to accept) and it not succinct.