Breaking Through the $\Omega(n)$-Space Barrier: Population Protocols Decide Double-Exponential Thresholds

Philipp Czerner
Department of Informatics, TU München, Germany

Abstract

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. A family of protocols deciding predicates φ_n is *succinct* if it uses $\mathcal{O}(|\varphi_n|)$ states, where φ_n is encoded as quantifier-free Presburger formula with coefficients in binary. (All predicates decidable by population protocols can be encoded in this manner.) While it is known that succinct protocols exist for all predicates, it is open whether protocols with $o(|\varphi_n|)$ states exist for *any* family of predicates φ_n. We answer this affirmatively, by constructing protocols with $\mathcal{O}(\log|\varphi_n|)$ states for some family of threshold predicates $\varphi_n(x) \Leftrightarrow x \geq k_n$, with $k_1, k_2, ... \in \mathbb{N}$. (In other words, protocols with $\mathcal{O}(n)$ states that decide $x \geq k$ for a $k \geq 2^{2^n}$.) This matches a known lower bound. Moreover, our construction for threshold predicates is the first that is not 1-aware, and it is almost self-stabilising.

2012 ACM Subject Classification Theory of computation → Distributed computing models

Keywords and phrases Distributed computing, population protocols, state complexity

Digital Object Identifier 10.4230/LIPIcs.DISC.2024.17

Related Version *Full Version*: `https://arxiv.org/abs/2204.02115` [18]

Funding *Philipp Czerner*: This work was supported by an ERC Advanced Grant (787367: PaVeS) and by the Research Training Network of the Deutsche Forschungsgemeinschaft (DFG) (378803395: ConVeY).

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. [28, 22]). 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 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.

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.

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, it is known that leaders enable an exponential speed-up [6, 1] in terms of the time that is needed to come to a consensus.

Space complexity. Many constructions in the literature need a large number of states. We estimate, for example, that the protocols of [6] need 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.

This motivates the study of *space complexity*, the minimal number of states necessary for a population protocol to decide a given predicate. Predicates are usually encoded as quantifier-free Presburger formulae with coefficients in binary. For example, the predicates $\varphi_n(x) \Leftrightarrow x \geq 2^n$ have length $|\varphi_n| \in \Theta(n)$. Formally we define space(φ) as the smallest number of states of any protocol deciding φ, and $\text{space}_L(\varphi)$ as the analogous function for protocols with a leader. Clearly, $\text{space}(\varphi)_L \leq \text{space}(\varphi)$.

The original construction in [4] showed $\text{space}(\varphi) \in \mathcal{O}(2^{|\varphi|})$ – impractically large. For the family of *threshold predicates* $\tau_n(x) \Leftrightarrow x \geq n$ Blondin, Esparza and Jaax [14] prove $\text{space}(\tau_n) \in \mathcal{O}(|\tau_n|)$, i.e. they have polynomial space complexity. For several years it was open whether similarly succinct protocols exist for every predicate. This was answered positively in [13], showing $\text{space}(\varphi) \in \mathcal{O}(\text{poly}(|\varphi|))$ for all φ.

Is it possible to do much better? For most predicates it is not; based on a simple counting argument one can show that for every family φ_n with $|\varphi_n| \in \mathcal{O}(n)$ there is an infinite subfamily $(\varphi'_n)_n \subseteq (\varphi_n)_n$ with $\text{space}_L(\varphi'_n) \in \Omega(|\varphi_n|^{1/4-\varepsilon})$, for any $\varepsilon > 0$ [14].

This covers threshold predicates and many other natural families of protocols (e.g. $\varphi_n(x) \Leftrightarrow x \equiv 0 \pmod n$ or $\varphi_n(x, y) \Leftrightarrow x \geq ny$). But it is not an impenetrable barrier, even for the case of threshold protocols: it does not rule out constructions that work for *infinitely many* (but not all) thresholds and use only, say, logarithmically many states. Indeed, if leaders are allowed this is known to be possible: [14] shows $\text{space}_L(\tau'_n) \in \mathcal{O}(\log|\tau'_n|)$ for some subfamily τ'_n of threshold predicates.

Recently, *general* lower bounds have been obtained, showing $\text{space}(\tau_n) \in \Omega(\log^{1-\varepsilon}|\tau_n|)$ for all $\varepsilon > 0$ [19, 20]. The same bound (up to $\varepsilon = 1/2$) holds even if the model is extended with leaders [24].

For leaderless population protocols, these results leave an exponential gap. In this paper we settle that question and show that, contrary to prevailing opinion, $\text{space}(\tau'_n) \in \mathcal{O}(\log|\tau'_n|)$ for some subfamily τ'_n of threshold predicates. In other words, we construct the first family of *leaderless* population protocols that decide double-exponential thresholds and break through the polynomial barrier.

Robustness. Since population protocols model computations where large numbers of agents interact, it is desirable that protocols deal robustly with noise. In a chemical reaction, for example, there can be trace amounts of unwanted molecules. So the initial configuration of the protocol would have the form $C_I + C_N$, where C_I is the "intended" initial configuration, containing only agents in the designated initial states, and C_N is a "noise" configuration, which can contain agents in arbitrary states.

Table 1 Prior results on the state complexity of threshold predicates $\varphi(x) \Leftrightarrow x \geq k$, for $k \in \mathbb{N}$. Upper bounds need only hold for infinitely many k. We elide exponentially dominated factors from lower bounds.

year	result	type	ordinary	with leaders
2018	Blondin, Esparza, Jaax [14]	construction	$\mathcal{O}(\|\varphi\|)$	$\mathcal{O}(\log\|\varphi\|)$
2021	Czerner, Esparza [19]	impossibility	$\Omega(\log\log\|\varphi\|)$	$\Omega(\text{ack}^{-1}\|\varphi\|)$
2021	Czerner, Esparza, Leroux [20]	impossibility	$\Omega(\log\|\varphi\|)$	
2022	Leroux [24]	impossibility		$\Omega(\log\|\varphi\|)$
2024	this paper	construction	$\mathcal{O}(\log\|\varphi\|)$	

For threshold predicates, specifically, we want to decide whether $|C_I| + |C_N|$ exceeds some threshold $k \in \mathbb{N}$, under some reasonable restrictions to C_I, C_N. However, all known threshold protocols fail even for the case $|C_N| = 1$. Is it possible to do better?

If C_N can be chosen arbitrarily, then the protocol has to work correctly for *all* input configurations. This property is known as *self-stabilisation*, and it has also been investigated in the context of population protocols [8, 16, 15]. However, it can only be achieved in extensions of the model (e.g. on specific communication graphs, or with a non-constant number of states). This is easy to see in the case of threshold predicates: if any configuration is stably accepting, then any smaller configuration is stably accepting as well. In particular, there is a stably accepting configuration with $k - 1$ agents.

While full self-stabilisation is impossible, in this paper we show that one can come remarkably close. We prove that our construction is *almost self-stabilising*, meaning that it computes the correct output for all C_I, C_N with $|C_I| \geq n$, where n is the number of states of the protocol. We do not constraint C_N at all. Since $n \in \mathcal{O}(\log\log k)$ in our protocol, this means that one can take an arbitrary configuration C_N one wishes to count, add a tiny amount of agents to the initial state, and the protocol will compute the correct output.

Related work. We consider the space complexity of families of protocols, each of which decides a different predicate. In another line of research, one considers a family of protocols for the *same* predicate, where each protocol is specialised for a fixed population size m.

In the original model of population protocols (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 avenue of research; here, the number of states depends on m (e.g. the protocol has $\mathcal{O}(\log m)$ states, or even $\mathcal{O}(\log\log m)$ states). In this model, faster protocols can be achieved [3, 26, 27].

It has also led to space-efficient, fast protocols, which stabilise within $\mathcal{O}(\text{polylog}\, m)$ parallel time, using a state-space that grows only slowly with the number of agents, e.g. $\mathcal{O}(\text{polylog}\, m)$ states [1, 12, 2, 10, 9, 11, 21]. 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].

2 Main result

We construct population protocols (without leaders) for an infinite family of threshold predicates $\varphi_n(x) \Leftrightarrow x \geq k_n$, with $k_1, \ldots \in \mathbb{N}$, proving an $\mathcal{O}(\log|\varphi_n|)$ upper bound on their state complexity. This closes the final gap in the state complexity of threshold predicates.

As in prior work, our result is not a construction for arbitrary thresholds k, only for an infinite family of them. It is, therefore, 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 $k \geq 2^{2^n}$.*

Proof. This will follow from theorems 3 and 5. ◀

The result is surprising, as prevailing opinion was that the existing constructions are optimal. This was based on the following:

- It is intuitive that population protocols with leaders have an advantage. In particular, one can draw a parallel to time complexity, where an exponential gap is proven: for some predicates protocols with leaders have $\mathcal{O}(\operatorname{polylog} m)$ parallel time, while all leaderless protocols have $\Omega(m)$ parallel time.
- The $\mathcal{O}(\log \log k)$-state construction from [14] crucially depends on having leaders.
- The technique to show the $\Omega(\log \log k)$ lower bound could, for the most part, also be used for a $\Omega(\log k)$ bound. Only the use of Rackoff's theorem, a general result for Petri nets, does not extend.
- There is a conditional impossibility result, showing that $\Omega(\log k)$ states are necessary for leaderless 1-aware protocols. [14] (Essentially, protocols where some agent knows at some point that the threshold has been exceeded.) All prior constructions are 1-aware.

Regarding the last point, our protocol evades the mentioned conditional impossibility result by being the first construction that is not 1-aware. Intuitively, our protocol only accepts provisionally and continues to check that no invariant has been violated. Based on this, we also obtain the following robustness guarantee:

▶ **Theorem 2.** *The protocols of Theorem 1 are almost self-stabilising.*

Overview. We build on the technique of Lipton [25], which describes a double-exponential counting routine in vector addition systems. Implementing this technique requires the use of procedure calls; our first contribution are *population programs*, a model in which population protocols can be constructed by writing structured programs, in Section 4. Every such program can be converted into an equivalent population protocol.

However, population programs provide weaker guarantees than the model of parallel programs used in [25]. Both models access registers with values in $\mathbb{N}$. In a parallel program these are initialised to 0, while in a population program *all* registers start with arbitrary values. This limitation is essential for our conversion into population protocols.

A straightforward implementation is, therefore, impossible. Instead, we have to adapt the technique to work with arbitrary initial configurations. Our second contribution, and the main technical difficulty of this result, is extending the original technique with error-checking routines to work in our model. We use a detect-restart loop, which determines whether the initial configuration is "bad" and, if so, restarts with a new initial configuration. The stochastic behaviour of population protocols ensures that a "good" initial configuration is reached eventually. Standard techniques could be used to avoid restarts with high probability and achieve an optimal running time, but this is beyond the scope of this paper.

A high level overview of both the original technique as well as our error-checking strategy is given in Section 5. We then give a detailed description of our construction in Section 6.

To get population protocols, we need to convert from population programs. We split this into two parts. First, we use standard techniques to lower population programs to *population machines*, an assembly-like programming language. In a second step we simulate arbitrary population machines by population protocols. This conversion is described in Section 7.

Finally, we introduce the notion of being almost self-stabilising in Section 8, and prove that our construction has this property.

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 write $C(S) := \sum_{q \in S} C(q)$ to denote the total number of elements in some $S \subseteq Q$, and set $|C| := C(Q)$. 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 $r \neq q$.

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

A sequence $\tau = (C_i)_{i \in \mathbb{N}}$ with $C_i \in \mathcal{C}$ is a *run* if $C_i \rightarrow C_{i+1}$ for all $i \in \mathbb{N}$. We say that τ *stabilises to* b, for $b \in \{\mathsf{true}, \mathsf{false}\}$, if there is an i s.t. C_j has output b for every $j \geq i$. A run τ is *fair* if $\cap_{i \geq 0}\{C_i, C_{i+1}, \ldots\}$ is closed under $\rightarrow$, i.e. every configuration that *can* be reached infinitely often *is*.

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

Let $\varphi : \mathbb{N}^I \rightarrow \{\mathsf{true}, \mathsf{false}\}$ 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 $\varphi(C)$, where fair run and stabilisation are defined as above.

4 Population Programs

We introduce population programs, which allows us to specify population protocols using structured programs. An example is shown in Figure 1.

Formally, a *population program* is a tuple $\mathcal{P} = (Q, \mathrm{Proc})$, 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.

```
procedure Main
  OF := false
  while ¬Test(4) do
    Clean
  OF := true
  while ¬Test(7) do
    Clean
  OF := false
  while true do
    Clean
```

```
procedure Test(i)
  for j = 1, ..., i do
    if detect x > 0 then
      x ↦ y
    else
      return false
  return true
```

```
procedure Clean
  if detect z > 0 then
    restart
  swap x, y
  while detect y > 0 do
    y ↦ x
```

Figure 1 A population program for $\varphi(x) \Leftrightarrow 4 \leq x < 7$ using registers x, y, z. Main is run initially and decides the predicate, Test(i) tries to move i units from x to y and reports whether it succeeded, and Clean checks whether z is empty and moves some number of units from y to x. If Clean detects an agent in z, it restarts the computation. As every run calls Clean infinitely often, this serves to reject initial configurations where z is nonzero; eventually the protocol will be restarted with $z = 0$. This is an illustrative example and some simplifications are possible. E.g. the instruction (**swap** x, y) in Clean is superfluous; additionally, instead of checking $z > 0$ one could omit that register entirely.

Primitives. Each register $x \in Q$ can take values in $\mathbb{N}$. 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 nondeterministic nonzero-check (**detect** $x > 0$), for $x \in Q$, 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 nonzero the check cannot return false infinitely often.
- A swap (**swap** x, y) exchanges the values of the two registers x, y. This primitive is not necessary, but it simplifies the implementation.

Loops and branches. Population programs use while-loops and if-statements, which function as one would expect.

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 has procedure calls, but no recursion. Procedures have no arguments, but we may have parameterised 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 one could inline every procedure call. The main reason to make use of procedures at all is succinctness: if our program contains too many instructions, the resulting population protocol has too many states.

Procedures may 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 OF, 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.

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.

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. [1] For example, in Figure 1 the swap-size is two: $(x, y), (y, x)$ can be swapped, but e.g. (x, z) cannot. If we add a (**swap** y, z) instruction at any point, then (x, z) can be swapped (transitively), and the swap-size would be 6.

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 $\sigma \in (\mathrm{Proc} \times \mathbb{N})^*$ is the call stack, storing names and currently executed instructions of called procedures. (E.g. $\sigma = ((\mathsf{Main}, 3), (\mathsf{Test}(4), 1))$ when Test is first called in Figure 1.) A configuration is *initial* if $\sigma = ((\mathsf{Main}, 1))$ 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 general notion 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 $C, f \to$ **restart**. If f may hang or not terminate, we write $C, f \to \bot$. Finally, we define $\mathrm{post}(C, f) := \{S : C, f \to S\}$.

5 High-level Overview

We give an intuitive explanation of our construction. This section has two parts. As mentioned, we use the technique of Lipton [25] to count to 2^{2^n} using $4n$ registers. We will give a brief explanation of the original technique in Section 5.1. Readers might also find the restatement of Liptons proof in [23] instructive – the Petri net programs introduced therein are closer to our approach, and more similar to models used in the recent Petri net literature.

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). 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.

[1] 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.

5.1 Double-exponential counting

The biggest limitation of population programs is their inability to detect absence of agents. This is reflected in the (**detect** $x > 0$) 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. In particular, it is impossible to implement a zero-check.

However, Lipton observes that 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. Lipton solves this by implementing 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 $\overline{z}$).

Lipton iterates this construction n times. 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 registers s.t. the necessary invariants hold. For our purposes, having a separate initialisation step is superfluous. Instead, we check whether the invariants hold in the initial configuration and restart (nondeterministically choosing a new initial configuration) if they do not.

5.2 Error detection

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" described above is still guaranteed to terminate, as either $x > 0$ or $\overline{x} \geq k$ must hold. However, it might falsely return $x = 0$ when it is not. The procedure we use above, to combine two k-bounded counter to simulate a k^2-bounded counter, 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 $k' \in \mathcal{O}(k)$.

If the invariant is violated in the other direction, i.e. $x + \overline{x} < k$ holds, 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. For the $x + \overline{x} > k$ case the crucial insight is that we cannot *detect* it, but 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. In the full construction, we have many levels of registers that rely on each other. Instead, we first consider a simplified model here 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 sum of all registers. For each possible value of m we pick one initial configuration C_m and design our procedure s.t.

- every initial configuration different from C_m will cause a restart, and
- if started on C_m it is *possible* that the procedure enters a state where it 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 $i \in \{0, ..., n\}$.
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 of 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 $\textsc{Check}$ terminate and they do not change the values of any register. The second is the simpler one to deal with: later, we will have multiple registers per level and our procedures only need to move agents between registers of the same level. This keeps the sum of registers of one level constant, this weaker property suffices for correctness.

Ensuring that all calls terminate 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 $x_i < N_i$.

However, we note 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 $\textsc{Check}$ by calling itself recursively. In this manner, we can implement $\textsc{Check}$ 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

In this section, we construct a population program $\mathcal{P} = (Q, \mathrm{Proc})$ to prove the following:

▶ **Theorem 3.** *Let $n \in \mathbb{N}$. There exists a population program deciding $\varphi(x) \Leftrightarrow x \geq k$ with size $\mathcal{O}(n)$, for some $k \geq 2^{2^{n-1}}$.*

Full proofs and formal definitions of this section can be found in the full version of the paper [18].

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.

Types of Configurations. As explained in the previous section, x and $\overline{x}$ are supposed to sum to a constant N_i, for a level i register $x \in \{x_i, y_i\}$, which we define via $N_1 := 1$ and $N_{i+1} := (N_i + 1)^2$. If this invariant holds, we can use $x, \overline{x}$ to simulate a N_i-bounded register, which has value x.

We cannot guarantee that this invariant always holds, so our program must deal with configurations that deviate from this. For this purpose, we classify configurations based on which registers fulfil the invariant, and based on the type of deviation.

A configuration $C \in \mathbb{N}^Q$ is *i-proper*, if the invariant holds on levels $1, ..., i$, and their simulated registers have value 0. This is a precondition for most routines. Sometimes we relax the latter requirement on the level i registers; C is *weakly i-proper* if it is $(i-1)$-proper and the invariant holds on level i.

If C is $(i-1)$-proper and not i-proper, then there are essentially two possibilities. Either $C \leq C'$ for some i-proper C' and we call C *i-low*, or $C(x) \geq C'$ for a weakly i-proper C' and we call C *i-high*. Note that it is possible that C is neither i-low nor i-high – these configurations are easy to exclude and play only a minor role. We can mostly ensure that i-low configurations do not occur, but procedures must provide guarantees when run on i-high configurations.

Finally, we say that C is *i-empty* if all registers on levels $i, ..., n+1$ are empty.

	x_1	$\overline{x}_1$	y_1	$\overline{y}_1$	...	x_{i-1}	$\overline{x}_{i-1}$	y_{i-1}	$\overline{y}_{i-1}$	x_i	$\overline{x}_i$	y_i	$\overline{y}_i$	...
i-proper	0	N_1	0	N_1	...	0	N_{i-1}	0	N_{i-1}	0	N_i	0	N_i	...
weakly i-proper	0	N_1	0	N_1	...	0	N_{i-1}	0	N_{i-1}	3	$N_i - 3$	$N_i - 7$	7	...
i-low	0	N_1	0	N_1	...	0	N_{i-1}	0	N_{i-1}	0	$N_i - 3$	0	N_i	...
i-high	0	N_1	0	N_1	...	0	N_{i-1}	0	N_{i-1}	3	N_i	7	$N_i - 5$	...
i-empty	2	4	8	3	...	5	3	0	7	0	0	0	0	...

Figure 2 Example configurations exhibiting the different types.

Summary. We use the following procedures.

- Main. Computation starts by executing this procedure, and Main ultimately decides the predicate $\varphi(x) \Leftrightarrow x \geq 2\sum_{i=1}^{n} N_i$.

Algorithm AssertEmpty.

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

procedure AssertEmpty.(i) $[i \leq n]$
  AssertEmpty$(i+1)$
  for $x \in Q_i$ **do**
    if detect $x > 0$ **then**
      restart
procedure AssertEmpty.(i) $[i = n+1]$
  if detect $\mathsf{R} > 0$ **then**
    restart

Algorithm AssertProper.

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

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

Algorithm Zero Check whether a register is equal to 0.

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

procedure Zero(x)
  while true **do**
    AssertProper$(i-1)$
    if detect $x > 0$ **then**
      return false
    if Large$(\overline{x})$ **then**
      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 1

procedure IncrPair(x, y)
  if Zero$(\overline{y})$ **then**
    swap $y, \overline{y}$
    if Zero$(\overline{x})$ **then**
      swap $x, \overline{x}$
    else $\overline{x} \mapsto x$
  else $\overline{y} \mapsto y$

- AssertEmpty.. Check whether a configuration is i-empty and initiate a restart if not.
- AssertProper.. Check whether a configuration is i-proper or i-low, initiate a restart if not.
- Large. Nondeterministically check whether a register $x \in Q_i$ is at least N_i.
- Zero. Perform a deterministic zero-check on a register $x \in Q_i$.
- IncrPair. As described in Section 5.1, we use two level i registers (which are N_i bounded) to simulate an N_{i+1}-bounded register. This procedure implements the increment operation for the simulated register.

Procedures AssertEmpty., AssertProper.. The procedure AssertEmpty. is supposed to determine whether a configuration is i-empty, which can easily be done by checking whether the relevant registers are nonempty.

Similarly, AssertProper. 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 AssertProper.(0) have no effect and can simply be omitted.

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

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

procedure Large(x) [for $i = 1$]
    if detect $x > 0$ **then**
        $x \mapsto \overline{x}$
        swap $x, \overline{x}$
        return true
    else
        return false

procedure Large(x) [for $i > 1$]
    if $\neg$Zero(x_{i-1}) $\vee$ $\neg$Zero(y_{i-1}) **then**
        restart
    while true **do**
        CheckProper($i - 2$)
        if detect $x > 0$ **then**
            $x \mapsto \overline{x}$
            IncrPair(x_{i-1}, y_{i-1})
            if Zero(x_{i-1}) $\wedge$ Zero(y_{i-1}) **then**
                swap $x, \overline{x}$
                return true
        else
            if Zero(x_{i-1}) $\wedge$ Zero(y_{i-1}) **then**
                return false
            if detect $\overline{x} > 0$ **then**
                $\overline{x} \mapsto x$
                IncrPair($\overline{x}_{i-1}, \overline{y}_{i-1}$)

Procedure 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 $\overline{y}$.

As we show later, IncrPair is "reversible" under only the weak assumption that the configuration $C \in \mathbb{N}^Q$ is i-high. More precisely, C, IncrPair$(x, y) \to C'$ implies C', IncrPair$(\overline{x}, \overline{y}) \to C$. Using this, we can show that Large, which calls IncrPair in a loop, terminates.

Procedure 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 termination, we use 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 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.

As mentioned above, IncrPair is reversible even under weak assumptions. This ensures that the random walk terminates, as it can always retrace its prior steps to go back to its starting point.

Procedure 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. The following lemma formalises this.

▶ **Lemma 4.** Main, *run on register configuration* $C \in \mathbb{N}^Q$, *can only restart or stabilise, and*
(a) *it may stabilise to* false *if* C *is* j*-low and* $(j+1)$*-empty, for some* $j \in \{1, ..., n\}$,
(b) *it may stabilise to* true *if* C *is* n*-proper, and*
(c) *it always restarts otherwise.*

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
            AssertProper(i)
            AssertEmpty(i + 1)
    OF := true
    while true do
        AssertProper(n)
```

7 Converting Population Programs into Protocols

In the previous section we constructed succinct population programs for the threshold predicate. We 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. This results in the following theorem:

▶ **Theorem 5.** *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 $i \in \mathcal{O}(n)$.*

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 finally described in Section 7.3. Here, we only highlight the key ideas of the conversion. The details can be found in the full version of the paper [18].

7.1 Formal Model

▶ **Definition 6.** *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 nonempty 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 $x \in \mathcal{F}_{V_x} \subseteq Q$. The size of $\mathcal{A}$ is $|Q| + |F| + \sum_{X \in F} |\mathcal{F}_X| + |\mathcal{I}|$.*

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

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 pointer 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 $(\textbf{detect}\ x > 0)$ 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 instructions do not operate on the actual registers x, y, but on the registers pointed to by V_x and V_y.) 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.

A precise definition of the semantics can be found in the full version of the paper [18].

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. We show only an example here, a detailed description of the conversion can be found in the full version of the paper [18].

procedure Main
  while detect $x > 0$ **do**
    $x \mapsto y$
  swap x, y

$\rightsquigarrow$

1: **detect** $x > 0$
2: $IP := \begin{cases} 5 & \text{if } CF \\ 3 & \text{else} \end{cases}$
3: $x \mapsto y$
4: $IP := 1$
5: $V_\square := V_x$
6: $V_x := V_y$
7: $V_y := V_\square$

Figure 3 Conversion to a population machine.

Control-flow, i.e. **if**, **while** and procedure calls are implemented via direct assignment to IP, the instruction pointer, as in lines 2 and 4 above. The statements (**detect** $x > 0$) and $(x \mapsto y)$ are translated one-to-one, but note that in the population machine their operands are first translated via the register map. For example, (**detect** $x > 0$) in line 1 checks whether the register pointed to by V_x is nonzero. Correspondingly, **swap** statements result in direct modifications to the register map: lines 5-7 swap the pointers V_x and V_y (and leave the registers they point to unchanged).

7.3 Conversion to Population Protocols

In this section, we only present a simplified version of our construction. In particular, we make use of multiway transitions to have more than two agents interact at a time. Our actual construction, described in the full version of the paper [18], avoids them and the associated overhead.

Let $\mathcal{A} = (Q, F, \mathcal{F}, \mathcal{I})$ denote a population machine. To convert this into a population protocol, we 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. They use states Q. For each pointer we use a unique agent, storing the value of the pointer in its state; they use states $\{X^v : X \in F, v \in \mathcal{F}_X\}$.

Let $X_1, ..., X_{|F|}$ denote some enumeration of F with $X_{|F|} = IP$, and let v_i denote the initial value of X_i. We use X_1 as initial state of the protocol. To goal is to have a unique agent for each pointer, so we implement a simple leader election. We use $*$ as wildcard.

$$X_i^*, X_i^* \mapsto X_i^{v_i}, X_{i+1}^{v_{i+1}} \qquad IP^*, IP^* \mapsto X_1^{v_1}, x$$

with $i \in \{1, ..., |F| - 1\}$. If two agents store the value of a single 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. Eventually, the protocol will thus reach a configuration with exactly one agent in $X_i^{v_i}$, for each i, and the remaining agents in Q.

Starting from this configuration, the instructions can be executed. We illustrate the mapping from instructions to transitions in the following example:

$$
\begin{array}{l}
1\text{:}\;\; x \mapsto y \\
2\text{:}\;\; \textbf{detect } x > 0 \\
3\text{:}\;\; IP := \begin{cases} 1 & \text{if } CF \\ 4 & \text{else} \end{cases} \\
4\text{:}\;\; OF := \neg CF
\end{array}
\quad \rightsquigarrow \quad
\begin{array}{lll}
IP^1, V_x^v, V_y^w, v & \mapsto IP^2, V_x^v, V_y^w, w & \text{for } v, w \in Q \\
IP^2, CF^*, V_x^v, v & \mapsto IP^3, CF^{\mathsf{true}}, V_x^v, v & \text{for } v \in Q \\
IP^2, CF^*, V_x^v, w & \mapsto IP^3, CF^{\mathsf{false}}, V_x^v, w & \text{for } w \neq v \\
IP^3, CF^{\mathsf{true}} & \mapsto IP^1, CF^{\mathsf{true}} & \\
IP^3, CF^{\mathsf{false}} & \mapsto IP^4, CF^{\mathsf{false}} & \\
IP^4, OF^*, CF^{\mathsf{true}} & \mapsto IP^5, OF^{\mathsf{false}}, CF^{\mathsf{true}} & \\
IP^4, OF^*, CF^{\mathsf{false}} & \mapsto IP^5, OF^{\mathsf{true}}, CF^{\mathsf{false}} &
\end{array}
$$

Figure 4 Converting instructions into transitions.

For example, in line 1 we want to move one agent from x to y and set the instruction pointer to 2 (from 1). Recall that the registers map to states of the population protocol via the register map, stored in pointers V_x, where $x \in Q$ is a register. We thus have the following agents initiating the transition:

- IP^1; the agents storing the instruction pointer currently stores the value 1,
- V_x^v; the register $x \in Q$ is currently mapped to state $v \in Q$,
- v; an agent in state v, i.e. representing one unit in register x,
- V_y^w; register y is mapped to state w.

The transition then moves v to state w, and increments the instruction pointer.

The above protocol does not come to a consensus. For this to happen, we use a standard output broadcast: 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.

8 Robustness of Threshold Protocols

A major motivation behind the construction of succinct protocols for threshold predicates is the application to chemical reactions. In this, as in other environments, computations must be able to deal with errors. Prior research has considered *self-stabilising* protocols [8, 16, 15]. Such a protocol must converge to a desired output regardless of the input configuration. However, it is easy to see that no population protocol for e.g. a threshold predicate can be self-stabilising (and prior research has thus focused on investigating extensions of the population protocol model).

In our definition of population programs, the program cannot rely on any guarantees about its input configuration, so they are self-stabilising by definition. However, when we convert to population protocols, we retain only a slightly weaker property, defined as follows:

▶ **Definition 7.** *Let $PP = (Q, \delta, I, O)$ denote a population protocol deciding φ with $|I| = 1$. We say that PP is* almost self-stabilising*, if every fair run starting at a configuration $C \in \mathbb{N}^Q$ with $C(I) \geq |Q|$ stabilises to $\varphi(|C|)$.*

So the initial configuration can be almost arbitrary, but it must contain a small number of agents in the initial state. In many contexts, this is a mild restriction. In a chemical reaction, for example, the number of agents (i.e. the number of molecules) is many orders of magnitude larger than the number of states (i.e. the number of species of molecules).

In particular, this is also much stronger than any prior construction. All known protocols for threshold predicates are 1-aware [14], and can thus be made to accept by placing a single agent in an accepting state.

▶ **Theorem 2.** *The protocols of Theorem 1 are almost self-stabilising.*

9 Conclusions

We have shown an $\mathcal{O}(\log\log n)$ upper bound on the state complexity of threshold 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.

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.

Our construction is almost self-stabilising, which shows that it is possible to construct protocols that are quite robust against *addition* of agents in arbitrary states. A natural next step would be to investigate the *removal* of agents: can a protocol provide guarantees in the case that a small number of agents disappear during the computation?

Threshold predicates can be considered the most important family for the study of space complexity, as they are 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 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.

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. doi:10.1145/1011767.1011810.

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

6 Dana Angluin, James Aspnes, and David Eisenstat. Fast computation by population protocols with a leader. *Distributed Comput.*, 21(3):183–199, 2008. doi:10.1007/S00446-008-0067-Z.

7 Dana Angluin, James Aspnes, David Eisenstat, and Eric Ruppert. The computational power of population protocols. *Distributed Comput.*, 20(4):279–304, 2007. doi:10.1007/S00446-007-0040-2.

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. 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. doi:10.4230/LIPICS.STACS.2020.40.

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. doi:10.4230/LIPICS.STACS.2018.16.

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. Breaking through the $\omega(n)$-space barrier: Population protocols decide double-exponential thresholds, 2024. arXiv:2204.02115.

19 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.

20 Philipp Czerner, Javier Esparza, and Jérôme Leroux. Lower bounds on the state complexity of population protocols. *CoRR*, 2021. doi:10.48550/arXiv.2102.11619.

21 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.

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

23 Javier Esparza. Decidability and complexity of petri net problems - an introduction. In Wolfgang Reisig and Grzegorz Rozenberg, editors, *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the volumes are based on the Advanced Course on Petri Nets, held in Dagstuhl, September 1996*, volume 1491 of *Lecture Notes in Computer Science*, pages 374–428. Springer, 1996. doi:10.1007/3-540-65306-6_20.

24 Jérôme Leroux. State complexity of protocols with leaders. In Alessia Milani and Philipp Woelfel, editors, *PODC '22: ACM Symposium on Principles of Distributed Computing, Salerno, Italy, July 25 - 29, 2022*, pages 257–264. ACM, 2022. doi:10.1145/3519270.3538421.

25 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.

26 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.

27 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.

28 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.