Counter Machines and Distributed Automata

Olivier Carton, Bruno Guillon, Fabian Reiter

Counter Machines and Distributed Automata

A Story about Exchanging Space and Time

Olivier Carton[1], Bruno Guillon[2], and Fabian Reiter[3]

[1] IRIF, Université Paris Diderot, France
olivier.carton@irif.fr,
[2] Department of Computer Science, University of Milan, Italy
guillon.bruno+cs@gmail.com,
[3] LSV, Université Paris-Saclay, France
fabian.reiter@gmail.com

Abstract. We prove the equivalence of two classes of counter machines and one class of distributed automata. Our counter machines operate on finite words, which they read from left to right while incrementing or decrementing a fixed number of counters. The two classes differ in the extra features they offer: one allows to copy counter values, whereas the other allows to compute copyless sums of counters. Our distributed automata, on the other hand, operate on directed path graphs that represent words. All nodes of a path synchronously execute the same finite-state machine, whose state diagram must be acyclic except for self-loops, and each node receives as input the state of its direct predecessor. These devices form a subclass of linear-time one-way cellular automata.

1 Introduction

Space and time are the two standard resources for solving computational problems. Typically, the more of these resources a computing device has at its disposal, the harder the problems it can solve. In this paper, we consider two types of devices whose usages of space and time turn out to be dual to each other.

On the one hand, we look at *counter machines*, which can use a lot of space. In the way we define them here, these devices act as language recognizers. Just like classical finite automata, they take a finite word as input, read it once from left to right, and then decide whether or not to accept that word. However, in addition to having a finite-state memory, such a machine also has a fixed number of counters, which can store arbitrarily large integer values (and are initially set to zero). The machine has read access to those values up to some fixed threshold. Whenever it processes a symbol of the input word, it can deterministically change its internal state and simultaneously update each counter x to a new value that is expressed as the sum of values of several counters $y_1, \dots, y_n$ and a constant c. (Every update consumes an input symbol, i.e., there are no epsilon transitions.) Our main concern are two special cases of this model: *sumless* counter machines, which can increment, decrement and copy counter values but not sum them up, and *copyless* counter machines, which can compute arbitrary sums but not use

the same counter more than once per update step. Both of these conditions entail that counter values can grow only linearly with the input length, and, as we will see, they yield in fact the same expressive power.

On the other hand, we look at *distributed automata*, which are devices that can use a lot of time. For our purposes, they also act as language recognizers, but their input word is given in form of a directed path graph whose nodes are labeled with the symbols of the word (such that the first symbol is on the source node). To run a distributed automaton on such a path, we first place a copy of the automaton on each node and initialize it to a state that may depend on the node's label. Then, the execution proceeds in an infinite sequence of synchronous rounds, where each node determines its next state as a function of its own current state and the current state of its incoming neighbor (i.e., the node to its left). Altogether, there are only a finite number of states, some of which are considered to be accepting. The automaton acts as a semi-decider and accepts the input word precisely if the last node of the path visits an accepting state at some point in time. Here, we are particularly interested in those distributed automata whose state diagram does not contain any directed cycles except for self-loops; we call them *quasi-acyclic*. They have the property that all nodes stop changing their state after a number of rounds that is linear in the length of the input word. Therefore, if a quasi-acyclic automaton accepts a given word, then it does so in linear time.

To sum up, we have a sequential model and a distributed model that consume space and time in opposite ways: given an input word of length n, a sumless or copyless counter machine uses time n and space linear in n, whereas a quasi-acyclic distributed automaton uses space n and linear time.[4] The purpose of this paper is to show that there really is a duality between the space of one model and the time of the other. In fact, we will prove that the two models are expressively equivalent. Besides being of independent interest, this result also relates to three separate branches of research.

Cellular automata. In theoretical computer science, cellular automata are one of the oldest and most well-known models of parallel computation (see, e.g., [8]). They consist of an infinite array whose cells are each in one of a finite number of states and evolve synchronously according to a deterministic local rule. In this regard, a distributed automaton over a labeled directed path can be viewed as a *(one-dimensional) one-way cellular automaton* with some permanent boundary symbol delimiting the input word [2]. This model has been studied as language recognizer, and differences between real time (i.e., time n for an input of length n) and linear time (i.e., time in $\mathcal{O}(n)$) have been highlighted – see [20] for a survey on language recognition by cellular automata. As explained below, our work initially takes its motivation from distributed computing, hence the choice of "distributed automata" rather than "cellular automata". Nevertheless, the results presented here may be viewed in terms of languages recognized by one-way

[4] We assume that counter machines store the values of their counters in unary encoding, and we measure the space usage of a distributed automaton by the number of nodes.

cellular automata, with the technical difference that the input words are reversed with respect to the usual definition of one-way cellular automata.[5]

A long-standing open problem in this area is the question whether or not one-way cellular automata working in unrestricted time can recognize every language in DSpace(n), i.e., the class of languages accepted by deterministic Turing Machines working in linear space. The latter actually coincides with the class of languages accepted by *(two-way) cellular automata* (see, e.g., [9]). By relating a subclass of one-way cellular automata with counter machines working in linear space, our results might be considered as a new approach towards describing the expressiveness of one-way cellular automata. Our contribution concerns a class of (reversed) languages included in the class of languages recognized by linear-time one-way cellular automata. Indeed, the quasi-acyclic restriction on distributed automata corresponds to a special case of one-way cellular automata in which each cell may change its state only a bounded number of times during an execution [21]. This is a strict subcase of one-way cellular automata working in linear time, as can be deduced, for instance, from [22, Prop 3]. More precisely, quasi-acyclic distributed automata correspond to *freezing cellular automata*, which are cellular automata in which each state change of a cell is increasing according to some fixed order on the states [5]. In particular, freezing cellular automata have *bounded communication* [10]. Conversely, as observed in [5], each one-way cellular automaton with bounded communication can be easily transformed into an equivalent freezing one.

Counter machines. A classical result due to Minsky states that Turing machines have the same computational power as finite-state machines equipped with two integer counters that can be arbitrarily often incremented, decremented, and tested for zero [15]. Such devices are often referred to as *Minsky machines*. Their Turing completeness led Fischer, Meyer, and Rosenberg to investigate the space and time complexities of machines with an arbitrary number of counters, viewed as language recognizers. In [4], they paid particular attention to *real-time machines*, where the number of increments and decrements per counter is limited by the length of the input word. Among many other things, they showed that increasing the number of counters strictly increases the expressive power of real-time machines, and that those devices become even more powerful if we equip them with the additional ability to reset counters to zero (in a single operation). Over four decades later, Petersen proved in [17] that for machines with a single counter, real time with reset is equivalent to linear time without reset, and that for machines with at least two counters, linear time is strictly more expressive. A further natural extension of real-time machines is to allow values to be copied from one counter to another (again, in a single operation). In [3], Dymond showed that real-time machines with copy can be simulated by linear-time machines without copy.

[5] Contrary to distributed automata, one-way cellular automata are usually represented with information transiting from right to left, that is, a cell receives the state from its right neighbor and the leftmost cell decides acceptance of the input, see, e.g., [9].

The general version of the counter machines defined in this paper can also be seen as an extension of the real-time machines of Fischer, Meyer, and Rosenberg. In addition to the reset and copy operations, we allow counter values to be summed up. Our formal notation takes inspiration from *cost register automata*, which were introduced by Alur et al. in [1]. Moreover, the concept of copylessness is borrowed from there. The authors are not aware of any previous work dealing with the specific counter machines defined in this paper. However, it follows from [3, Thm 2.1] and our main result that sumless and copyless counter machines form a subclass of the linear-time counter machines defined in [4].

Distributed computing and logic. The original motivation for this paper comes from a relatively recent project that aims to develop a form of descriptive complexity [7] for distributed computing [13, 16]. In that context, distributed automata are regarded as a class of weak distributed algorithms, for which it is comparatively easy to obtain characterizations by logical formulas. Basically, these automata are the same as those described above, except that they can run on arbitrary directed graphs instead of being confined to directed paths. In order to make this possible, each node is allowed to see the set of states of its incoming neighbors (without multiplicity) instead of just the state of its left neighbor. On graphs with multiple edge relations $(E_1, \ldots, E_r)$, the nodes see a separate set for each relation. The first result in this direction was obtained by Hella et al. in [6], where they showed that distributed automata with constant running time are equivalent to a variant of basic modal logic on graphs. The link with logic was further strengthened by Kuusisto in [11], where a logical characterization of unrestricted distributed automata was given in terms of a modal-logic-based variant of Datalog. Then, Reiter showed in [18] that the least fixpoint fragment of the modal μ-calculus captures an asynchronous variant of quasi-acyclic distributed automata. Motivated by these connections to modal logic, a field at the frontier between decidability and undecidability, the emptiness problem for distributed automata was investigated in [12]. The authors observed that the problem is undecidable for arbitrary automata on directed paths (which implies undecidability on arbitrary graphs), as well as for quasi-acyclic automata on arbitrary graphs. But now, the main result of the present paper supersedes both of these findings: since, by a simple reduction from the halting problem for Minsky machines, the emptiness problem for sumless and copyless counter machines is undecidable, we immediately obtain that the problem is also undecidable for quasi-acyclic distributed automata on directed paths. It must, however, be stressed that such undecidability results have been known for a long time within the community of cellular automata. For instance, it was shown by Seidel in [19] that the emptiness problem for real-time one-way cellular automata is undecidable (see also [14]). This was later strengthened by Kutrib and Malcher, who proved in [10] that the problem remains undecidable even if we restrict ourselves to automata with bounded communication. Thereby they provided an undecidability result that is stronger than our corollary, given that quasi-acyclic distributed automata do not necessarily work in real time.

Outline. The remainder of the paper is devoted to proving our main result:

Theorem 1. *The following three classes of devices are effectively equivalent.*

1. *Copyless counter machines on nonempty finite words.*
2. *Sumless counter machines on nonempty finite words.*
3. *Quasi-acyclic distributed automata on pointed directed paths.*

All the necessary definitions are introduced in Section 2. The statement then follows from several translations provided in the subsequent sections: we have "$1 \to 2$" by Proposition 7 in Section 3, then "$2 \to 3$" by Propositions 9 and 10 in Sections 3 and 4, and finally "$3 \to 1$" by Proposition 11 in Section 5. We conclude with a detailed summary of these translations in Section 6 (see Figure 4) and some perspectives for future work.

2 Preliminaries

We denote the set of nonnegative integers by $\mathbb{N} = \{0, 1, 2, \dots\}$, the set of positive integers by $\mathbb{N}_+ = \mathbb{N} \setminus \{0\}$, and the set of integers by $\mathbb{Z} = \{\dots, -1, 0, 1, \dots\}$. The power set of any set S is written as 2^S. Furthermore, for values $m, n \in \mathbb{Z}$ such that $m \leq n$, we define the interval notation $[m : n] \coloneqq \{i \in \mathbb{Z} \mid m \leq i \leq n\}$ and the cutoff function cut_m^n, which truncates its input to yield a number between m and n. The latter is formally defined as $\mathrm{cut}_m^n \colon \mathbb{Z} \to [m : n]$ such that $\mathrm{cut}_m^n(i)$ is equal to m if $i < m$, to i if $m \leq i \leq n$, and to n if $i > n$.

Let Σ be a finite set of symbols. A *word* over Σ is a finite sequence $w = a_1 \dots a_n$ of symbols in Σ. We write $|w|$ for the length of w and Σ^+ for the set of all nonempty words over Σ. A *language* over Σ is a subset of Σ^+.

Example 2 (running). As a running example, we consider the language L of nonempty words in $\{a, b, c\}^+$ whose prefixes all have at least as many a's as b's and at least as many b's as c's: $L = \{w \mid \text{for every prefix } u \text{ of } w,\ |u|_a \geq |u|_b \geq |u|_c\}$, where $|w|_\sigma$ denotes the number of σ's in w, for $\sigma \in \Sigma$. For instance, the words *aaabbc* and *aabbac* belong to L, whereas the word *abacac* does not.

2.1 Counter machines

Let X be a finite set of counter variables and h be a positive integer. We denote by $\Xi(X, h)$ the set of *counter expressions* over X and h generated by the grammar $e ::= x + e \mid c$, where $x \in X$ and $c \in [-h : h]$. An *update function* for X given h is a map $\xi \in \Xi(X, h)^X$ that assigns a counter expression to each counter variable.

Definition 3 (Counter Machine). *A* k-counter machine *with* h-access *over the alphabet* Σ *is a tuple* $M = (P, X, p_0, \tau, H)$*, where* P *is a finite set of states,* X *is a set containing precisely* k *distinct counter variables,* $p_0 \in P$ *is an initial state,* $\tau \colon P \times [-h : h]^X \times \Sigma \to P \times \Xi(X, h)^X$ *is a transition function, and* $H \subseteq P$ *is a set of accepting states.*

Such a counter machine "knows" the exact value of each counter that lies between the thresholds $-h$ and h; values smaller than $-h$ are "seen" as $-h$, and similarly, values larger than h are "seen" as h. Furthermore, it has the ability to add (in a single operation) constants between $-h$ and h to its counters. The technical details are explained in the following.

Let $M = (P, X, p_0, \tau, H)$ be a counter machine with h-access over the alphabet Σ, and let $w = a_1 \dots a_n$ be a word in Σ^+. A *valuation* of X is a map $\nu \in \mathbb{Z}^X$ that assigns an integer value to each counter variable $x \in X$. The initial valuation is $\nu_0 = \{x \mapsto 0 \mid x \in X\}$. Any valuation $\nu \in \mathbb{Z}^X$ gives rise to an *extended valuation* $\hat{\nu} \in \mathbb{Z}^{\Xi(X,h)}$, which assigns values to counter expressions in the natural way, i.e., $\hat{\nu}(c) = c$ and $\hat{\nu}(x + e) = \nu(x) + \hat{\nu}(e)$, for $c \in [-h\,{:}\,h]$ and $x \in X$. A *memory configuration* of M is a tuple $C = (p, \nu) \in P \times \mathbb{Z}^X$. The *run* of M on w is the sequence of memory configurations $R = (C_0, \dots, C_n)$ such that $C_0 = (p_0, \nu_0)$, and if $C_l = (p, \nu)$ and $\tau(p, \operatorname{cut}_{-h}^{+h} \circ \nu, a_{l+1}) = (p', \xi)$, then $C_{l+1} = (p', \hat{\nu} \circ \xi)$. The machine M *accepts* the word w if it terminates in an accepting state, i.e., if $C_n \in H \times \mathbb{Z}^X$. The *language* of M (or language *recognized* by M) is the set of all words accepted by M.

We call an update function $\xi \in \Xi(X, h)^X$ *sumless* if it does not allow sums of multiple counter variables, i.e., if for all $x \in X$, the expression $\xi(x)$ is either c or $y + c$, for some $c \in [-h\,{:}\,h]$ and $y \in X$. Note that such an update function allows us to copy the value of one counter to several others, since the same counter variable y may be used in more than one expression $\xi(x)$. On the other hand, ξ is *copyless* if every counter variable $y \in X$ occurs in at most one expression $\xi(x)$, and at most once in that expression. (However, sums of distinct variables are allowed.) By allowing each counter to be used only once per step, this restriction ensures that the sum of all counter values can grow at most linearly with the length of the input word. A counter machine M is called *sumless* or *copyless* if its transition function τ makes use only of sumless or copyless update functions, respectively. As shown in this paper, the two notions are expressively equivalent.

Example 4 (running). The language L from Example 2 is accepted by the sumless and copyless 2-counter machine $M = (\{p, r\}, \{x, y\}, p, \tau, \{p\})$, with τ defined by:

$$\tau\big(s, (c_x, c_y), \sigma\big) = \begin{cases} \big(p,\, \{x \coloneqq x + 1,\ y \coloneqq y\ \ \}\big) & \text{if } s = p \qquad\quad\ \text{and } \sigma = a, \\ \big(p,\, \{x \coloneqq x - 1,\ y \coloneqq y + 1\}\big) & \text{if } s = p,\ c_x > 0 \text{ and } \sigma = b, \\ \big(p,\, \{x \coloneqq x,\ \ y \coloneqq y - 1\}\big) & \text{if } s = p,\ c_y > 0 \text{ and } \sigma = c, \\ \big(r,\, \{x \coloneqq x,\ \ y \coloneqq y\ \ \}\big) & \text{otherwise,} \end{cases}$$

where c_x and c_y denote the values $\operatorname{cut}_{-1}^{+1} \circ \nu(x)$ and $\operatorname{cut}_{-1}^{+1} \circ \nu(y)$ respectively. Intuitively, M uses the counter x to compare the number of a's with those of b's, and the counter y to compare the number of b's with those of c's. The counter x (respectively y) is incremented each time the letter a (respectively b) is read and it is decremented each time the letter b (respectively c) is read. When a counter with value 0 has to be decremented, the machine enters the rejecting sink state r.

2.2 Distributed automata

Let Σ be a finite set of symbols. A *Σ-labeled directed graph*, abbreviated *digraph*, is a structure $G = (V, E, \lambda)$, where V is a finite nonempty set of nodes, $E \subseteq V \times V$ is a set of directed edges, and $\lambda\colon V \to \Sigma$ is a labeling function that assigns a symbol of Σ to each node. Isomorphic digraphs are considered to be equal. If v is a node in V, we call the pair (G, v) a *pointed digraph* with *distinguished node* v. Moreover, if uv is an edge in E, then u is called an *incoming neighbor* of v.

A *directed path*, or *dipath*, is a digraph $G = (V, E, \lambda)$ that has a distinct *last node* v_{last} such that each node v in V has at most one incoming neighbor and exactly one way to reach v_{last} by following the directed edges in E. A *pointed dipath* is a pointed digraph (G, v_{last}) that is composed of a dipath and its last node. We shall identify each word $w \in \Sigma^+$ with the pointed Σ-labeled dipath of length $|w|$ whose nodes are labeled with the symbols of w, i.e., the word $a_1 a_2 \dots a_n$ will be identified with the pointed dipath $(a_1) \to (a_2) \to \cdots \to (a_n)$.

We first give a rather general definition of distributed automata on arbitrary digraphs, and then slightly modify our notation for the special case of dipaths.

Definition 5 (Distributed Automaton). *A (finite)* distributed automaton *over Σ-labeled digraphs is a tuple $A = (Q, \delta_0, \delta, F)$, where Q is a finite set of states, $\delta_0\colon \Sigma \to Q$ is an initialization function, $\delta\colon Q \times 2^Q \to Q$ is a transition function, and $F \subseteq Q$ is a set of accepting states.*

Let $A = (Q, \delta_0, \delta, F)$ be a distributed automaton over Σ-labeled digraphs, and let $G = (V, E, \lambda)$ be a corresponding digraph. The (synchronous) *run* of A on G is an infinite sequence $\rho = (\rho_0, \rho_1, \rho_2, \dots)$ of maps $\rho_t\colon V \to Q$, called *configurations*, which are defined inductively as follows, for $t \in \mathbb{N}$ and $v \in V$:

$$\rho_0(v) = \delta_0(\lambda(v)) \qquad \text{and} \qquad \rho_{t+1}(v) = \delta\big(\rho_t(v),\, \{\rho_t(u) \mid uv \in E\}\big).$$

For $v \in V$, the automaton A *accepts* the pointed digraph (G, v) if v visits an accepting state at some point in the run ρ of A on G, i.e., if there exists $t \in \mathbb{N}$ such that $\rho_t(v) \in F$.

The above definition could be easily extended to cover r-relational digraphs, i.e., digraphs with r edge relations $E_1, \dots, E_r$, for some $r \in \mathbb{N}_+$. It suffices to choose a transition function of the form $\delta\colon Q \times (2^Q)^r \to Q$, thereby allowing the nodes to see a separate set of states for each of the r relations. With this, one could easily simulate two-way (one-dimensional) or even higher-dimensional cellular automata. However, for our purposes, a single edge relation is enough.

A *trace* of a distributed automaton $A = (Q, \delta_0, \delta, F)$ is a finite nonempty sequence $q_1, \dots, q_n$ of states in Q such that for $1 \leq i < n$, we have $q_i \neq q_{i+1}$ and $\delta(q_i, S_i) = q_{i+1}$ for some $S_i \subseteq Q$. We say that A is *quasi-acyclic* if its set of traces is finite. In other words, A is quasi-acyclic if its state diagram does not contain any directed cycles, except for self-loops. In this case, we will refer to $\ell = \max\{n \mid A \text{ has a trace of length } n\}$ as the *maximum trace length* of A. Furthermore, a quasi-acyclic automaton A is said to have *at most* $(k+1)$ *loops per trace* if $(k+1) = \max\{n \mid A \text{ has a trace containing } n \text{ looping states}\}$. Here,

a *looping state* is a state $q \in Q$ such that $\delta(q, S) = q$ for some $S \subseteq Q$. Notice that every trace of a quasi-acyclic automaton must end in a looping state, since transition functions are defined to be total. (This is why we write "$k + 1$".)

In this paper, we regard distributed automata as word acceptors, and thus we restrict their input to dipaths. Therefore, in our particular context, a distributed automaton is the same thing as a (one-dimensional, reversed) one-way cellular automaton (see, e.g., [20]). This allows us to simplify our notation: transition functions will be written as $\delta\colon Q_\emptyset \times Q \to Q$, where $Q_\emptyset$ is a shorthand for $Q \cup \{\emptyset\}$. A node whose left neighbor's current state is p and whose own current state is q will transition to the new state $\delta(p, q)$; if there is no left neighbor, p has to be replaced by $\emptyset$. Note that we have reversed the order of p and q with respect to their counterparts in Definition 5, as this seems more natural when restricted to dipaths. We say that the *language* of A (or language *recognized* by A) is the set of words, seen as pointed dipaths, accepted by A.

As usual, we say that two devices (i.e., counter machines or distributed automata) are *equivalent* if they recognize the same language.

Example 6 (running). We describe here a distributed automaton A that accepts the language L from Example 2, regarded as a set of dipaths. To this end, we first reformulate the property that every prefix contains at least as many a's as b's and at least as many b's as c's: it is equivalent to the existence of an injective mapping from nodes labeled by b to nodes labeled by a and from nodes labeled by c to nodes labeled by b such that each node can only be mapped to some (possibly indirect) predecessor to its left. Our automaton A implicitly creates such an injective mapping by forwarding all a's and b's to the right until they are "consumed" by matching b's and c's.

The device uses two tracks that may contain the symbols a, b, or "$-$", i.e., its states are pairs in $\{a, b, -\} \times \{a, b, -\}$. Initially, a node labeled by the letter $\sigma \in \Sigma = \{a, b, c\}$ is in the state (x, y), where x is equal to "$-$" if $\sigma = a$, to a if $\sigma = b$, and to b if $\sigma = c$, and y is equal to "$-$" if $\sigma = c$, and to σ otherwise. The first track is the *expectation track*; its content indicates which letter the node should receive from its left neighbor in order to eventually accept (the special symbol "$-$" means "nothing is expected"). The second track is the *communication track*; its content is sent to the node's right neighbor (the special symbol "$-$" means "nothing is sent"). If a node is expecting a letter σ and receives σ from its left neighbor, then that node switches to the state $(-, -)$. This means that the node is no longer expecting any letter and does not transmit anything to its right neighbor (since the letter σ has already been "consumed"). Additionally, A uses two special states $\bot$ and $\top$, which propagate errors and acceptance, respectively. When a node enters one of these two states, it stays in that state forever. An error always propagates to the right neighbor. In contrast, a node enters state $\top$ if it receives an acceptance message from the left (i.e., it receives $\top$ or $\emptyset$) and its expectation has been fulfilled.

Figure 1 shows the runs of A on the dipaths *aabbac* (accepted) and *abacac* (rejected). Observe that A is not quasi-acyclic, since, for instance, the last node of the dipath *aabbac* switches from state $(-, -)$ to $(-, b)$ and then again to $(-, -)$.

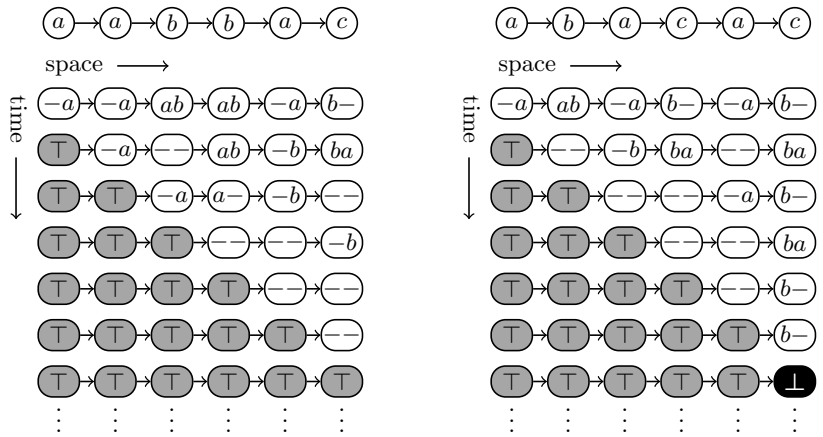


Fig. 1. The runs of the distributed automaton from Example 6 on *aabbac* and *abacac*.

3 Translating between counter machines

We start with the translation from copyless to sumless counter machines, followed by two constructions that allow us, in some cases, to focus on counters with non-negative values and 1-access.

Proposition 7. *For every copyless k-counter machine with h-access, we can effectively construct an equivalent sumless (2^k)-counter machine with $(k \cdot h)$-access.*

Proof (sketch). The idea is simply to introduce a dedicated counter for each subset of counters Y of the original machine M, and use this dedicated counter to store the sum of values of the counters in Y. Call this sum the *value* of Y. Since M is copyless, it uses each of its counters at most once in any update function ξ. Therefore, the next value of Y with respect to ξ can be expressed in terms of the current value of some other subset Y' and a constant between $-|Y| \cdot h$ and $|Y| \cdot h$. This allows us to derive from ξ a sumless update function ξ' that operates on subsets of counters and uses constants in $[-kh : kh]$. □

Sometimes it is helpful to assume that a counter machine never stores any negative values in its counters. For copyless and sumless machines, this does not lead to a loss of generality. We only need the statement for sumless machines, but in fact Proposition 7 implies that it also holds for copyless machines (at the cost of increasing the number of counters).

Proposition 8. *For every sumless k-counter machine with h-access, we can effectively construct an equivalent machine that is also sumless with k counters and h-access, but whose counters never store any negative values.*

Proof (sketch). It suffices to represent each counter x of the original machine in such a way that the absolute value of x is stored in a counter and its sign is

retained in finite-state memory. As the machine is sumless, we do not have to deal with the issue of computing the sum of a positive and a negative counter value. □

In Definition 3, we have introduced counter machines with h-access, for some arbitrary $h \in \mathbb{N}_+$. This simplifies some of our proofs, but we could have imposed $h = 1$ without losing any expressive power. The following proposition states this in full generality, although the (easier to prove) restriction to sumless machines would be sufficient to establish our main result.

Proposition 9. *For every k-counter machine M with h-access, we can effectively construct an equivalent $(h \cdot k)$-counter machine M' with 1-access. If M is copyless or sumless, then so is M'. Moreover, if M is sumless, M' requires only k counters.*

Proof (sketch). The key idea is that M' represents each counter x of M by h counters $x_0, \dots, x_{h-1}$ over which the value of x is distributed as uniformly as possible. That is, the value of x is equal to the sum of the values of $x_0, \dots, x_{h-1}$, and any two of the latter values differ by at most 1. If M is sumless, there is a simpler way: it suffices to represent x by a single counter storing the value of x divided by h, and to keep track of the remainder in the finite-state memory. □

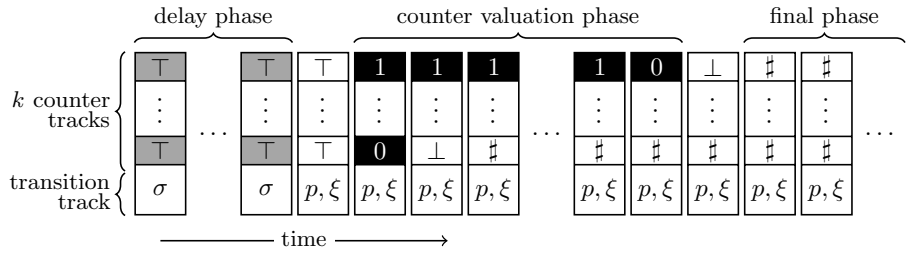
4 From counter machines to distributed automata

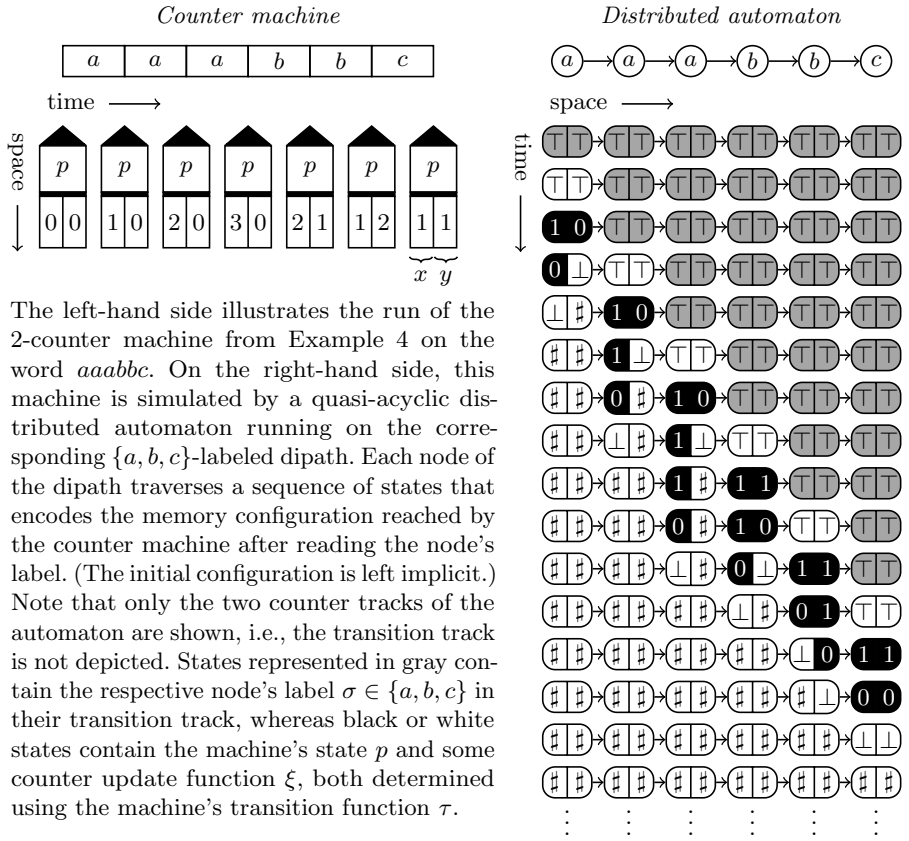
Next, we present the translation from sumless counter machines to quasi-acyclic distributed automata.

Proposition 10. *For every sumless k-counter machine with 1-access, we can effectively construct an equivalent quasi-acyclic distributed automaton with at most $(k + 2)$ loops per trace.*

Proof (sketch). Our construction uses classical techniques from cellular automata theory, similar to simulations of finite automata (see, e.g., [9, Lem 11]) and counter machines (see, e.g., [5, Thm 1]) by one-way cellular automata. Let us point out that, contrary to the construction in [5], we allow the copy operation on counters here. An example of the simulation is shown in Figure 2.

We now explain the main idea. On an input dipath corresponding to some word w, the sequence of states traversed by our distributed automaton at the i-th node is an encoding of the memory configuration (p, ν) that is reached by the simulated counter machine after reading the i-th symbol of w. (The initial configuration is not encoded.) This sequence of states is of the following form:





The left-hand side illustrates the run of the 2-counter machine from Example 4 on the word *aaabbc*. On the right-hand side, this machine is simulated by a quasi-acyclic distributed automaton running on the corresponding $\{a, b, c\}$-labeled dipath. Each node of the dipath traverses a sequence of states that encodes the memory configuration reached by the counter machine after reading the node's label. (The initial configuration is left implicit.) Note that only the two counter tracks of the automaton are shown, i.e., the transition track is not depicted. States represented in gray contain the respective node's label $\sigma \in \{a, b, c\}$ in their transition track, whereas black or white states contain the machine's state p and some counter update function ξ, both determined using the machine's transition function τ.

Fig. 2. Simulating a counter machine with a distributed automaton to prove Proposition 10. The depicted counter machine is the same as in Example 4. But the resulting automaton differs from the one given in Example 6. In particular, it is quasi-acyclic.

Here, each rectangular block represents a state of the distributed automaton. The symbol σ corresponds to the node's label and ξ is the update function that has been used to enter the memory configuration (p, ν). Counter values are encoded in unary, i.e., the value $\nu(x)$ of a counter x is the number of 1's on the associated counter track. (By Proposition 8, we assume the values are never negative.)

The delay phase is used to leave enough time for information to transit. We increase it by 2 at each position, in order to be able to compute decrementation. Hence, at the i-th node, the delay phase lasts for $(2i-1)$ rounds. (This corresponds to the gray states in Figure 2.)

Since each counter track associated with a counter x contains a sequence of the form $\top^{2i} 1^{\nu(x)} 0 \bot \sharp^{\omega}$, we are guaranteed that the simulating distributed automaton has at most $(k+2)$ loops per trace. □

5 From distributed automata to counter machines

As the last piece of the puzzle, we now show how to convert a quasi-acyclic distributed automaton into an equivalent copyless counter machine.

Proposition 11. *For every quasi-acyclic distributed automaton with at most* $(k+1)$ *loops per trace and maximum trace length* ℓ*, we can effectively construct an equivalent copyless* k*-counter machine with* ℓ*-access.*

Proof (sketch). Basically, after our counter machine M has read the i-th symbol of the input word w, its memory configuration will represent the sequence of states traversed by the simulated distributed automaton A at the i-th node of the dipath corresponding to w. This exploits the quasi-acyclicity of A to represent the infinite sequence of states traversed by a node as a finite sequence of pairs in $Q \times (\mathbb{N}_+ \cup \{\infty\})$, where values other than 1 and ∞ are stored in the counters. An example illustrating the construction is provided in Figure 3.

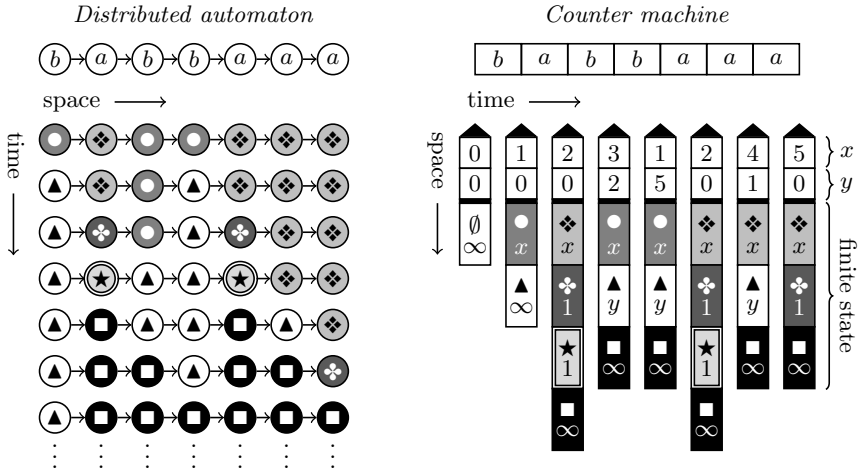


Fig. 3. Simulating a distributed automaton with a counter machine to prove Proposition 11. The left-hand side depicts the run of a quasi-acyclic distributed automaton on the $\{a, b\}$-labeled dipath that corresponds to the word *babbaaa*. This automaton has at most three loops per trace; its set of states Q consists of the states ❖, ●, ▲, ■, which have self-loops, and the states ✤, ★, which do not. On the right-hand side, the automaton is simulated by a copyless 2-counter machine whose memory configurations encode infinite sequences of states of the automaton as finite sequences of pairs in $Q \times (\mathbb{N}_+ \cup \{\infty\})$. Values different from 1 and ∞ are stored in the two counters x and y.

The crux of the proof is the following: if the i-th node remains in the same state for more than ℓ rounds, then (by quasi-acyclicity) the sequence of states traversed during that time by the $(i+1)$-th node must become constant (i.e., repeating always the same state) no later than the ℓ-th round. Thus, to compute

the entire state sequence of the $(i+1)$-th node, M does not need to know the exact numbers of state repetitions in the i-th node's sequence. It only needs to know these numbers up to threshold ℓ and be able to sum them up. □

6 Conclusion

We have now completed the proof of Theorem 1, which states the equivalence of (1) copyless and (2) sumless counter machines on finite words and (3) quasi-acyclic distributed automata on pointed dipaths. More precisely, we have established the following translatability results, which are visualized in Figure 4:

1. A copyless k-counter machine with h-access can be translated into an equivalent sumless (2^k)-counter machine with $(k \cdot h)$-access (by Proposition 7).
2. A sumless k-counter machine with h-access, can be transformed into an equivalent (sumless k-counter) machine that has merely 1-access (by Proposition 9), which in turn can be translated into an equivalent quasi-acyclic distributed automaton with at most $(k+2)$ loops per trace (by Proposition 10).
3. A quasi-acyclic distributed automaton with at most $(k+1)$ loops per trace and maximum trace length ℓ can be translated into an equivalent copyless k-counter machine with ℓ-access (by Proposition 11).

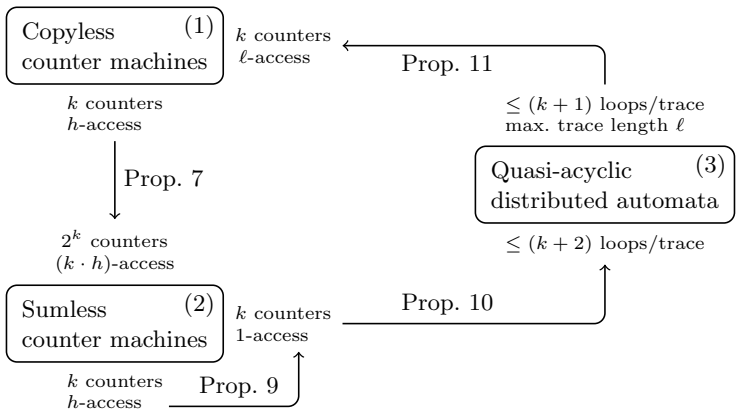


Fig. 4. The translations involved in the proof of Theorem 1.

This cycle of translations suggests that the number of counters of copyless and sumless counter machines is closely related to the maximum number of loops per trace of quasi-acyclic distributed automata. However, the precise relationship is left open. In particular, as of the time of writing, the authors do not know whether the exponential blow-up of the number of counters in Proposition 7 could be avoided.

In addition, there are several natural directions in which the present work might be extended. First of all, the models of computation concerned by Theorem 1 are special cases of two more general classes of word acceptors, namely the unrestricted counter machines of Definition 3 and the unrestricted distributed automata of Definition 5 on pointed dipaths (or equivalently, reversed one-way cellular automata). It is thus natural to ask whether our result carries over to stronger (sub)classes of counter machines and distributed automata. Instead of counter machines, one might also consider sequential machines with more freely accessible memory, such as restricted read-write tapes. Second, one could conversely try to establish similar connections for weaker classes of devices. In particular, it would be interesting to find a distributed characterization of the real-time counter machines of Fischer, Meyer, and Rosenberg [4], which are both copyless and sumless. Third, all of the models considered in this paper are one-way, in the sense that counter machines scan their input from left to right and distributed automata on dipaths send information from left to right. Hence, another obvious research direction would be to investigate the connections between (suitably defined) two-way versions. Finally, for the sake of presentational simplicity, we have only looked at deterministic models. It seems, however, that our proofs could be easily extended to cover nondeterministic or even alternating devices. We leave this open for future work.

Acknowledgments

We are grateful to the anonymous reviewers for their constructive comments. We also thank Martin Kutrib and Pierre Guillon for interesting discussions, especially concerning the connection of our results with the field of cellular automata. This work was partially supported by the ERC project EQualIS (FP7-308087) and the DeLTA project (ANR-16-CE40-0007).

References

1. Alur, R., D'Antoni, L., Deshmukh, J., Raghothaman, M., Yuan, Y.: Regular functions and cost register automata. In: LICS'13. pp. 13–22. IEEE Computer Society (2013)
2. Dyer, C.: One-way bounded cellular automata. Information and Control 44(3), 261–281 (1980)
3. Dymond, P.: Indirect addressing and the time relationships of some models of sequential computation. Computers & Mathematics with Applications 5(3), 193–209 (1979)
4. Fischer, P., Meyer, A., Rosenberg, A.: Counter machines and counter languages. Mathematical Systems Theory 2(3), 265–283 (1968)
5. Goles, E., Ollinger, N., Theyssier, G.: Introducing Freezing Cellular Automata. In: Cellular Automata and Discrete Complex Systems. TUCS Lecture Notes, vol. 24, pp. 65–73. Turku, Finland (Jun 2015), hal-id: hal-01294144
6. Hella, L., Järvisalo, M., Kuusisto, A., Laurinharju, J., Lempiäinen, T., Luosto, K., Suomela, J., Virtema, J.: Weak models of distributed computing, with connections to modal logic. Distributed Computing 28(1), 31–53 (2015)

7. Immerman, N.: Descriptive complexity. Graduate texts in computer science, Springer (1999)
8. Kari, J.: Theory of cellular automata: A survey. Theor. Comput. Sci. 334(1-3), 3–33 (2005)
9. Kutrib, M.: Cellular automata - A computational point of view. In: New Developments in Formal Languages and Applications, vol. 113, pp. 183–227. Springer (2008)
10. Kutrib, M., Malcher, A.: Cellular automata with sparse communication. Theor. Comp. Sci. 411(38-39), 3516–3526 (Aug 2010)
11. Kuusisto, A.: Modal logic and distributed message passing automata. In: CSL'13. LIPIcs, vol. 23, pp. 452–468 (2013)
12. Kuusisto, A., Reiter, F.: Emptiness problems for distributed automata. In: GandALF'17. EPTCS, vol. 256, pp. 210–222 (2017)
13. Lynch, N.: Distributed Algorithms. Morgan Kaufmann (1996)
14. Malcher, A.: Descriptional complexity of cellular automata and decidability questions. Journal of Automata, Languages and Combinatorics 7(4), 549–560 (2002)
15. Minsky, M.: Recursive unsolvability of post's problem of "tag" and other topics in theory of turing machines. Annals of Mathematics 74(3), 437–455 (1961)
16. Peleg, D.: Distributed Computing: A Locality-Sensitive Approach, SIAM Monographs on Discrete Mathematics and Applications, vol. 5. Society for Industrial and Applied Mathematics (SIAM) (2000)
17. Petersen, H.: Simulations by time-bounded counter machines. Int. J. Found. Comput. Sci. 22(2), 395–409 (2011)
18. Reiter, F.: Asynchronous distributed automata: A characterization of the modal mu-fragment. In: ICALP'17. LIPIcs, vol. 80, pp. 100:1–100:14 (2017)
19. Seidel, S.: Language recognition and the synchronization of cellular automata. Tech. Rep. 79-02, Department of Computer Science, University of Iowa (1979)
20. Terrier, V.: Language recognition by cellular automata. In: Handbk. of Nat. Comp., pp. 123–158. Springer (2012)
21. Vollmar, R.: On Cellular Automata with a Finite Number of State Changes. In: Knödel, W., Schneider, H.J. (eds.) Parallel Processes and Related Automata, vol. 3, pp. 181–191. Springer, Vienna (1981)
22. Vollmar, R.: Some remarks about the "efficiency" of polyautomata. International Journal of Theoretical Physics 21(12), 1007–1015 (Dec 1982)