Terminating Population Protocols via some Minimal Global Knowledge Assumptions[☆,☆☆]

Othon Michail[a,*], Paul G. Spirakis[a,b]

[a] *Computer Technology Institute & Press "Diophantus" (CTI), Patras, Greece*
[b] *Department of Computer Science, University of Liverpool, UK*

Abstract

We extend the population protocol model with a *cover-time service* that informs a walking state every time it covers the whole network. This represents a known upper bound on the cover time of a random walk. The cover-time service allows us to introduce *termination* into population protocols, a capability that is crucial for any distributed system. By reduction to an oracle-model we arrive at a very satisfactory lower bound on the computational power of the model: we prove that it is *at least as strong as a Turing Machine of space* $\log n$ *with input commutativity*, where n is the number of nodes in the network. We also give a $\log n$-space, but nondeterministic this time, upper bound. Finally, we prove interesting similarities of this model to linear bounded automata.

Keywords:
population protocol, cover-time service, rendezvous-based communication, interaction, counter machine, absence detector, linear-bounded automaton

1. Introduction

Networks of tiny artifacts will play a fundamental role in the computational environments and applications of tomorrow. As a result, over the last decade, there has been a strong focus on theoretical models of pervasive systems, consisting of great numbers of computationally restricted, communicating entities. One such model, called the *Population Protocol* (*PP*) model, has been recently introduced by Angluin *et al.* [AAD+06]. Their aim was to model sensor networks consisting of tiny computational devices (called *agents* or *nodes*) with sensing capabilities that follow some unpredictable and uncontrollable mobility pattern. Due to the minimalistic nature of their model, the class of computable predicates was proven [AAER07] to be fairly small: it is the class of *semilinear predicates* [GS66], which does not support e.g. multiplications, exponentiations, and many other important operations on input variables. Additionally, *population protocols do not halt*. The agents cannot know whether the computation is completed. Instead, they forever interact in pairs while their outputs (but not necessarily their states) stabilize to a certain value.

An interesting question that quickly emerged was whether complex computations could be performed by using simple protocols and combining their functionality. Given the protocols stabilizing behavior, their sequential composition turned out to be a highly non-trivial task. To circumvent this problem, Angluin *et al.* introduced the *stabilizing inputs PPs* [AAC+05] and they showed that multiple protocols can run in parallel

☆Supported in part by the project "Foundations of Dynamic Distributed Computing Systems" (FOCUS) which is implemented under the "ARISTEIA" Action of the Operational Programme "Education and Lifelong Learning" and is co-funded by the European Union (European Social Fund) and Greek National Resources.
☆☆A preliminary version of the results in this paper has appeared in [MCS12].
*Corresponding author (Telephone number: +30 2610 960200, Fax number: +30 2610 960490, Postal Address: Computer Technology Institute & Press "Diophantus" (CTI), N. Kazantzaki Str., Patras University Campus, Rio, P.O. Box 1382, 26504, Greece).
Email addresses: michailo@cti.gr (Othon Michail), P.Spirakis@liverpool.ac.uk (Paul G. Spirakis)

and once one stabilized the others could run correctly (by taking appropriate actions to restore correct execution) using the stabilized output of the former as their input. This approach is, however, fairly slow in terms of the number of interactions (provided some probabilistic assumption on the interaction pattern) since it requires to implement phase clocks based on epidemic protocols (see [AAE08]).

In this work, we follow an alternative approach. We augment the original model of computation with a *cover-time service* (we abbreviate the new model as *CTS*) that informs a walking state every time it covers the whole network. This is simply a known upper bound on the cover time of a random walk. This allows us to introduce *termination* into population protocols, a capability that is crucial for any distributed system. Then we reduce this model to population protocols augmented with an *absence detector*. An absence detector is an oracle that gives hints about *which states are not present in the population*. Each process can interact with this special agent (the absence detector) that monitors other agents in the system, and maintains flags for each state of the protocol. The rest of the model is the same as the PP model. All agents, apart from the absence detector, are modeled as *finite-state machines* that run the *same protocol*. Agents interact in pairs according to some *interaction graph* which specifies the permissible interacting pairs, and during an interaction they update their states according to the common program. *No agent can predict or control its interactions*. Within this framework, we explore the computational capabilities of this new extension, that we call *Population Protocols with Absence Detector* (*AD*), and study its properties on a purely theoretical ground. As we shall see, the AD model is computationally stronger than PPs but this is not what sets it apart. A major new feature of this model is its capability to perform *halting computations*, which allows sequential execution of protocols (i.e. sequential *composition*). Note that although we are currently unaware of how to construct such detectors, in the future, our detector may be implemented via a Bulletin Board regarding the existing states (e.g. each device marks its current state in the board, and all devices can read this board). Such Boards can be implemented easily and have been used in the past [Edi86].

2. Other Previous Work

In the population protocol model [AAD$^+$06], n computational agents are passively mobile, interact in ordered pairs, and the temporal connectivity assumption is a *strong global fairness condition* according to which all configurations that may always occur, occur infinitely often. These assumptions give rise to some sort of structureless interacting automata model. The usually assumed *anonymity* and *uniformity* (i.e. n is not known) of protocols only allow for commutative computations that eventually stabilize to a desired configuration. Most computability issues in this area have now been established. Constant-state nodes on a complete interaction network (and several variations) compute the *semilinear predicates* [AAER07]. Semilinearity persists up to $o(\log\log n)$ local space but not more than this [CMN$^+$11]. If constant-state nodes can additionally leave and update fixed-length pairwise marks then the computational power dramatically increases to the commutative subclass of $\mathbf{NSPACE}(n^2)$ [MCS11a]. If nodes are equipped with unique ids and are also allowed to store a fixed number of other nodes' ids then again an extremely powerful model is obtained being equivalent to the commutative subclass of $\mathbf{NSPACE}(n\log n)$ [GR09]. For a very recent introductory text see [MCS11b]. Finally, our CTS model is different from the cover times considered in [BBCK10] in that we allow protocols to know the cover times and in that the cover times in [BBCK10] refer to the time for an agent to meet all other agents in the network.

3. Our Results - Roadmap

In Sections 4 and 5, the newly proposed models are formally defined. Subsection 5.1 in particular, defines halting and output stabilizing computations, as well as the classes of predicates that the AD model can compute in both cases. In Section 6, we illustrate the new model with a simple leader election protocol and give some properties of the AD concerning halting computations. Section 7 first establishes the computational equivalence of the CTS and AD models and then deals with the computational power of the latter. In particular, Section 7.1 shows that all semilinear predicates (whose class is denoted by **SEM**) are stably computable by halting ADs. In Section 7.2, several improved computational lower bounds and an upper

bound are presented. In particular, it is first shown that the class **HAD**, of all predicates computable by some AD with a unique leader, includes all multiplication predicates of the form $(bN_1^{d_1}N_2^{d_2}\cdots N_k^{d_k} < c)$, where b, c, d_i, k are constants, $b, c \in \mathbb{Z}$ and $d_i, k \in \mathbb{Z}^+$. We do so by constructing an AD (Protocol 2) that performs *iterative computation*. Then in Subsection 7.2.1 it is shown that halting ADs can compute any predicate whose support (corresponding language on the input alphabet) is decidable by a Turing Machine (TM) of $O(\log n)$ space. This is shown by *simulating a One Way k-Counter Machine* (k-CM) [FMR68, Min61] with halting ADs. Moreover, it is shown that all predicates in **HAD** are stably computable by a TM of $O(\log^2 n)$ space. Finally, some similarities of the AD model with *Multilset Linear Bounded Automata with Detection* (*MLBAD*s) are pointed out and it is established that ADs can simulate such automata. In Section 9, we conclude and present potential future research directions.

4. A Cover-time Service

We equip pairwise-interacting agents with the following natural capability: *swapping states can know when they have covered the whole population*. Note that we refer to states and not to nodes. A node may possibly not be ever able to interact with all other nodes, however if nodes constantly swap their states then the resulting random walk must be capable of covering the whole population (e.g. in a complete graph the cover time of a random walk is $n \log n$).

We assume a unique leader in the population which jumps from node to node. What we require is that the leader state *knows* when it has passed from all nodes and we require this to hold iteratively, that is after it knows that it has covered the whole population it can know the same for the next walk, and so on. So we just assume a *cover-time service* which is a black-box for the protocol. We call this extension of PPs with leader and a cover-time service the *Cover-Time Service* (*CTS*) model.

Formally, we are given a population V of n agents s.t. initially a node u is in state $(l, D, 0)$ while all other nodes are in state $\perp$. What we require is that $D \in \mathbb{N}$ satisfies the following. If in every interaction (v, w), s.t. the state of v is (l, D, i) and the state of w is $\perp$, v updates to $\perp$ and w to $(l, D, i+1)$ (swapping their states and increasing i by one) if $D > i + 1$ and to $(l, D, 0)$ otherwise, then in every D consecutive steps $s_1, \ldots, s_D$ (where we can w.l.o.g. assume that a single interaction occurs in each step) it holds that $\{z \in V : z \text{ has obtained } l \text{ at least once in the interval } [s_1, s_D]\} = V$. That is D is an upper bound on the time needed for a swapping state (here l) to visit all nodes (called the *cover-time*). The leader state, no matter which node it lies on, can detect the coverage of V when the step/interaction counter i becomes equal to D. We assume that both D and i are only used for coverage detection and not as additional storage for internal computation (nodes keep operating as finite-state machines). Another way to appreciate this is by imagining that all nodes have access to a global clock that ticks every D rounds. Finally, observe that D is always an upper bound on $n - 1$, because at least $n - 1$ interactions are needed for a state to cover V. Of course, in some cases it might be possible for nodes to have more drastic knowledge, like for example knowledge of n. Then the mechanism of detecting coverage is similar but more intuitive. All nodes have a *status* component which takes values from $\{bottom, top\}$. All are *bottom* apart from the initial leader which is *top* and has the counter token $i = 1$. When the leader meets a *bottom*, it switches it to *top*, the leader swaps from one node to the other, and i is increased by one. When $i = n$ (n is known in advance in this scenario), n distinct *bottom*s (the initial leader inclusive) have been converted to *top* so a covering has been completed. The next covering is performed by setting $i = 1$ and converting *top*s to *bottom*s this time, and so on. Again, we assume that n and the local counter i of size n are only used for covering detection and not for any other sort of internal computation (as stated above, the cover-time service is a black-box that a protocol may use only for covering detection).

We explore the computability of the CTS model. In particular, we arrive at an exact characterization of its computational power. We do so by reducing the CTS model to an artificial but convenient variant of population protocols that is equipped with a powerful oracle-node capable of detecting the presence or absence of any state from the population. At a first glance our model may seem to be of pure theoretical interest. However, we expect that even "more applied" variations of population protocols equipped with some similar capability of detecting termination (even via more local mechanisms) may one way or another reduce to our model.

5. Absence Detectors

A *Population Protocol with Absence Detector* (AD) is a 7-tuple $(X, Y, Q, I, \omega, \delta, \gamma)$ where X, Y and Q are finite sets and X is the *input alphabet*, Y is the *output alphabet*, Q is a set of *states*, $I : X \to Q$ is the *input function*, $\omega : Q \to Y$ is the output function, δ is the transition function $\delta : Q \times Q \to Q \times Q$ and γ is the *detection transition function* $\gamma : Q \times \{0,1\}^{|Q|} \to Q$. If $\delta(a, b) = (c, d)$, where $a, b, c, d \in Q$, we call $(a, b) \to (c, d)$ a *transition* and we define $\delta_1(a, b) = c$ and $\delta_2(a, b) = d$. We also call transition any $(q, a) \to c$, where $q, c \in Q$, $a \in \{0,1\}^{|Q|}$ so that $\gamma(q, a) = c$.

An AD runs on the nodes of an interaction graph $G = (V, E)$ where G is a directed graph without self-loops and multiple edges, V is a *population* of n *agents* plus a single *absence detector* ($n + 1$ entities in total), and E is the set of permissible, ordered interactions between two agents or an agent and the absence detector. An absence detector is a special node whose state is a vector $a \in \{0,1\}^{|Q|}$, called *absence vector*, always representing the absence or not of each state from the population; that is, $q \in Q$ is absent from the population in the current configuration iff $a[q] = 1$. From now on we will denote the absence detector by a unless stated otherwise. Throughout this work we consider only complete interaction graphs, that is all agents may interact with each other and with the absence detector.

Initially, each agent except the absence detector senses its environment (as a response to a global start signal) and receives an input symbol from X. We call an *input assignment* to the population, any string $x = \sigma_1\sigma_2 \ldots \sigma_n \in X^*$, where by n we denote the population size. Then all agents that received an input symbol apply the input function on their symbols and obtain their initial states. Given an input assignment x the absence detector is initialized by setting $a[q] = 0$ for all $q \in Q$ so that $\exists \sigma_k \in x : I(\sigma_k) = q$ and $a[q] = 1$ for all other $q \in Q$.

A *population configuration*, or more briefly a *configuration* is a mapping $C : V \to Q \cup \{0,1\}^{|Q|}$ specifying a state $q \in Q$ for each agent of the population and a vector $a \in \{0,1\}^{|Q|}$ for the absence detector. We call an *initial configuration*, a configuration that specifies the initial state of each agent of the population and the initial absence vector of the absence detector w.r.t. a given input assignment x (as previously described). Let C, C' be two configurations and $u \in V - \{a\}$, $a \in \{0,1\}^{|Q|}$ be an agent and the absence vector of the detector, respectively. We denote by $C(u)$ the state of agent $u \in V$ under configuration C. We say that C *yields* C' *via encounter* $(u, a) \in E$ and denote by $C \xrightarrow{(u,a)} C'$, if $C'(u) = \gamma(C(u), a)$, $C'(w) = C(w)$, $\forall w \in (V - \{u, a\})$ and $C'(a) = a'$ so that $a'[q] = 0$, $\forall q \in Q$ where $\exists w \in V : C'(w) = q$ and $a'[q] = 1$ otherwise. The previous transition can be similarly defined for the reverse interaction (a, u). In addition, given two distinct agents $u, v \in V$, where $u, v \neq a$, we say that C *yields* C' *via encounter* $e = (u, v) \in E$ and denoted by $C \xrightarrow{e} C'$, if $C'(u) = \delta_1(C(u), C(v))$, $C'(v) = \delta_2(C(u), C(v))$, $C'(w) = C(w)$, for all $w \in (V - \{u, v, a\})$ and $C'(a) = a'$ updated as previously. We say that C can go to C' in one step, denoted $C \to C'$, if $C \xrightarrow{t} C'$ for some $t \in E$. We write $C \xrightarrow{*} C'$ if there is a sequence of configurations $C = C_0, C_1, \ldots, C_k = C'$, such that $C_i \to C_{i+1}$ for all i, $0 \leq i < k$, in which case we say that C' is *reachable* from C.

We call an *execution* any finite or infinite sequence of configurations $C_0, C_1, C_2, \ldots$, where C_0 is an initial configuration and $C_i \to C_{i+1}$, for all $i \geq 0$. The interacting pairs are chosen by an adversary. A strong global *fairness condition* is imposed on the adversary to ensure the protocol makes progress. An infinite execution is *fair* if for every pair of configurations C and C' such that $C \to C'$, if C occurs infinitely often in the execution then so does C'. An adversary scheduler is fair if it always leads to fair executions. A *computation* is an infinite fair execution. An interaction between two agents is called *effective* if at least one of the initiator's or the responder's states is modified (that is, if C, C' are the configurations before and after the interaction, respectively, then $C' \neq C$).

Note that since X, Y, and Q are finite, the description of an AD is independent from the population size n. Moreover, agents cannot have unique identifiers (uids) since they are unable to store them in their memory. As a result, the AD model preserves both *uniformity* and *anonymity* properties that the basic Population Protocols have.

5.1. Stable Computation

We call a *predicate over* X^* any function $p : X^* \to \{0,1\}$. p is called *symmetric* if for every $x \in X^*$ and any x' which is a permutation of x's symbols, it holds that $p(x) = p(x')$ (in words, permuting the input symbols does not affect the predicate's outcome). In this work we are interested in the computation of symmetric predicates.

A configuration C is called *output stable* if for every configuration C' that is reachable from C it holds that $\omega(C'(u)) = \omega(C(u))$ for all $u \in V$, where $\omega(C(u))$ is the output of agent u under configuration C. In simple words, no agent changes its output in any subsequent step and no matter how the computation proceeds. We assume that a is the only agent that does not have an output. So the output of the population concerns only the rest of the agents.

A predicate p over X^* is said to be *stably computable* by the AD model, if there exists a AD $\mathcal{A}$ such that for any input assignment $x \in X^*$, any computation of $\mathcal{A}$ on a complete interaction graph of $|x| + 1$ nodes beginning from the initial configuration corresponding to x reaches an output stable configuration in which all agents except a output $p(x)$.

The existence of an absence detector allows for halting computations. We say that an AD $\mathcal{A} = (X_{\mathcal{A}}, Y_{\mathcal{A}}, Q_{\mathcal{A}}, I_{\mathcal{A}}, \omega_{\mathcal{A}}, \delta_{\mathcal{A}}, \gamma_{\mathcal{A}})$ is *halting* if there are two special subsets $Q_{h\_accept}, Q_{h\_reject} \subseteq Q_{\mathcal{A}}$ s.t. every agent eventually goes in some $q \in Q_{h\_accept}$ ($q \in Q_{h\_reject}$ resp.) and from that point on stops participating in effective interactions (i.e. it halts), giving output 1 (0 resp.). We say that a predicate p over X^* is *computable by a halting AD* $\mathcal{A}$ if for any input assignment $x \in X^*$, any computation of $\mathcal{A}$ on a complete interaction graph of $|x| + 1$ nodes beginning from the initial configuration corresponding to x reaches an output stable configuration in which, *after a finite number of interactions*, all agents, except for a, are in states of Q_{h_accept} if $p(x) = 1$ and of Q_{h_reject} otherwise.

Let $\mathbf{SPACE}(f(n))$ ($\mathbf{NSPACE}(f(n))$) be the class of languages decidable by some (non) deterministic TM in $O(f(n))$ space. For any class $\mathbf{L}$ denote by $\mathbf{SL}$ its commutative subclass. In addition, we denote by $\mathbf{SEM}$, the class of the semilinear predicates, consisting of all predicates definable by first-order logical formulas of Presburger arithmetic (see, e.g., [GS66]).

6. Examples and Properties

We begin with a leader-election AD. $X = \{1\}$, $Q = \{l, f, l_{halt}, f_{halt}\}$, $I(1) = f$, δ is defined as $(l, f) \to (l, f_{halt})$, and γ as $(f, a) \to l$, if $a[l] = 1$ and $(l, a) \to l_{halt}$, if $a[f] = 1$. State l is the leader state and state f is the follower (or non-leader) state. Note that both the output alphabet and the output function are not specified since the output is meaningless in this setting. The interactions that are not specified in δ and γ are ineffective.

Proposition 1. *The above protocol is a leader election AD, that is eventually all nodes terminate with a single $u \in V\backslash\{a\}$ in state l and all other $v \in V\backslash\{u, a\}$ in state f.*

Proof. Initially all nodes are in state f (i.e. followers). When the first (f, a) interaction occurs, state l is absent from the population and $a[l] = 1$ is true, thus f becomes l which is now a unique leader in the population. From now on, $a[l] = 0$ so (f, a) becomes ineffective and no more leaders can be created. The unique leader starts setting one after the other all followers to f_{halt} and when eventually fs become exhausted it will hold that $a[f] = 1$, consequently a single (l, a) interaction will also make the unique leader halt by turning its state to l_{halt}. □

The following are some interesting properties of the AD model. We say that an AD $\mathcal{A}$ has *stabilizing states* if every execution of $\mathcal{A}$ reaches a state-stable configuration C, that is a configuration C such that $C' = C$ for all C' reachable from C. In terms of absence vectors, $a \in \{0,1\}^{|Q|}$ is state-stable iff for all $q_1, q_2 \in Q$ (not necessarily distinct) such that $a[q_1] = a[q_2] = 0$, it holds that $\delta(q_1, q_2) = (q_1, q_2)$ and $\gamma(q_1, a) = q_1$.

Proposition 2. *Any AD with stabilizing states has an equivalent halting AD.*

Proof. As the AD has stabilizing states, it follows that $\{0,1\}^{|Q|}$ can be partitioned into a state-stable subset and a state-unstable subset. If we let all agents know in advance the above partitioning (note that this is constant information, so storing it is feasible) then we have the required termination criterion; that is, an agent halts iff it encounters a detector with a state-stable absence vector. □

From now on, we only consider ADs that halt.

A very interesting feature of ADs is that they can be sequentially composed. This means that given two ADs $\mathcal{A}$ and $\mathcal{B}$ we can construct a AD $\mathcal{C}$ which has the input of $\mathcal{A}$ and the output of $\mathcal{B}$ given $\mathcal{A}$'s output as input. First, $\mathcal{C}$ runs as $\mathcal{A}$ on its inputs and once the absence detector detects $\mathcal{A}$'s halt, $\mathcal{C}$ starts $\mathcal{B}$'s execution on using the output of $\mathcal{A}$ as input. The next theorem exploits the sequential composition of ADs to show that any AD can assume the existence of a unique leader.

Proposition 3. *Any AD $\mathcal{A}$ has an equivalent AD $\mathcal{B}$ that assumes a unique leader which does not obtain any input.*

Proof. For the one direction, $\mathcal{B}$ may trivially simulate $\mathcal{A}$ by ignoring the leader. Then for all computations of $\mathcal{A}$ on n agents there is an equivalent computation of $\mathcal{B}$ on $n+1$ agents. For the other direction, $\mathcal{A}$ first elects a unique leader and then simulates $\mathcal{B}$ by considering the input of the agent that has been elected as a leader as a "virtual" agent. The leader creates a bit which moves between the non-leaders. Whenever the leader encounters the bit it interacts with the virtual agent that it carries in its own state. The role of the leader in the "virtual" interaction, that is, whether it is the initiator or the responder can be determined by its role in the real interaction in which it encountered the bit. Note that $\mathcal{B}$'s computations on $n+1 \geq 3$ agents are simulated by $\mathcal{A}$ on n agents. □

Based on this fact, we only consider ADs that assume the existence of such a unique leader in the initial configuration that is responsible for all effective interactions (non-leader interactions do not cause state modifications). Observe that ignoring the effective interactions between the non-leaders does not affect the computational power of the model. Effective interactions $(a,b) \to (a',b')$ between two non-leaders can be mediated by the leader as follows: $(a,l) \to (a,l_a)$, $(l_a,b) \to (l_{a,b},b')$, $(a,l_{a,b}) \to (a',l)$. The other direction holds trivially, because ignoring effective interactions between non-leaders is a special case of not ignoring them. We denote by **HAD** *the class of all predicates computable by some AD with a unique leader.*

6.1. The Power of 2 protocol

We now construct an AD that computes the non-semilinear predicate $(N_1 = 2^d)$, which is true if the number of 1s in the input is a power of 2 (Protocol 1). This protocol illustrates the ability of ADs to perform iterative computations (which is impossible in the original PP model).

The protocol essentially implements a classical TM deciding the language consisting of all strings of 1s whose length is a power of 2 (see e.g. page 145 of [Sip06]). The TM in each iteration cuts the number of 1s in half. If during an iteration it finds an odd number of 1s it rejects and if it ever finds two 1s it accepts (we have here disregarded the trivial case of a single 1 that is of course a power of 2). In our case, the input is distributed and stored in the second component of the non-leaders' state, where the first component of a non-leader is f and the first component of a leader is l. The second component of a leader simulates the state of the TM. The protocol exploits the fact that even without an ordering on the input symbols (which is the case for a TM tape but not for a passively interacting system) it is still possible to cut half of them and to determine their parity. The leader does this by marking the 1s alternately as $\bar{1}$ and $1'$. Marking guarantees that the same input symbol will not be counted twice. In particular, the $\bar{1}$s form the eliminated half while each $1'$ will be restored to 1 in the end of the iteration to form the set of 1s that will be halved in the next iteration. All operations discussed so far can also be implemented in the original population protocol model. The only distinguishing operation is the ability to detect termination of an iteration. This is achieved via the absence detector. In particular, when the leader learns from the absence detector that there are no more 1s in the population (which occurs when all 1s are marked) while there are still even $1'$s (the latter encoded by leader's state q_2), the next iteration begins. In case the $1'$s are odd (encoded by leader's state q_3) the leader rejects. Acceptance occurs if at some iteration both the 1s and the $1'$s have been exhausted and the

Protocol 1 *Power of 2*

1: $X = \{1\}, Q = (\{l\} \times \{q_0, q_1, q_2, q_3, q_4\}) \cup (\{f\} \times \{1, \bar{1}, 1'\}) \cup \{q_{accept}, q_{reject}\}$,
2: $I(1) = (f, 1)$ only for the non-leaders,
3: the leader is initialized to (l, q_0),
4: δ:

$$\begin{aligned}
(l, q_0), (f, 1) &\to (l, q_1), (f, \bar{1}) \\
(l, q_1), (f, 1) &\to (l, q_2), (f, \bar{1}) \\
(l, q_2), (f, 1) &\to (l, q_3), (f, 1') \\
(l, q_3), (f, 1) &\to (l, q_2), (f, \bar{1}) \\
(l, q_4), (f, 1') &\to (l, q_4), (f, 1)
\end{aligned}$$

5: γ:

$$\begin{aligned}
(l, q_2), a &\to q_{accept}, \text{ if } a[f, 1] = a[f, 1'] = 1 \\
&\to (l, q_4), \text{ if } a[f, 1] = 1 \text{ and } a[f, 1'] = 0 \\
(l, q_3), a &\to q_{reject}, \text{ if } a[f, 1] = 1 \text{ and } a[f, 1'] = 0 \\
(l, q_4), a &\to (l, q_1), \text{ if } a[f, 1'] = 1
\end{aligned}$$

leader is in state q_2, which corresponds to having found precisely two 1s (in fact a single one erased in this iteration as the other was erased from the very beginning and is always assumed to exist throughout the execution).

7. Computational Power

We now explore the computational power of the CTS model via the AD model. In particular, we provide several lower bounds and an upper bound for the class **HAD**. By Theorem 1 (presented below) these results carry over to the class of languages computable by CTS protocols.

Theorem 1. *The CTS model is computationally equivalent to the leader-AD model.*

Proof. To simulate absence detectors by the CTS model we have the unique leader l to play both the role of the absence detector and the role of the leader of the computation. Whenever the leader wants to talk to the absence detector it begins the process to form the absence vector. To do this it starts walking throughout the graph keeping track of the existence or not of all possible states; that is if it encounters some state q then it simply remembers the existence of this state. When l is informed by the service that its walk has been completed (that is it has covered all nodes) it knows that it has obtained the correct absence vector (being true for those states that were not found in the population).

For the other direction, we show how the absence detection model can know when a walk has covered the whole population: as the leader state walks over the graph it leaves marks on the nodes it crosses. Any time that a mark is left, the walking state waits to talk to the absence detector and asks it whether there is some node without a mark. When the absence detector says that all nodes have been marked the walking state knows that it has covered the whole population. □

7.1. PPs vs ADs

In [AAER07], they defined the k-truncate of a configuration $c \in \mathbb{N}^Q$ as $\tau_k(c)[q] := \min(k, c[q])$ for all $q \in Q$.

Lemma 1. *For all finite k and any initial configuration $c \in \mathbb{N}^Q$, there is an AD that aggregates in one agent $\tau_k(c)$.*

Proof. The unique leader is aware of the finite bound k and initiates a $|Q|$-vector full of zeros except for a 1 in the position of its own state (note that since the leader election protocol is halting we are allowed to first elect a leader and then execute a second procedure based on the assumption of a leader). When a leader interacts with a non-leader, then the non-leader halts and if the leader's counter corresponding to the non-leader's state was less than k, then the leader increments it by one. The leader halts when the absence detector informs it that non-leaders are absent. □

Theorem 2. SEM $\subseteq$ HAD.

Proof. It was proved in [AAER07] that, for any PP with stabilizing outputs, there exists a finite k such that a configuration is output stable iff its k-truncate is output stable (and the output values are preserved). We let the AD know the k corresponding to the simulated PP. The AD-leader performs a constant number of simulation steps, e.g. k, and then does the following. It marks all non-leaders one after the other, while gathering the k-truncate of their current configuration c. When the detector informs the leader that no unmarked non-leaders have remained, the leader checks whether $\tau_k(c)$ is output-stable (since k is finite and independent of the population size, we may as in Proposition 2 assume that the leader knows in advance the subset of output stable k-truncates). If it is, then c must also be output stable and the protocol halts. If not, then neither is c and the leader drops this truncate, restores one after the other all non-leaders and when no marked non-leader has remained it continues the PP's simulation for another constant number of steps, and so on. □

Taking into account Theorem 2 and the non-semilinear power of 2 predicate (Protocol 1) we have that **SEM $\subsetneq$ HAD**.

7.2. Better Lower Bounds and an Upper Bound

We construct now an AD that computes the predicate $(bN_1^{d_1}N_2^{d_2}\cdots N_k^{d_k} < c)$, where b and c are integer constants and d_i and k are nonnegative constants. We again make w.l.o.g. the assumption of a unique leader, and for further simplification we forget about the leader's input.

To simplify the description we first present an AD that computes $(bN_1^d < c)$ for integers b, c and nonnegative d, where in general N_i is used to denote the number of agents with input i. The idea is to have the leader maintain a counter of the sum computed so far, initially equal to $-c$. Let for example both b and c be positive. So, initially the leader begins with a negative sum and then it tries to exploit the nodes with input 1 in order to add b to its sum N_1^d times and check whether the sum remains < 0 or not. Of course, it cannot add the whole sum to its memory because its state-pace is finite and independent of n. Instead, if at some point the sum becomes ≥ 0, the leader can terminate and output 0 (i.e. that the predicate is not true) because in the sequel the sum cannot change sign again (it can only increase by adding more bs to it). So, the interesting part of the protocol, which we now explain, is to count (potentially) up to N_1^d. The idea is for the protocol to construct one after the other all possible d-tuples of the form $(m_1, m_2, \ldots, m_d)$, where $1 \leq m_i \leq N_1$. Observe that there are N_1^d such distinct tuples. To do this, the protocol initializes every agent with input 1 with a d-tuple of bits, initially all equal to 0. The protocol sets a value to each m_i by setting the number of agents whose component i is 1. Initially, for all $1 \leq i \leq d-1$ it sets an i component to 1. In this way it constructs the first $(d-1)$-tuple, $(1, 1, \ldots, 1)$. Then it starts converting one after the dth component of each agent to 1, thus creating one after the other the d-tuples $(1, 1, \ldots, 1, 1), (1, 1, \ldots, 1, 2), \ldots, (1, 1, \ldots, 1, N_1)$, every time also adding b to the leader's sum. That all possible d-tuples corresponding to the $(d-1)$-tuple $(1, 1, \ldots, 1)$, can be detected by the absence of a 0 from the dth component of the agents, i.e. when all of them are 1 and $m_d = N_1$ holds. Then the leader restores all d-th components to 0, converts another $d-1$ component to 1 in order to create the next $(d-1)$-tuple, $(1, 1, \ldots, 2)$, and repeats the previous process to construct again all d-tuples corresponding to it. By recursive application of this idea to all the components of the d-tuple it is not hard to see that all possible combinations can be constructed.

To present the protocol formally, define $[c] := \{0, 1, \ldots, |c|\}$ if $c < 0$ and $[c] := \{-c, -c+1, \ldots, 0\}$ if $c \geq 0$. Define u_{-i} to be the subvector of a vector u consisting of all components of u except from component i. We write a vector u as (j, u_{-i}) when we want to emphasize that component i of u has the value j. Given an absence vector a, $a[j, u_{-i}] = 1$ is true iff (j, u_{-i}) is absent from the population for all u_{-i}. The protocol, called *VarPower*, is presented in Protocol 2.

Protocol 2 *VarPower*

1: $X = \{s_1\}, Q = (\{l_1, l_2, \ldots, l_d, l_1^e, l_2^e, \ldots, l_d^e\} \times [c]) \cup \{0,1\}^d \cup \{q_{accept}, q_{reject}\}$,
2: $I(s_1) = 0^d$,
3: the initial state of the leader is $(l_1, -c)$,
4: δ:

$$\begin{aligned}
(l_i, w), (0, u_{-i}) &\to (l_{i+1}, w), (1, u_{-i}), \text{ if } i < d \\
&\to q_{accept}, \text{ if } i = d \text{ and } c \geq 0, w + b \leq -c \text{ or } c < 0, w + b < 0 \\
&\to q_{reject}, \text{ if } i = d \text{ and } c \geq 0, w + b \geq 0 \text{ or } c < 0, w + b \geq -c \\
&\to (l_i, w + b), (1, u_{-i}), \text{ if } i = d \text{ and } c \geq 0, -c \leq w + b < 0 \text{ or} \\
&\qquad\qquad c < 0, 0 \leq w + b < |c| \\
(l_i^e, w), (1, u_{-i}) &\to (l_i^e, w), (0, u_{-i})
\end{aligned}$$

5: γ:

$$\begin{aligned}
(l_i, w), a &\to (l_i^e, w), \text{ if } a[0, u_{-i}] = 1 \text{ and } i > 1 \\
&\to q_{accept}, \text{ if } a[0, u_{-i}] = 1, i = 1 \text{ and } w < 0 \\
&\to q_{reject}, \text{ if } a[0, u_{-i}] = 1, i = 1 \text{ and } w \geq 0 \\
(l_i^e, w), a &\to (l_{i-1}, w), \text{ if } a[1, u_{-i}] = 1
\end{aligned}$$

We now extend the above construction to devise an AD for the predicate $(bN_1^{d_1} N_2^{d_2} \cdots N_k^{d_k} < c)$, where b and c are integer constants and k is a nonnegative constant. The idea is simple. The leader now holds a k-vector of vectors, l, where l_i is a d_i-vector of states, similar to those of Protocol 2, in order to execute k copies of Protocol 2. The leader still holds a unique counter initialized to $-c$. Similarly, each agent has k components, one for each subprotocol. The AD, in fact, produces all possible assignments of states to l_{ij}. Initially, one step of each subprotocol is executed, then all steps of subprotocol k is executed, then k is reinitialized, $k-1$ is proceeded for one step and again all possible steps of k are executed, when all possible combinations of $k-1$ and k have been exhausted, $k-2$ proceeds for one step, and all possible combinations of $k-1$ and k are reproduced, and so on. After each step, except for the first $k-1$ steps, the terminating conditions of Protocol 2 are checked and if no one is satisfied b is added to the leader's counter.

Finally, by exploiting the above constructions we devise an AD that computes the predicate

$$\sum_{d_1, d_2, \ldots, d_k = 0}^{l} a_{d_1, d_2, \ldots, d_k} N_1^{d_1} N_2^{d_2} \cdots N_k^{d_k} < c,$$

where $a_{d_1, d_2, \ldots, d_k}$ and c are integer constants and l and k are nonnegative constants. Here, a difference to the previous protocol is that we have many copies of it running in parallel, their number being equal to the number of nonzero coefficients, and each one of them adds to the counter its own coefficient $a_{d_1, d_2, \ldots, d_k}$. A key difference is that the counter bounds are now set to $-s, s$, where $s := \max(\max_{d_1, d_2, \ldots, d_k = 0, \ldots, l} |a_{d_1, d_2, \ldots, d_k}|, |c|)$, and that when we say "in parallel" we can implement this in a round-robin fashion, and let the protocol terminate when no subprotocol can proceed without exceeding the bounds. Then the halting decision simply depends on whether the leader's counter is negative or not. We conclude with the following lower bound on **HAD**.

Theorem 3. *Any predicate of the form*

$$\sum_{d_1,d_2,\ldots,d_k=0}^{l} a_{d_1,d_2,\ldots,d_k} N_1^{d_1} N_2^{d_2} \cdots N_k^{d_k} < c,$$

where $a_{d_1,d_2,\ldots,d_k}$ *and* c *are integer constants and* l *and* k *are nonnegative constants, is in* **HAD**.

7.2.1. Simulating a Counter Machine

In this Section, we prove that ADs and *one-way (online) counter machines* (CMs) [FMR68, Min61] can simulate each other.

A one-way (online) k-counter machine (k-CM) [FMR68, Min61] consists of a finite control unit, k nonnegative integer counters, and an input terminal. Formally, a k-CM is an 8-tuple $(Q_p, Q_a, \Sigma, M, K, s_0, q_{accept}, q_{reject})$, where Q_p, Q_a, Σ are all finite sets and

1. Q_p is the set of *polling states*,
2. Q_a is the set of *autonomous states*,
3. Σ is the *input alphabet*,
4. $M : (Q_a \cup (Q_p \times \Sigma)) \times \{0,1\}^k \to Q_a \cup Q_p$ is the *state transition function*,
5. $K : (Q_a \cup (Q_p \times \Sigma)) \times \{0,1\}^k \to \{-1,0,1\}^k$ is the *counter updating function*,
6. $s_0 \in Q_p$ is the *initial state*,
7. $q_{accept} \in Q_p$ is the *accept state*, and
8. $q_{reject} \in Q_p$ is the *reject state*.

Define $sg : \mathbb{N} \to \{0,1\}$ as $sg(x) = 0$ if $x = 0$ and $sg(x) = 1$ otherwise and extend sg to $\mathbb{N}^k$ by $sg(x_1, x_2, \ldots, x_k) = (sg(x_1), sg(x_2), \ldots, sg(x_k))$. A configuration of a k-CM is a member of $(Q_a \cup Q_p) \times \Sigma^* \times \mathbb{N}^k$. We write $(q, aw, (x_1, \ldots, x_k)) \to (q'', w, (y_1, \ldots, y_k))$ where $q \in Q_p$ if $M(q, a, sg(x_1, \ldots, x_k)) = q''$ and $(x_1, \ldots, x_k) + K(q, a, sg(x_1, \ldots, x_k)) = (y_1, \ldots, y_k)$ and we write $(q', w, (x_1, \ldots, x_k)) \to (q''', w, (z_1, \ldots, z_k))$ where $q' \in Q_a$ if $M(q', sg(x_1, \ldots, x_k)) = q'''$ and $(x_1, \ldots, x_k) + K(q', sg(x_1, \ldots, x_k)) = (z_1, \ldots, z_k)$.

In words, a k-CM works as follows. Whenever it is in a polling state (note that it begins from a polling initial state) it reads and consumes the first symbol of the input, it also reads its own state and the set of zero counters and updates its own state and independently adds -1, 0, or 1 to each one of the k counters. On the other hand, when it is in an autonomous state it does the same except from reading and consuming any symbol from the input. The machine halts when it enters one of the states q_{accept} and q_{reject}.

The *space* required by a CM in processing its input is the maximum value that any of its counters obtains in the course of the computation. A language $L \subseteq \Sigma^*$ is said to be *CM-decidable in* $O(f(n))$ *space* if some CM which operates in space $O(f(n))$ accepts any $w \in L$ and rejects any $w' \in \Sigma^* \backslash L$. Let **CMSPACE**$(f(n))$ (**NCMSPACE**$(f(n))$ for nondeterministic CMs) be the class of all languages that are CM-decidable in $O(f(n))$ space. Recall that by **SCMSPACE**$(f(n))$ (**SNCMSPACE**$(f(n))$) we denote its symmetric subclass. The following well-known theorem states that any CM of space $O(f(n))$ can be simulated by a TM of space $O(\log f(n))$ and conversely.

Theorem 4 ([FMR68])**.** **CMSPACE**$(f(n)) =$ **SPACE**$(\log f(n))$ *and* **NCMSPACE**$(f(n)) =$ **NSPACE**$(\log f(n))$.

The above result can also be found as Lemma 3, page 94, in [Iba04].

Corollary 1. **SCMSPACE**$(f(n)) =$ **SSPACE**$(\log f(n))$ *and* **SNCMSPACE**$(f(n)) =$ **SNSPACE**$(\log f(n))$.

We are now ready to show that ADs can simulate linear-space CMs that compute symmetric languages. We first present a proof idea to simplify the reading of the formal proof. The CM consists of a control unit, an input terminal, and a constant number of counters. The AD simulates the control unit by electing a unique leader, which is responsible for carrying out the simulation. The input terminal corresponds to the actual input slots of the agents, each agent obtaining a single input symbol from the input alphabet Σ of the CM. The k counters are stored by creating a k-vector of bits in the memory of each agent. In this manner,

each counter is distributed across the agents. The value of the ith counter at any time is determined by the number of 1s appearing in the ith components of the agents. Since the number of agents is equal to the number of input symbols the space of each counter is linear to the input size (in fact, we can easily make this $O(n)$ by allowing c bits in each component instead of just one). To take a step, the CM reads or not the next symbol from the input and the sign (0 or positive) of each tape and then, if it read the input, moves to the next input symbol and updates the contents of the counters. The leader of the AD waits or not to encounter an agent whose input is not erased (unread), in the former case erases that input symbol, and waits to encounter the absence detector to learn the set of zero counters. When the latter happens, the leader obtains a vector of -1s, 0, and 1, representing the value to be added to each counter. From that point on, the leader adds these values wherever possible until all of them have been added. Then the leader continues the simulation as above.

Theorem 5. $\mathbf{SCMSPACE}(n) \subseteq \mathbf{HAD}$.

Proof. Take any $L \in \mathbf{SCMSPACE}(n)$. Let $\mathcal{S} = (Q_p, Q_a, \Sigma, M, K, s_0, q_{accept}, q_{reject})$ be a k-CM that decides L in space $O(n)$. We construct an AD $\mathcal{A} = (X, Q, I, \delta, \gamma)$ that computes L by simulating $\mathcal{S}$ (it is easy to see that there is no need for specifying an output alphabet and an output function). By Proposition 3, $\mathcal{A}$ may assume the existence of a unique leader that does not obtain any input.

Let ε denote the empty string. Define $\bar{X} := \{\bar{\sigma} \mid \sigma \in X\}$ for any set X. Define also $b = (b_1, b_2, \ldots, b_k) \in \{0,1\}^k$ by $b_i := 1$, for all $1 \leq i \leq k$, iff there exists an agent with non-null u_i, that is, in terms of the absence vector a, iff there exists a $\sigma \in X \cup \bar{X}$ and a $u = (1, u_{-i}) \in \{0,1\}^k$ such that $a[\sigma, (1, u_{-i})] = 0$. Finally, given any $v \in \{-1,0,1\}^k$ and $u \in \{0,1\}^k$, define $t_1(v,u) \in \{-1,0,1\}^k$ and $t_2(v,u) \in \{0,1\}^k$ as follows ($t_{1,i}(v,u)$, $t_{2,i}(v,u)$ denote the ith components of $t_1(v,u)$ and $t_2(v,u)$, respectively): $t_{1,i}(v,u) := 0$ if $v_i + u_i \in \{0,1\}$ and $t_{1,i}(v,u) := v_i$ otherwise, and $t_{2,i}(v,u) := v_i + u_i$ if $v_i + u_i \in \{0,1\}$ and $t_{2,i}(v,u) := u_i$ otherwise. Let us make these a more intuitive. First, v is used in the protocol to denote the k-vector of the leader from $\{-1,0,1\}^k$. The value of each component i corresponds to the value that must be added to the counter encoded in the i-th components of the agents. When the leader, while still having non-null v, encounters another agent with vector $u \in \{0,1\}^k$, it tries to add to u as much as it can from v. In particular, if $v_i + u_i$ is in $\{0,1\}$, then the sum can be stored in u_i without violating the capacity of the counter. If this is the case, then v_i becomes 0 (so that it is not added a second time to some other agent) and u_i is updated to the sum. On the other hand, if $v_i + u_i \notin \{0,1\}$ (which is for example the case if the leader is trying to subtract 1 from a 0 or to add 1 to a 1) then addition cannot be performed. In this case, both v_i and u_i maintain their values and the leader will try in future interactions to find some other agent whose ith component value will permit addition. It is not hard to verify that these operations are correctly implemented if the leader updates v to $t_1(v,u)$ and the other agent updates u to $t_2(v,u)$. The formal construction is presented in Protocol 3.

Note that in contrast to $\mathcal{S}$, $\mathcal{A}$ reads its input in an arbitrary order, that is, given input x, $\mathcal{A}$ reads a permutation x' of x. L being commutative guarantees that this does not affect the output of $\mathcal{S}$. □

Finally, by combining Corollary 1 and Theorem 5 we obtain the following TM equivalent characterization of the lower bound of Theorem 5.

Corollary 2. $\mathbf{SSPACE}(\log n) \subseteq \mathbf{HAD}$.

Proof. $\mathbf{SSPACE}(\log n) = \mathbf{SCMSPACE}(n) \subseteq \mathbf{HAD}$. □

We next show that any predicate in **HAD** is decidable by a NTM of logarithmic space.

Theorem 6. $\mathbf{HAD} \subseteq \mathbf{SNSPACE}(\log n)$.

Proof. The proof is similar to the one of Theorem 15 in [AAD+06]. Given an AD $\mathcal{A} = (X, Y, Q, I, \omega, \delta, \gamma)$ that stably computes a predicate p in the family of complete interaction graphs, let L_p be the support of p on X^*, that is, the set of strings $x \in X^*$ such that $p(x) = 1$. We present a NTM that decides L_p in $O(\log n)$ space. To accept the input assignment $x \in L_p$, the TM must verify two conditions: that there is a configuration C reachable from the initial configuration corresponding to x in which all agents have output 1, and that there is no configuration C' reachable from C in which some agent has output 0.

Protocol 3 *AD $\mathcal{A}$ simulating CM $\mathcal{S}$*

1: $X = \Sigma$, $Q = [\{l\} \times (Q_p \cup Q_a) \times (X \cup \{-1, 0, 1\}^k \cup \{\varepsilon\})] \cup [(X \cup \bar{X}) \times \{0, 1\}^k] \cup \{q_{accept}, q_{reject}\}$,
2: $I(\sigma) = (\sigma, 0^k)$ for all $\sigma \in X$
3: the initial state of the leader is (l, s_0),
4: δ:

$$\begin{aligned}(l, q), (\sigma, u) &\to (l, q, \sigma), (\bar{\sigma}, u), \text{ if } q \in Q_p \text{ and } \sigma \in X\\ (l', q, v), (\sigma, u) &\to (l', q, t_1(v, u)), (\sigma, t_2(v, u)), \text{ if } t_1(v, u) \neq 0^k\\ &\to (l, q), (\sigma, t_2(v, u)), \text{ otherwise}\end{aligned}$$

5: γ:

$$\begin{aligned}(l, q, \sigma), a &\to (l', M(q, \sigma, b), K(q, \sigma, b)) \text{ if } M(q, \sigma, b) \notin \{q_{accept}, q_{reject}\}\\ &\to (q_{accept}, \cdot) \text{ if } M(q, \sigma, b) = q_{accept}\\ &\to (q_{reject}, \cdot) \text{ if } M(q, \sigma, b) = q_{reject}\\ (l, q), a &\to (l', M(q, b), K(q, b)) \text{ if } q \in Q_a \text{ and } M(q, b) \notin \{q_{accept}, q_{reject}\}\\ &\to (q_{accept}, \cdot) \text{ if } q \in Q_a \text{ and } M(q, b) = q_{accept}\\ &\to (q_{reject}, \cdot) \text{ if } q \in Q_a \text{ and } M(q, b) = q_{reject}\end{aligned}$$

The first condition is verified by guessing and checking a sequence of multiset representations of configurations reaching such a C. Note that due to the symmetry stemming from the interaction graph and from the constant description of the protocols, these representations are polynomial in size (for each state we keep a counter whose value is the number of agents in this state in the current configuration). In addition, the information stored in the absence detector is also of constant size (a bit for each state of the AD). Thus the previous representation requires $O(\log n)$ space (where $n = |x|$) for a single configuration to be stored. The second condition is the complement of a similar reachability condition. This is also in **NSPACE**$(\log n)$ because this class is closed under complement for all space functions $\geq \log n$ (see [Imm88]). □

By Savitch's theorem [Sav70] we have the following corollary:

Corollary 3. **HAD** $\subseteq$ **SSPACE**$(\log^2 n)$.

In summary, we have obtained the following bounds on **HAD** (from Corollaries 2 and 3).

Corollary 4. **SSPACE**$(\log n) \subseteq$ **HAD** $\subseteq$ **SSPACE**$(\log^2 n)$.

8. Similarities to Linear Bounded Automata

In this section, we examine the computational power of the AD model in comparison to the power of Multiset Linear Bounded Automata with Detection (MLBAD) [CVMVM01, Vas08]. To do this, we focus on a more general version of ADs that halt only in the accept case (recognizers). We first prove that nondeterministic ADs of this sort are computationally equivalent to the deterministic ones. Then by exploiting this we will show that recognizing ADs can simulate MLBADs which with the help of one result from the literature will give us that recognizing ADs can compute any language produced by random context grammars.

A *multiset linear bounded automaton with detection* (MLBAD) (see, e.g., [CVMVM01, Vas08]) has a bag able to store a finite multiset and a read-write head which can pick up (read and remove) a symbol from the bag and add at most one symbol to the contents of the bag. Adding a symbol is meant as increasing the number of copies of that symbol by 1. Being in a state, the multiset linear bounded automaton

either reads a symbol which occurs in the bag, changes its state (nondeterministically) and adds or not a symbol to the bag contents or reads the absence of a symbol (detection) from the bag and updates its state (nondeterministically). When the machine stops (no move is possible anymore) in a final state with an empty bag, we say that the input multiset is accepted, otherwise it is rejected. Formally, a multiset linear bounded automaton (shortly, MLBA) is a construct

$$\mathcal{M} = (Q, \Sigma, B, f, q_0, F),$$

where Q and F are the finite sets of states and final states, respectively, Σ and B are the input and the bag alphabets, respectively, $\Sigma \subseteq B$, q_0 is the initial state, and f is the transition mapping from $Q \times B$ into the set of all subsets of $Q \times (B \cup \{\varepsilon\})$, where ε denotes the empty string. As in the case of Multiset Finite Automata (MFAs), a configuration is a pair (q, b), where q is the current state and $b \in \mathbb{N}^B$ is the content of the bag (a vector of nonnegative integers indexed by B). We write

$$(q, b) \to (q', b')$$

if and only if there exists $\sigma \in B$ such that

- $(q', g) \in f(q, \sigma)$, $g \neq \varepsilon$, $b[\sigma] \geq 1$, $b'[\sigma] = b[\sigma] - 1$, $b'[g] = b[g] + 1$, and $b'[c] = b[c]$ for all $c \in B$, $c \notin \{\sigma, g\}$, or
- $(q', \varepsilon) \in f(q, \sigma)$, $t[\sigma] \geq 1$, $b'[\sigma] = b[\sigma] - 1$, and $b'[c] = b[c]$ for all $c \in B$, $c \neq \sigma$.

The reflexive and transitive closure of this operation is denoted by $\stackrel{*}{\to}$. The macroset accepted by A is defined by

$$Rec(M) = \{b \mid (q_0, b) \stackrel{*}{\to} (q_f, \varepsilon) \text{ for some } q_f \in F\}.$$

Note that a MLBAD is by definition nondeterministic. Denote by **LMLBAD** the class of languages accepted by MLBADs. Also denote by **mRC** the class of languages produced by multiset random context grammars. A random context grammar is a semi-conditional grammar where the permitting and forbidden contexts of all productions are subsets of the set of nonterminals [Das04, RS97]. The definition of a multiset random context grammar is a direct extension of the definition of a string random context grammar. Multiset random context grammars contain only non-erasing context-free rules. The following theorem is from [CVMVM01].

Theorem 7 ([CVMVM01])**.** **mRC** $\subseteq$ **LMLBAD**.

We will show now that any MLBAD can be simulated by a recognizing AD. We call an AD $\mathcal{A}$ *recognizing AD* if it only halts in the accept case, that is, given input assignment x it halts if $p(x) = 1$ and just stabilizes otherwise. Let **RAD** denote *the class of all predicates computable by some recognizing AD*. In this notation, we will show that **LMLBAD** $\subseteq$ **RAD**. To simplify the discussion we must first define the nondeterministic version of ADs. A *nondeterministic AD* (*NAD*) is an AD with the transition functions extended as follows:

- $\delta : Q \times Q \to 2^{Q \times Q}$
- $\gamma : Q \times \{0, 1\}^Q \to 2^Q$

A NAD stably accepts its input if some branch of its computation output stabilizes to the value 1, and accepts it if some branch of its computation halts in an accepting state. Denote by **RNAD** the class of predicates computable by some recognizing NAD. The following lemma establishes that nondeterminism does not add computational power to ADs.

Lemma 2. *For any accepting NAD there exists an accepting AD that simulates it (***RNAD** $\subseteq$ **RAD***).*

Proof. Take such an NAD $\mathcal{A}$. We construct an AD $\mathcal{B}$ that simulates $\mathcal{A}$. Define s as the maximum number of nondeterministic transitions over both δ and γ. $\mathcal{B}$ first elects a leader. Then the leader assigns to the agents the numbers $1, 2, \ldots, s$ in a modular fashion (when s is reached, it continues from 1). Denote by $val(u)$ the value assigned to an agent u. In the sequel of the execution the leader plays the role of the coordinator of interactions between the non-leaders. When the leader interacts with some agent u, it collects u's state and marks it. Then it waits for an interaction with some other agent v. Now that the leader knows the pair $(C(u), C(v))$ of the interacting agents' states it must make a nondeterministic choice between those in $\delta(C(u), C(v))$. To do that, it waits for another interaction with some agent w and the nondeterministic choice to be made is $val(w)$ modulo $|\delta(C(u), C(v))|$ if we assume an ordering on the nondeterministic transitions. The nondeterminism of the choice made this way stems from the interaction pattern imposed by the adversary. In this manner, $\mathcal{B}$ performs a search on $\mathcal{A}$'s nondeterministic tree. Similar things happen when the leader interacts with an absence detector. In this way, a branch of $\mathcal{A}$'s tree is followed in a nondeterministic manner. We must also guarantee that all possible paths will eventually be followed. Before taking any step, $\mathcal{B}$ nondeterministically reinitializes the simulation (that is, goes back to the root of the tree). This can be achieved by exploiting the values modulo 2, where 0 results reinitialization and 1 continuation. It is not hard to see by invoking fairness that eventually all paths will be followed, however one can make the simulation more practical if the reinitialization occurs after n^d steps, where d is a large integer constant (this can be achieved by having a protocol similar to Protocol 2 running in parallel and proceeding one step whenever the simulation takes one step. When this protocol halts, the simulation must be reinitialized). Finally, we let $\mathcal{B}$ always output 0 unless an accepting halting state is reached (in which all agents have halted and accepted). It is clear that if $\mathcal{A}$ has an accepting branch, then eventually $\mathcal{B}$ will follow it, it will detect $\mathcal{A}$'s halting and it will also accept and halt. If $\mathcal{A}$ has no accepting branch then $\mathcal{B}$ forever outputs 0. □

Lemma 3. LMLBAD $\subseteq$ RNAD.

Proof. Take any MLBAD $\mathcal{M} = (Q_{\mathcal{M}}, \Sigma, B, f, q_0, F)$. We construct an NAD $\mathcal{A} = (X, Y, Q_{\mathcal{A}}, I, O, \delta, \gamma)$ that simulates $\mathcal{M}$. From Proposition 1 we assume the existence of a unique leader in the initial configuration of $\mathcal{A}$. Moreover, we can assume w.l.o.g. that the leader has no input, since if not it could transfer its input to some non-leader and simulate the existence of some additional agent (treat this agent as two distinct agents). Partition $Q_{\mathcal{A}}$ to two sets Q_l and Q_n containing the leader and non-leader states, respectively. We set $Q_l = Q \cup \{q_{accept}\}$, $X = \Sigma$, $Y = \{0, 1\}$, $Q_n = B \cup \bar{B} \cup \{h\}$, where $\bar{B} := \{\bar{b} \mid b \in B\}$ and h is the halting state, $l := q_0$ (the initial state of the leader), $\omega(q_{accept}) = 1$ and $\omega(q) = 0$ for all $q \in Q_{\mathcal{A}} \backslash \{q_{accept}\}$. Now it is trivial to construct the transition function δ and the detection transition function γ. The state of the leader is the state of the automaton $\mathcal{M}$ and the states of the non-leaders play the role of the symbols in the bag. δ has effective interactions of the form $Q_l \times Q_n \to Q_l \times Q_n$ and works as the transition mapping f of $\mathcal{M}$. Whenever a symbol (state in $B \subseteq Q_n$) is erased by f, δ maps it to its corresponding state in $\bar{B}$. γ maps the leader to the q_{accept} halting state whenever states from B are absent and the leader is in a state in $F \subseteq Q_l$. Once the q_{accept} state is no longer absent (that is, the leader and thus $\mathcal{M}$ has accepted the input) γ maps all states from $\bar{B}$ (that is, the non-leaders) to the halting state h. □

Thus, from Theorem 7 and Lemmas 2 and 3, we have that

Theorem 8. mRC $\subseteq$ RAD.

In words, this says that recognizing ADs can compute any language produced by multiset random context grammars.

9. Conclusions

In this work we proposed the CTS model, a new extension of the PP model of Angluin *et al.* that additionally assumes the existence of a cover-time service. By reduction to the absence detector oracle model we were able to investigate and almost completely characterize the computational power of the new

model. The introduced global knowledge enables CTSs to perform halting computations, a feature that is missing from the original PP model. We explored the properties and the computability of the new model and focused more on halting computations. We showed that all predicates in $\mathbf{SSPACE}(\log n)$ are also in **HAD** and that the latter is a subset of $\mathbf{SSPACE}(\log^2 n)$. Finally, the comparison of the computability of our model to that of MLBADs lead us to the result that recognizing ADs can compute any language produced by random context grammars.

Many interesting questions remain open. The bounds given in this work for halting ADs are not tight. An exact characterization of **HAD** is still elusive. In addition, what happens in the case where the detector does not always correctly detect or in the case where it does not have full knowledge (e.g. may predict a fraction of the absent nodes)? Do the protocols presented here work correctly in the case of a weak, a faulty, or even an adversarial detector? Moreover, how is the computability of graph properties of the interaction graph affected by the absence detector's presence? Finally, can one simplify the proof of the upper bound of PPs [AAER07] by simulating them by a one-way 1-CM or by a nondeterministic pushdown automaton?

Acknowledgements. We would like to particularly thank James Aspnes for bringing to our attention the similarity of our model to the eventual leader detector Ω of [FJ06]. We would also like to thank some anonymous reviewers for their useful comments that have helped us improve our work.

References

[AAC+05] D. Angluin, J. Aspnes, M. Chan, M. J. Fischer, H. Jiang, and R. Peralta. Stably computable properties of network graphs. In *Distributed Computing in Sensor Systems: First IEEE International Conference DCOSS*, volume 3560 of *LNCS*, pages 63–74. Springer-Verlag, June 2005.

[AAD+06] D. Angluin, J. Aspnes, Z. Diamadi, M. J. Fischer, and R. Peralta. Computation in networks of passively mobile finite-state sensors. *Distributed Computing*, pages 235–253, March 2006.

[AAE08] D. Angluin, J. Aspnes, and D. Eisenstat. Fast computation by population protocols with a leader. *Distributed Computing*, 21[3]:183–199, 2008.

[AAER07] D. Angluin, J. Aspnes, D. Eisenstat, and E. Ruppert. The computational power of population protocols. *Distributed Computing*, 20[4]:279–304, November 2007.

[BBCK10] J. Beauquier, J. Burman, J. Clement, and S. Kutten. On utilizing speed in networks of mobile agents. In *Proceeding of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, PODC '10, pages 305–314, New York, NY, USA, 2010. ACM.

[CMN+11] I. Chatzigiannakis, O. Michail, S. Nikolaou, A. Pavlogiannis, and P. G. Spirakis. Passively mobile communicating machines that use restricted space. *Theor. Comput. Sci.*, 412[46]:6469–6483, 2011.

[CVMVM01] E. Csuhaj-Varjú, C. Martin-Vide, and V. Mitrana. Multiset automata. In *Proceedings of the Workshop on Multiset Processing: Multiset Processing, Mathematical, Computer Science, and Molecular Computing Points of View*, WMP '00, pages 69–84, London, UK, 2001. Springer-Verlag.

[Das04] J. Dassow. Grammars with regulated rewriting. In *Formal Languages and Applications*, pages 249–273. Springer, 2004.

[Edi86] J. L. Edighoffer. *Distributed, replicated computer bulletin board service.* PhD thesis, Stanford, CA, USA, 1986. UMI order no. GAX86-19742.

[FJ06] M. Fischer and H. Jiang. Self-stabilizing leader election in networks of finite-state anonymous agents. In *Proceedings of the 10th international conference on Principles of Distributed Systems*, OPODIS'06, pages 395–409, Berlin, Heidelberg, 2006. Springer-Verlag.

[FMR68] P. C. Fischer, A. R. Meyer, and A. L. Rosenberg. Counter machines and counter languages. *Mathematical Systems Theory*, 2[3]:265–283, 1968.

[GR09] R. Guerraoui and E. Ruppert. Names trump malice: Tiny mobile agents can tolerate byzantine failures. In *36th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 5556 of *Lecture Notes in Computer Science*, pages 484–495. Springer-Verlag, 2009.

[GS66] S. Ginsburg and E. H. Spanier. Semigroups, presburger formulas, and languages. *Pacific Journal of Mathematics*, 16:285–296, 1966.

[Iba04] O. H. Ibarra. On the computational complexity of membrane systems. *Theor. Comput. Sci.*, 320:89–109, June 2004.

[Imm88] N. Immerman. Nondeterministic space is closed under complementation. *SIAM J. Comput.*, 17[5]:935–938, 1988.

[MCS11a] O. Michail, I. Chatzigiannakis, and P. G. Spirakis. Mediated population protocols. *Theor. Comput. Sci.*, 412:2434–2450, May 2011.

[MCS11b] O. Michail, I. Chatzigiannakis, and P. G. Spirakis. *New Models for Population Protocols.* N. A. Lynch (Ed), Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool, 2011.

[MCS12] O. Michail, I. Chatzigiannakis, and P. G. Spirakis. Terminating population protocols via some minimal global knowledge assumptions. In A. Richa and C. Scheideler, editors, *Stabilization, Safety, and Security of Distributed Systems*, volume 7596 of *Lecture Notes in Computer Science*, pages 77–89. Springer Berlin Heidelberg, 2012.

[Min61] M. L. Minsky. Recursive unsolvability of post's problem of "tag" and other topics in theory of turing machines. *The Annals of Mathematics*, 74[3]:437–455, Nov. 1961.

[RS97] G. Rozenberg and A. Salomaa, editors. *Handbook of formal languages, vol. 1: word, language, grammar.* Springer-Verlag New York, Inc., 1997.

[Sav70] W. J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *J. Comput. System Sci.*, 4[2]:177–192, 1970.

[Sip06] M. Sipser. *Introduction to the Theory of Computation, Second Edition, International Edition.* Thomson Course Technology, 2006.

[Vas08] G. Vaszil. Multiset grammars, multiset automata, and membrane systems. In *Colloquium on the Occasion of the 50th Birthday of Victor Mitrana*, pages 1–10. Springer-Verlag, 2008.